# AUTONOMOUS MALWARE DECEPTION AND ORCHESTRATION

by

Md Sajidul Islam Sajid

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2023

Approved by:

Dr. Jinpeng Wei

Dr. Bei-Tseng Chu

Dr. Weichao Wang

Dr. Yasin Raja

# ABSTRACT

MD SAJIDUL ISLAM SAJID. Autonomous malware deception and orchestration.
(Under the direction of DR. JINPENG WEI)

Traditional approaches to cyber defense lack the agility to effectively counter stealthy and undetectable attacks, placing defenders at a disadvantage. In response to this imbalance, Active Cyber Deception (ACD) has emerged as a promising solution by dynamically orchestrating deceptive environments to mislead and disrupt attackers' decision-making processes. However, developing efficient and effective deception systems necessitates the integration of human intelligence and comprehensive malware analysis to comprehend attack behaviors and automate deception strategies.

This dissertation presents three innovative approaches in the field of ACD. Firstly, DodgeTron combines dynamic analysis using symbolic execution tools and machine learning to automate the creation of deception schemes against malware. It achieves this by categorizing malware into known families and employing HoneyThings. Secondly, symbSODA performs dynamic analysis on real-world malware and conducts data flow analysis to extract malicious sub-graphs (MSGs). These MSGs are then mapped to the MITRE ATT&CK framework using Natural Language Processing, enabling the creation of a Deception Playbook for deceiving specific malicious behaviors through deceptive API hookings. Finally, ranDecepter integrates active cyber deception to identify ransomware in its early stages and utilizes binary reset (orchestration) methods to repurpose the malware to exhaustively transmit encryption information (including keys) to the attacker, thereby effectively depleting their available resources.

Comprehensive evaluations validate the accuracy and effectiveness of these approaches in deceiving adversaries, reducing analysis time, and mitigating malware threats. This research significantly contributes to the field of active cyber deception and offers efficient and scalable solutions for safeguarding digital systems against

sophisticated attacks.

# DEDICATION

I take this opportunity to dedicate this achievement to my dear elder sister, Sajia, who has been a constant pillar of support in my life, much like a mother. I am deeply grateful to her for the significant role she has played in shaping my journey and accomplishments.

# ACKNOWLEDGEMENTS

I would like to extend my heartfelt gratitude to my advisor, Dr. Jinpeng Wei, for his unwavering support, patience, and encouragement throughout my doctoral studies. Additionally, I am deeply grateful to my co-advisor, Dr. Ehab Al-Shaer, for their valuable guidance. My sincere appreciation also goes to the members of my dissertation committee: Dr. Weichao Wang, Dr. Bill Chu, and Dr. Yasin Raja. Their insightful thoughts, comments, and suggestions were instrumental in shaping my research. I would like to take a moment to extend a special acknowledgment to my beloved wife, Jeba, whose unwavering support has been a pillar of strength throughout all the highs and lows of my journey. Her boundless love and encouragement have been instrumental in keeping me motivated, especially during the most challenging times. I firmly believe that this achievement is as much hers as it is mine. Additionally, I am deeply grateful to my mother for her constant prayers and caring presence, which have provided me with the inner strength needed to persevere. I would also like to express my gratitude to my lab mates, who have offered invaluable assistance with their suggestions and contributions to writing papers. This achievement truly belongs to all of you who believed in me and supported me wholeheartedly along the way. Your unwavering encouragement has been a driving force behind my success, and for that, I am profoundly thankful.

TABLE OF CONTENTS

LIST OF TABLES xi

LIST OF FIGURES xiii

CHAPTER 1: Introduction 1

1.1. Objectives 2

1.2. Contributions and Dissertation Structure 4

1.3. A Comparative Analysis of DodgeTron, symbSODA, and ran-Decepter 6

CHAPTER 2: DodgeTron: Towards Autonomous Cyber Deception Using Dynamic Hybrid Analysis of Malware 9

2.1. Introduction 9

2.2. on Background on Information Stealers 12

2.3. DodgeTron: Approach Overview 13

2.4. Malware Deception Playbooks Construction (Offline) phase 14

2.4.1. Malware Categorization 14

2.4.2. Playbook Creation 17

2.5. Dynamic Deception Scheme Creation (online) phase 21

2.5.1. Detection Agent in the online phase 21

2.5.2. Analysis Agent in the online phase 22

2.5.3. Planning Agent in the online phase 23

2.5.4. Actuating Agent in the online phase 23

2.6. Implementation and Evaluations 24

2.6.1. Dataset 24

2.6.2. Experiment I: Evaluating the precision of DodgeTron in extracting deception parameters 25

2.6.3. Experiment II: Evaluating the accuracy of DodgeTron in the online phase 27

2.6.4. Performance analysis of DodgeTron in terms of execution time 29

2.6.5. Performance analysis of our classifier 29

2.7. Related Work 31

2.8. Discussion & Conclusion 33

CHAPTER 3: symbSODA: Configurable and Verifiable Orchestration Automation for Active Malware Deception 34

3.1. Introduction 34

3.2. Threat Model and Assumptions 38

3.3. Deception Playbook Creation 39

3.3.1. Malicious Sub-graphs (MSG) Extraction 40

3.3.2. MSG Classifier 48

3.3.3. Deception Factory Synthesis 54

3.4. Real-time Orchestration 67

3.5. Evaluations 71

3.5.1. Case studies 72

3.5.2. Evaluation of MSG extraction 75

3.5.3. Comparison with other state-of-the-art tools in terms of discovering malware behaviors/capabilities 77

3.5.4. Comparison with existing Sandboxes 77

3.5.5. MSG Classifier Evaluation 79

3.5.6. Performance Analysis of symbSODA 83

3.5.7. End-to-End Accuracy of symbSODA 86

3.6. Related Work 88

3.7. Discussion and Conclusion 91

CHAPTER 4: ranDecepter: Empowering Defense Against Ransomware Attacks through Active Cyber Deception and Binary Reset 93

4.1. Introduction 93

4.2. Background 97

4.2.1. Ransomware and it's behaviors 97

4.2.2. Malicious Sub-graphs (MSGs) 98

4.2.3. Deception Strategies 99

4.3. Threat Model and Assumptions 100

4.4. System overview 101

4.4.1. Offline Phase: Deception DLL Creation 102

4.4.2. Realtime Phase: Ransomware Detection using Embedded Deception (API) Hooks 106

4.4.3. Reset Phase: Exhausting attackers' resources by repeatedly initiating malware through binary orchestration 111

4.5. Evaluations 115

4.5.1. Dataset 115

4.5.2. Evaluation of Accuracy and Effectiveness against Ransomware 115

4.5.3. Evaluation against the Benign Applications 117

4.5.4. Accuracy and Performance Analysis of Binary Orchestration in the Reset Phase 119

4.6. Related work 123

4.6.1. Non-Deception Based Ransomware Detection 123

4.6.2. Deception-Based Ransomware Detection 124

4.7. Discussion & Conclusion 125

CHAPTER 5: Conclusion and Future Work 127

5.1. Conclusion 127

5.2. Limitations and Future Work 129

REFERENCES 131

# LIST OF TABLES

TABLE 1.1: A Comparison of Design Choices for Autonomous Cyber Deception Frameworks - DodgeTron, symbSODA, and ranDecepter. 7

TABLE 3.1: Deception ploy creation and verification ($D_1$ = diversion, $D_2$ = distortion, $D_3$ = depletion, $D_4$ = discovery). 56

TABLE 3.2: List of the Pre and Post conditions for each API(selective cases, the actual table consists of all possible combinations) 62

TABLE 3.3: Comparison with other State-of-the-art tools in terms of discovering malware behaviors/capabilities using GT1 and their individual recall values 77

TABLE 3.4: Comparison of symbSODA with other related tools in terms of detecting different MITRE ATT&CK techniques within malware execution. 78

TABLE 3.5: Number of techniques (T) and procedures (P) discovered by symbSODA compared to Cuckoo sandbox, Any.run and SODA 79

TABLE 3.6: MSG Classifier Parameters 81

TABLE 3.7: Top-n Accuracy of MSG Classifier 81

TABLE 3.8: Top-n accuracies after analysis as well as excluding StackOverflow and enriching component. 82

TABLE 3.9: Malware deception overhead ($T_1$ = time without deception, $T_2$ = time with symbSODA deception, O = Overhead). 84

TABLE 4.1: Deception ploy planning for ransomware (O=original file, D=Destination file, Pwd= password, S=session key, CO=content of the original file, P= public key). N.B. FileHeader writing is optional, hence noted in italic font. 104

TABLE 4.2: Ransomware dataset 116

TABLE 4.3: Ransomware detecting stages and the required time comparison with and without our system. 116

TABLE 4.4: Comparison of false positives using benign applications to test our system. The table displays the presence of ransomware-like behavior and relevant MSGs indicated by a checkmark (✓). The abbreviations CF, ENC, FD, and RN represent CreateFile with known ransomware extensions, Encryption, File deletion/overwrite, and Ransom note creation, respectively. The last two columns compare the response delay for the same task conducted without and with our system, with the increment (INC) of response time provided in the last column within brackets. 119

TABLE 4.5: Comparison of Time Between Binary Orchestration/Reset and VM Reset Approaches 120

TABLE 4.6: Resource Depletion at the Attacker's End: Extra Database Entries Generated by Binary Orchestration and the Additional Space Required to Store Them 122

# LIST OF FIGURES

FIGURE 2.1: Workflow of DodgeTron in the Malware Deception Playbooks Construction (Offline) phase 15

FIGURE 2.2: Architecture of Cluster Analyzer 16

FIGURE 2.3: Potential Candidate APIs for DPE 19

FIGURE 2.4: Workflow of DodgeTron during the Dynamic Deception Scheme Creation (online) phase 22

FIGURE 2.5: Precision of DodgeTron across different malware families in term of deception parameter extraction 26

FIGURE 2.6: Overall accuracy of DodgeTron across different malware families in term of deceiving the malware 28

FIGURE 2.7: Time Comparison of *deep analysis* vs *light analysis* across different malware families and the speedup factor 30

FIGURE 2.8: The trade-off between homogeneity score and number of clusters in hierarchy 31

FIGURE 3.1: Deception Playbook Creation 41

FIGURE 3.2: API Call Tracer: (a) Our implemented approach automates the template code generation process, (b) The Hooking Engine is responsible for intercepting API calls, and (c) The Symbolic Execution Engine is responsible of executing the malware symbolically. (Rectangular shapes indicate agents (scripts) capable of processing, while parallelograms indicate I/O data for respective agents. 43

FIGURE 3.3: Execution trace of a malware sample generated in Cuckoo Sandbox (N.B. The size of the parameter“buffer" is large hence denoted as “FileContent" instead of the original content) 44

FIGURE 3.4: MSGs generated from malware traces 44

FIGURE 3.5: An MSG that corresponds to malware actions in a real-world attack scenario 47

FIGURE 3.6: MITRE techniques and APIs vector representation extraction. 49

FIGURE 3.7: MSG to MITRE technique classification 49

FIGURE 3.8: Call flow with and without API Hooking. In some case, Response is return via the original API (5a, 5b), otherwise, Detour function responds directly (5). 65

FIGURE 3.9: A code snippet: How the deception technique is implemented inside API hooks 66

FIGURE 3.10: Communication among symbSODA's different segments, agents and the victim under attack. 68

FIGURE 3.11: User interface to select deception actions for different malware behavior. 69

FIGURE 3.12: The process of identifying conflicting deception actions and disabling them on the interface level 70

FIGURE 3.13: Malware source code: evasion check 72

FIGURE 3.14: symbSODA deployment time with different ploys 85

FIGURE 3.15: Avg time of a single OES to orchestrate and serve multiple OECs 85

FIGURE 3.16: Assume Guarantee Verification Time 85

FIGURE 3.17: Accuracy of symbSODA across different malware types 85

FIGURE 4.1: Malware execution trace to MSG conversion 99

FIGURE 4.2: Overall system and data flow 101

FIGURE 4.3: Detailed illustration showcasing the various phases and components of our system, depicting their data flow and decision-making processes. 106

FIGURE 4.4: This figure shows the identified execution chains (EC1 and EC2). On the left side, we presented the MSGs ransomware use to perform "Transfer Key(s)". Red colored APIs indicate the end of their respective MSG. 112

FIGURE 4.5: Workflow of binary orchestration. 113

FIGURE 4.6: Our system integrates deception mechanisms during the Staging and Encryption stages to identify and neutralize/defuse ransomware encryption within its kill chain 122

# CHAPTER 1: Introduction

The field of cybersecurity faces a significant challenge due to the lack of agility in defending against cyber threats. Attackers often have the upper hand in discovering targets and planning their attacks covertly, exploiting the limitations of existing security measures. While extensive research has been conducted on detecting and predicting attacks, adversaries continuously adapt by scanning networks, learning about countermeasures, and developing new evasion techniques. This dynamic nature of cyber warfare demands a proactive and adaptable approach to counter these threats effectively.

Active Cyber Deception (ACD) has emerged as effective means to reverse this asymmetry in cyber warfare by dynamically orchestrating the cyber deception environment to mislead attackers and corrupting their decision-making process. However, the development of efficient active deception systems has relied heavily on manual analysis and human intelligence to understand attackers' behaviors based on malware actions. This manual approach significantly limits the ability of cyber deception to respond promptly to new threats, hindering its effectiveness.

Additionally, existing deception approaches suffer from limitations such as lack of agility, robustness, and automation. These approaches often rely on static deployment and configurations, making them easily distinguishable [1] from real systems by skilled attackers [2–5]. To overcome these limitations, a systematic and automated approach is required to identify the environmental values that need to be altered to deceive malware effectively.

Furthermore, existing deception techniques primarily focus on thwarting attackers at specific stages of the kill chain. For example, some well-known deception tech-

niques use honeypots [1, 6, 7], honeyfiles [8–10], honeypatches [11], honeybugs [12], decoys [13] and moving target defense [14–16] to mislead attackers during the collection and credential access phase. On the other hand, others employ malicious traffic redirection [17] and network randomization [18] to deter attackers during the command and control phase. However, only a few approaches consider every phase of the kill chain [13, 19, 20], and they often lack the ability to provide customized deception plans to users based on their specific requirements. Network randomization [18] and moving target defense [14–16]. This limitation calls for the development of comprehensive deception strategies that can adapt to different attack scenarios and provide tailored deception tactics.

## 1.1 Objectives

The objective of this dissertation is to introduce an autonomous cyber deception system that addresses the limitations mentioned earlier. The system combines dynamic malware analysis and symbolic execution techniques to identify and extract deception features from malware execution traces. These features include deception parameters related to system resources (files and registries), sequences of API calls associated with malicious behaviors, and critical addresses within the malware that can be exploited to transform it into a communication channel for continuously providing misleading information to attackers. The dissertation aims to provide deception as a service through three different approaches: orchestrating a deceptive environment using HoneyThings, orchestrating API responses, or orchestrating the malware binary itself. The specific objectives of this dissertation are as follows:

- Develop a system that combines symbolic execution with dynamic malware analysis to automate the extraction of deception parameters from malware execution traces. This system aims to identify target systems and disrupt attackers' objectives by dynamically orchestrating the cyber deception environment.

- Create a highly configurable and verifiable autonomous cyber deception system that analyzes real-world malware, identifies API patterns representing attack techniques, and constructs conflict-free Deception Playbooks for effective real-time deception orchestration.

- Introduce a deception-based ransomware detection approach that operates at the API level, eliminating the need for managing decoy files. This approach should extract critical addresses within ransomware binaries to reset the encryption process, effectively transmitting encryption information and keys back to the attacker repeatedly.

To achieve these objectives, three deception approaches are proposed and implemented in this dissertation:

- DodgeTron: An autonomous cyber deception framework that combines dynamic analysis, symbolic execution, and machine learning to automate the creation of deception schemes using HoneyThings against malware.

- symbSODA: A dynamic security orchestration, automation, and deception system that leverages symbolic execution and data flow analysis to extract comprehensive malicious sub-graphs (MSGs) from malware execution traces. MSGs are mapped to the MITRE ATT&CK framework to determine malware behaviors and activate relevant deception ploys.

- ranDecepter: A ransomware detection and mitigation approach that utilizes API-level deception and binary orchestration. It identifies critical addresses within ransomware binaries to establish a looping mechanism, depleting the attacker's resources and mitigating the impact of ransomware attacks.

## 1.2 Contributions and Dissertation Structure

This dissertation consists of three main contributions, each presented in a separate chapter. The contributions and the corresponding structure of the dissertation are as follows:

### Chapter 2: DodgeTron

- **Contributions:** The proposal and implementation of DodgeTron, an autonomous cyber deception framework that combines dynamic analysis, symbolic execution tools, and machine learning to create real-time deception schemes using HoneyThings against malware.

- **Descriptions:** DodgeTron performs deep analysis on malware using selective symbolic execution, gathering malware traces through multipath exploration. Clustering techniques are then used to categorize malware into known families, identifying key representative samples. Deception-oriented parameter extraction is conducted on these key representatives, generating a deception playbook. HoneyThings, such as honey-registry, honey-files, and fake configurations, are generated and stored in the Deception Playbook, mapped to the key representative. When new malware is detected, the system classifies it into a known family and utilizes the Deception Playbook and mapping to deploy necessary HoneyThings for deceiving the malware.

- **Evaluations:** The accuracy of DodgeTron is evaluated with 953 recent malware samples, achieving an average accuracy of 91.18%. The analysis time is optimized by 1.1x to 2.8x.

### Chapter 3: symbSODA

- **Contributions:** The proposal and implementation of symbSODA, an autonomous cyber deception system that uses symbolic execution and API hooking

to extract malicious sub-graphs, map them to the MITRE ATT&CK framework, and create a conflict-free deception playbook to detect and deceive unknown malware.

- **Descriptions:** symbSODA extracts comprehensive malicious sub-graphs (MSGs) from malware execution traces and maps them to the MITRE ATT&CK framework. This knowledge base is used to create the Deception Playbook, which contains deception course-of-actions for specific malicious behaviors. At runtime, symbSODA detects MSGs in unknown malware and executes deception ploys through embedded deceptive API hooks. It includes a Deception Planning Verifier to ensure consistent and conflict-free deception actions.

- **Evaluations:** symbSODA is evaluated using recent malware, achieving a high accuracy of 95% in deceiving malware with minimal overhead and deployment time. It demonstrates a 97% recall value in MSG extraction and an 88.75% top-1 accuracy in MSG-to-MITRE mapping.

Chapter 4: ranDecepter

- **Contributions:** This chapter presents the proposal and implementation of ranDecepter, an innovative API-level deception solution that detects ransomware using API-level deception. It depletes attacker resources by employing binary reset (orchestration) to establish a looping mechanism that continuously transmits encryption information (including keys) back to the attacker.

- **Descriptions:** ranDecepter leverages the extracted malicious sub-graphs (MSGs) obtained from symbSODA and employs API-level deception without making any modifications to the original file system. This approach effectively determines ransomware during its lifecycle without compromising sensitive resources. Furthermore, it automates the identification of critical addresses within ransomware

binaries, enabling the establishment of a looping mechanism. By restarting the ransomware from the beginning, this mechanism transmits encryption information and keys back to the attacker, effectively depleting their resources. ranDecepter eliminates the need for managing and distributing decoy files, making it scalable and cost-effective.

- **Evaluations:** ranDecepter is extensively evaluated using 15 real-world malware samples and 8 benign applications. The evaluation results demonstrate the outstanding performance of ranDecepter. It achieves 100% accuracy in ransomware detection without any false positives and minimal impact on response time. Additionally, ranDecepter surpasses the VM reset approach in binary orchestration, delivering significant time savings. The evaluation results indicate that ranDecepter can generate a maximum of 8,513 entries, with a minimum of 2,532 entries, in the attacker's database within a 24-hour period, effectively depleting their resources.

## 1.3 A Comparative Analysis of DodgeTron, symbSODA, and ranDecepter

DodgeTron, symbSODA, and ranDecepter are three distinct approaches within the field of active cyber deception. To comprehensively analyze the similarities and differences among these approaches, it is essential to understand the four deception strategies introduced in this dissertation: FakeFailure (FF), FakeSuccess (FS), FakeExecute (FE), and NativeExecute (NE). These strategies dictate our response when encountering malware. FakeFailure simulates a failed operation, FakeSuccess simulates a successful operation with static content, FakeExecute remotely performs the malware's actions, and NativeExecute allows the malware to run and observe its behavior. Each strategy presents a unique approach to deceiving attackers and disrupting their intentions. Despite their differences, these approaches share some common elements. As integral components of the autonomous cyber deception framework,

Table 1.1: A Comparison of Design Choices for Autonomous Cyber Deception Frameworks - DodgeTron, symbSODA, and ranDecepter.

| **Design choices** | **DodgeTron** | **symbSODA** | **ranDecepter** |
|---|---|---|---|
| **Target** | Autonomous Cyber Deception and Orchestration | | |
| **Target malware type** | InfoStealer | RAT, InfoStealer, Spyware, Ransomware | Ransomware |
| **4D Deception goals** | Diversion, Depletion, Discovery | Diversion, Depletion, Discovery and Distortion | Diversion, Depletion |
| **Deception features** | Deception parameter (File and Register level) | MSG, MSG2MITRE and API Hooking (API level) | Critical API detection to fake response (API level) Critical addresses (Binary level) |

these approaches focus on delivering malware deception and orchestration capabilities. However, their distinctive strengths lie in targeting diverse malware families, accomplishing specific deception goals, and leveraging a wide array of deception features. A concise summary of their design choices can be found in Table 1.1.

In terms of their techniques and approaches, DodgeTron extracts deception parameters from malware execution traces, clusters malware into known families, and deploys HoneyThings to create a deceptive environment. By running the malware within this environment, DodgeTron utilizes the FakeExecute strategy to misinform the malware about its actions and goals.

In contrast, symbSODA leverages symbolic execution and data flow analysis. It extracts malicious sub-graphs from malware execution traces and maps them to the MITRE ATT&CK framework to understand the high-level behavior of the malware. symbSODA creates a deception playbook that includes various ploys tailored to each sub-graph, employing different deception goals and strategies. During runtime, symbSODA detects malware behavior using the extracted sub-graphs, understands malware behavior using the MSG-to-MITRE mapping and executes the corresponding deception ploys through embedded deceptive API hooks. This flexibility allows symbSODA to offer multiple deception modes, including FakeFailure, FakeSuccess, FakeExecute, and TrueScenario. If the FakeExecute strategy is chosen, symbSODA can use DodgeTron for environment orchestration to perform remote execution.

ranDecepter, on the other hand, focuses on improving deception-based ransomware detection approaches by utilizing API-level deception instead of file-level deception.

It builds upon the MSGs obtained from symbSODA and employs API-level deception with FakeSuccess as the chosen strategy. By monitoring the program's runtime behavior without modifying the original file system, ranDecepter can detect ransomware at its early stages without compromising the integrity of the original files. Additionally, ranDecepter introduces a novel orchestration method called "Binary reset," which allows the malware to be reset to its initial state without restarting or modifying the underlying environment (Host). This approach offers the advantage of scalability and cost-effectiveness since it eliminates the need for managing and distributing decoy files.

In conclusion, while DodgeTron, symbSODA, and ranDecepter share the common goal of active cyber deception and utilize different deception strategies, they differ in their techniques, focuses, deployment methods, and scalability. DodgeTron emphasizes environment orchestration, symbSODA leverages symbolic execution and mapping to the MITRE ATT&CK framework, and ranDecepter places emphasis on API-level deception and introduces a unique orchestration method through “Binary reset."

The dissertation is based upon the following publications.

- Sajid, Md Sajidul Islam, et al. "Dodgetron: Towards autonomous cyber deception using dynamic hybrid analysis of malware." 2020 IEEE Conference on Communications and Network Security (CNS). 2020.

- Sajid, Md Sajidul Islam, et al. "SODA: A system for cyber deception orchestration and automation." Annual Computer Security Applications Conference (ACSAC). 2021.

- Sajid, Md Sajidul Islam, et al. "symbSODA: Configurable and Verifiable Orchestration Automation for Active Malware Deception." ACM Transactions on Privacy and Security (TOPS). 2023 (Minor revision).

# CHAPTER 2: DodgeTron: Towards Autonomous Cyber Deception Using Dynamic Hybrid Analysis of Malware

## 2.1 Introduction

Active cyber deception (ACD) has emerged as an effective and complementary defense technique to overcome the challenges faced by traditional detection and prevention strategies. The basic idea of cyber deception is to deliberately introduce misinformation or misleading functionality into cyber space, which tricks adversaries in such a way that their attacks become ineffective or infeasible. An effective ACD mechanism can achieve 4D goals: (1) *deflect* adversaries to false targets, (2) *distort* adversaries' perception about the environment, (3) *deplete* adversaries' resources and (4) *discover* adversaries' motives, tactics and techniques [21].

Although cyber deception has been successfully applied in numerous settings [7, 11, 22–31], existing deception techniques lack agility, resilience and automation. The best-known technique to achieve deception is honeypot. However, it has complex configuration issues and lacks randomness, which makes it distinguishable from real systems [32]. Other well-known deception techniques are network randomization [18] and moving target defense [14] [33]. Although these techniques have been successful in deceiving the adversaries, they were designed in an ad-hoc manner to counter *specific attacks* and they heavily relied on *manual* analysis and decision making. This manual process is time-consuming and not scalable against the fast evolving nature of cyber attacks. Therefore, we need an automated approach capable of decision making through reasoning in real-time.

In this chapter, we present an autonomous cyber deception framework, called DodgeTron, that automatically analyzes malware and creates cyber deception schemes.

We make an important observation that malware often interacts with the victim system to determine the configuration (e.g., keyboard layout and IP address) and valuable information (e.g., the files) of the environment in order to reach its goals. Therefore, we can manipulate such interactions to deceive the adversary behind the malware. For example, to deceive malware that steals login credentials stored in the file system of the victim environment, we can plant files with honey credentials before the malware runs. We call such values (e.g., keyboard layout and credential files), which are necessary conditions for the success of attacks but are also configurable or misrepresentable by the environment, *deception parameters*.

DodgeTron aims to have two desirable features: (1) *autonomous*, meaning that it can take a piece of unknown malware and orchestrate and execute an effective deception scheme based on that malware without any human intervention; (2) *practical*, meaning that the whole process can be finished within a reasonable amount of time.

To achieve the design goals, DodgeTron combines hybrid dynamic analysis and machine learning. It performs deep dynamic analysis on known malware samples, uses clustering of the execution logs to identify key representative samples and then constructs a Deception Playbook that will be used later to deceive similar malware. During deep analysis, DodgeTron executes malware samples using symbolic execution that facilitates analyzing malware's interactions with the system through API (system/library) calls. Such API call analysis utilizes multipath exploration to assist DodgeTron in choosing feasible and cost-effective deception candidates. As these candidates are associated with different accuracy and cost, DodgeTron solves an optimization problem to find out the most cost-effective candidate sets. Finally, the chosen candidates (i.e., the deception parameters) are used to create deception ploys, which are stored in a knowledge base called Deception Playbook. Because symbolic execution is often slow and expensive, DodgeTron adopts a hybrid approach: it performs deep analysis and learning of the knowledge base only offline; during runtime,

it performs light-weight analysis on new malware samples and uses the execution logs to map (classify) the samples to deception ploys. Our evaluation result shows that this hybrid approach gives effective deception (e.g., 91.18% success rate) and good performance (e.g., 1.1x to 2.8x speedup).

We note that DodgeTron is based on our previous work called gExtactor [34], which performs deep analysis on malware samples to derive deception parameters. However, gExtractor is too slow to support real time deception. Therefore, we add machine learning and the hybrid dynamic analysis in this approach to overcome the limitations of gExtractor.

To evaluate DodgeTron, this dissertation focuses on one major type of malware, information stealers, or simply InfoStealers. Such malware seeks sensitive and personal information such as credit card numbers, cryptocurrency wallets, browser data and email credentials. They come in many families (e.g., Emotet, Zbot, Zeus, Raccoon, Khalesi and LokiBot) and have caused serious financial loss affecting hundreds of millions of people around the world. The InfoStealers are good candidates for deception because we can feed them misinformation (such as honey passwords) to lure the adversary to monitored networks. The challenge is how to automatically find out what kind of misinformation to feed. We have experimentally confirmed the feasibility of deceiving such InfoStealers. Specifically, we make the following contributions:

- We propose a deception-oriented autonomous framework named DodgeTron that is capable of creating deception schemes against InfoStealers through systematic binary-level analysis and reasoning in real-time.

- We design a hybrid and machine learning-based approach that uses deep analysis to ensure effectiveness and quick classification to achieve real-time deception.

- We evaluated DodgeTron with 953 recent InfoStealers, which demonstrates the ability of DodgeTron to generate automated deception schemes against malware.

The remainder of the chapter is organized as follows: Section 4.2 provides background on information stealers. Section 4.3 gives an overview of our approach. Section 4.4 presents how *deception playbooks* are created offline. Section 4.5 presents how we create *Online Deception Schemes*. Section 4.6 provides implementation details and evaluation results. Related work is discussed in Section 4.7. Finally, limitations, future work and conclusions are presented in Section 2.8.

## 2.2 on Background on Information Stealers

Information stealers are a type of malicious software (malware) designed to infiltrate systems and surreptitiously extract sensitive and valuable information. These sophisticated cyber threats pose a significant risk to individuals, organizations, and even governments, as they can lead to data breaches, financial losses, and reputational damage. Understanding the capabilities and operations of information stealers is crucial for developing effective defense mechanisms and mitigating their impact.

Capabilities of Information Stealers:

- Data Harvesting: Information stealers are adept at collecting various types of data from compromised systems. This can include login credentials, banking details, personal identification information (PII), intellectual property, and sensitive corporate information. They employ techniques such as keylogging, clipboard monitoring, screen capturing, and browser session hijacking to obtain this data.

- Communication and Exfiltration: Once information stealers have gathered the desired data, they need a means to communicate with their operators and exfiltrate the stolen information. They establish command-and-control (C&C) infrastructure, often using covert channels and encrypted communication protocols, to transmit the stolen data securely to remote servers under the control of threat actors.

## 2.3 DodgeTron: Approach Overview

The primary goal of DodgeTron is to design automated deception schemes against InfoStealers in general and mislead adversary by presenting falsified data. To achieve it, we need to analyze malware thoroughly, extract deception parameters, design a deception scheme based on extracted deception parameters and execute the deception scheme. We divide DodgeTron into four agents (*Detection agent, Analysis agent, Planning agent and Actuating agent*) based on their activities and purposes. These four agents operate in two phases: (1) *Malware Deception Playbook Construction phase* and (2) *Dynamic of Deception Scheme Creation phase*.

*Malware Deception Playbook Construction* phase is also knows as the *offline phase*. The goal of the *offline* phase (Fig 2.1) is to create the *Deception Playbook*, a knowledge-base that stores optimal candidates to design effective deception schemes against malware having specific set of behaviors. In this phase, the *analysis agent* performs symbolic execution on the malware samples and collects execution traces in the form of logs. This specific execution is called *deep analysis*. The reason for choosing symbolic execution is that it supports multi-path exploration in a malicious program; thus facilitate us understanding malicious behaviors closely. We train our clustering model with the collected execution traces which supports two basic actions: (1) it automatically generates clusters (groups) indicating samples containing similar behaviors. (2) it identifies key representative samples from the cluster. To create deception playbook, the *analysis agent* performs deception-oriented parameter extraction only from these key representatives. Each of such extracted parameter is a candidate for designing deception schemes. However, it is not feasible to consider all of these candidates. Therefore, the *analysis agent* selects only a subset of the deception candidates based on feasibility and cost. Next, we generate HoneyThings (honey-registry, honey-files and fake configurations) for these selected candidatesstore them into the *Deception Playbook* by keeping a mapping to the key representative . At this point, DodgeTron

is equipped to create deception schemes for new malware samples.

*Dynamic of Deception Scheme Creation* phase is also known as the online phase. In the *online* phase (Fig 2.4), the *detection agent* detects malware in a system and forwards it to the *analysis agent* to perform *light analysis*. As the *deep analysis* is costly and we need to design the deception scheme for the new malware sample within a time constraint, we leverage *light analysis* to save time. We have already created our clustering model in the previous phase. Now, on the arrival of new malware instance (zero-day), *analysis agent* classifies the unknown malware (zero-day) to the known clusters of malware and identifies the key representative sample that depicts this new malware. Next, the *planning agent* identifies HoneyThings associated with the selected key representative from *Deception Playbook*. Afterward, the *planning agent* creates actuation tasks that are specific instructions about how to implant HoneyThings in order to present falsified data. Finally, the *actuating agent* performs these actuation tasks and configures other relevant settings.

## 2.4 Malware Deception Playbooks Construction (Offline) phase

During the *malware deception playbooks construction* phase, DodgeTron performs *deep analysis* to collect execution traces. We train our clustering model with these traces for finding key representative samples. DodgeTron derives *Deception Playbook* from these key representative samples that will be used later on to deceive similar malware. The trained clustering model facilitates in classifying new malware in the next phase. Fig 2.1 shows the workflow of DodgeTron in the *offline* phase.

### 2.4.1 Malware Categorization

The *analysis agent* consists of a host machine that includes emulation and monitoring tools and a guest machine that executes malware. We instrument the guest machine and execute user-provided scripts from the host machine. These scripts are written to intercept Windows API function calls. These scripts are called annotation

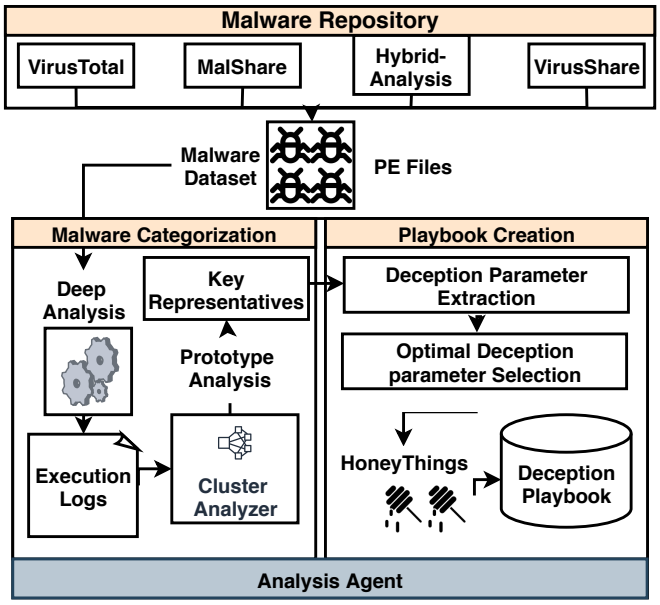



Figure 2.1: Workflow of DodgeTron in the Malware Deception Playbooks Construction (Offline) phase

plugins, which provide monitoring control over the guest machine where malware are executed. In the offline phase, the *analysis agent* performs two significant tasks. First, it performs the *deep analysis* on the malware samples to train the cluster model analyzer; and the trained model will be utilized later on to classify zero-day malware samples. Second, *the analysis agent* groups malware samples into clusters and identifies key representative samples for the clusters.

**Deep Analysis** The primary goal of *deep analysis* is to perform an in-depth and multi-path dynamic analysis on the malware samples. The deep analysis module is built on top of a selective symbolic execution engine [35]. In *deep analysis*, our annotation plugins can intercept and change the original API calls with both concrete and symbolic parameter values. This mechanism facilitates us in collecting execution traces on both sides of the conditional branching. For example, if the malware is searching for a specific file, conditional branching would go either to the succeeded branch (if the file exists) or to the failed branch (if the file does not exist), but not both. As *deep analysis* employs symbolic execution, it can traverse both sides of the branching by assigning symbolic values to the conditional variables during

the analysis. Thus, *deep analysis* explores multi-paths within the malware, which is crucial for observing more about the malware's behaviors. After completion of the *deep analysis*, the execution traces are forwarded to our cluster analyzer for training and locating key representative samples.

**Cluster Analyzer** In the offline phase, the cluster analyzer takes the execution traces as inputs and presents the key representative samples. The cluster analyzer comprises four components, as depicted in Fig 2.2.

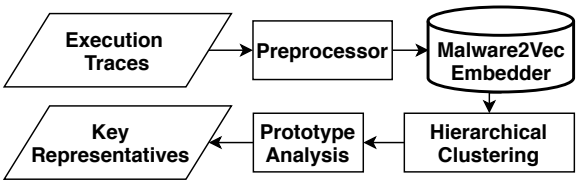



Figure 2.2: Architecture of Cluster Analyzer

• **Preprocessor:** Each execution trace is converted into an API trace vector by appending the API names and their associated parameters to a vector. Given a trace T with a sequence of APIs $T = P_1, P_2, ..., P_n$ and their associated parameters $Y = y_1, y_2, ..., y_n$ where $y_i = \{\rho_{i,1}, \rho_{i,2}, ..., \rho_{i,m}\}$ represents the set of parameters associated with $P_i$. Each parameter in $y_i$ has an index, however, not all APIs have the same number of parameters. Hence, to create an identical vector, we use padding to make the lengths similar. The final vector generated by the preprocessor includes both APIs and their parameters for each execution trace and can be represented as:

$$V = [P_{1(}\rho_{1,1}, \rho_{1,2}, ..., \rho_{1,m)}, P_{2(}\rho_{2,1}, \rho_{2,2}, ..., \rho_{2,m)}, ..., P_{n(}\rho_{n,1}, \rho_{n,2}, ..., \rho_{n,m)}] \qquad (2.1)$$

• **MalTrace2Vec Embedder:** It takes the API trace vectors and builds word embedding vector representation (feature vector) for every malware family. MalTrace2Vec is built upon Doc2Vec [36], which is an extended version of Word2Vec [37]. Word2Vec is being widely used for text analytics and natural language processing to find similarity between words, sentences and documents. It is an unsupervised framework that

leverages neural networks to learn continuous distributed vector representation for a piece of text. Our approach takes sequences of API calls (using a sliding window of size $h$) and computes the embedding vector, which preserves the order of the APIs recorded during the execution. We use Distributed Memory (DM) to calculate the probability of the next word given context and the document ID:

$$DM = P(doc\_id, w_{i-h}, ..., w_i | w_{i+1}). \tag{2.2}$$

where $doc\_id$ represents the class (family or sub-family), $w$ denotes the words in the corpus, and $h$ represents the size of the sliding window.

• **Hierarchical Clustering:** This component takes the family embedding vectors generated by MalTrace2Vec and represents them as a tree (or dendrogram) with respect to their similarities. Since the families are categorized under one type of malware, they are likely to have very similar behaviors and similar traces with minor variations. The agglomerative *hierarchical clustering technique* enables us to append the most similar traces to larger groups. To this end, every trace vector is considered as a cluster. Next, we compute the distance matrix of each using the cosine similarity metric. Similar groups are merged and formed as a single cluster. It keeps merging two clusters until only one cluster persists.

• **Prototype Analysis** This component performs analysis on the hierarchically represented clusters and returns key representatives for each cluster. To this end, sampler takes each cluster as a mutually exclusive distribution and return a portion of data points from each randomly.

### 2.4.2 Playbook Creation

At this stage, DodgeTron has already identified the key representative samples and their traces. These traces are precisely related to malicious behaviors. *Deception Playbook* is derived from these traces. The *analysis agent* performs three major

tasks during this phase: *deception parameter extraction*, *optimal deception parameter selection* and *deception playbook creation*. During *deception parameter extraction*, the *analysis agent* performs deception-oriented analysis on the execution logs to extract all possible deception candidates. It is not feasible to design deception schemes by considering all of these deception candidates. Therefore, the *analysis agent* performs *optimal deception parameter selection* to select only a few deception candidates based on feasibility, optimization and cost-effectiveness. Finally the *analysis agent* creates *Deception Playbook* with these optimal deception candidates.

**Deception Parameter Extraction (DPE)** All the collected system parameters from the *deep analysis* are not crucial in designing deception. Therefore, the *analysis agent* extracts parameters based on the following three criteria (T1, T2, T3) that are significant in designing deception schemes:

- **Completeness for Resilience: (T1):** We extract deception parameters from all paths that lead to the malicious goal in order to construct resilient coverage of all behaviors. For example, one of the behaviors of InfoStealers is to perform file-scanning. Therefore, All the APIs related to scanning should be considered. Malware can perform scanning using either FindFirstFile or NtQueryAttributeFile, even both. As we are creating deception schemes against InfoStealers, we focus on all those APIs related to information stealing for deception parameter extraction. Fig 2.3 indicates the APIs that are considered to comply with T1.

- **Goal Dependency (T2):** If selected APIs have several parameters, we exclude those parameters that do not lead to malicious goals. For example, if the API FindFirstFileExA is selected, it has 6 parameters: *lpFileName, fInfoLevelId, lpFindFileData, fSearchOp, lpSearchFilter and dwAdditionalFlags*. However, only the first parameter *lpFileName* holds file related information and indicates both the name and location of the file from where the malware is stealing information which can be replaced with a honey-file.

• **Consistency (T3):** We preserve the integrity of the system from attackers' point of view while presenting falsified data. We must consider the inter-dependency of the deception parameters because lying about one parameter without lying about its dependents might expose the deception. For example, if an InfoStealer checks the version of FileZilla from Windows registry before stealing the credential, we must put falsified data both in the registry and file-system to preserve the integrity.

There might be multiple deception parameters that satisfy criteria T2. For example, an InfoStealer may steal from various FTP clients. The *analysis agent* extracts several parameters that are related to these multiple FTP clients. Let V be the set of deception candidates obtained by the *analysis agent*. Each of the candidates in set V might be associated with different costs, as inter-dependent parameters must be selected with each of these parameters to satisfy T3. All of these potential system parameters are called deception candidates that fulfill T1, T2 and T3. The *analysis agent* calculates the cost of all possible candidates in set V to find out the optimal cost-effective candidates set.

**Optimal Deception Parameter Selection (ODP)** The deception parameter selection can be defined as a problem of selecting a subset from the set V, such that: (1) At least one parameter is selected from V to comply with T1, (2) If a parameter is chosen, all its dependencies are also selected (to comply with T3), (3) The selected

| Malicious Goal | Related APIs |
|---|---|
| Directory path retrieval | GetSystemDirectory, GetCurrentDirectory |
| File scanning | FindFirstFile, SearchPath, FindNextFile, NtQueryAttributesFile, GetFileAttributes, GetFileSize |
| Operation on file content | NtReadFile, NtWriteFile |
| Registry value query | RegQueryValueExW |
| Exfiltration | InternetCrackUrlA, ObtainUserAgentString, socket, connect, setsocketopt, send, closesocket |

Figure 2.3: Potential Candidate APIs for DPE

subset must be cost-optimal. To model this optimization problem, we define another set consisting of boolean variables $d_1, d_2, .., d_m$ for each of the parameters extracted by the *analysis agent*, where $d_i$ is set to 1 if the i-th parameter in V is selected for deception; otherwise, $d_i$ is 0. Then, We compute the total deception cost, C, that represents the cumulative cost of all the selected deception parameters.

$$C = \sum_{i \in [1,m]} (d_i \, ? \, \delta(v_i) : 0) \tag{2.3}$$

Here, $\delta(.) : V \rightarrow Z^*$ is a function that determines the cost of deception through each candidate parameter, $Z^*$ is the set of non-negative integers, $v_i$ is the i-th element in the set V and $(\psi \, ? \, v_1 : v_2)$ represents the if-then-else construct that evaluates to the value $v_1$ if $\psi$ is true and to the value $v_2$ if $\psi$ is false. To satisfy T3, we add another set of constraints to capture the dependencies of each selected candidate. The dependency constraints can be represented as follows:

$$\wedge_{v_i \in V} (d_i \rightarrow \wedge_{j \in \epsilon(v_i)} d_j) \tag{2.4}$$

We solve the constraints optimization problem using a solver [38]. The objective of this optimization is to minimize the cumulative deception cost denoted by $C$ in Equation 1. The outcome would be a set of deception parameters that satisfies our T1, T2, T3 criteria with optimal cost.

**Deception Playbook Creation and Population** At this stage, we obtain a set of system parameters that satisfy T1, T2, T3 criteria. These finalized system parameters are called deception candidates. The selected deception candidates are critical for malware to achieve their goals and potential entities that can be replaced with HoneyThings to mislead the adversary. Based on these selected deception candidates, we generate relevant HoneyThings. It is not feasible to model these HoneyThings

with automated file/registry/configuration generator as each of these files/registries/ configurations maintains different structures and hold unpredictable contents. Therefore, we create all the HoneyThings by installing the actual software and learn which values can be altered to produce more valid HoneyThings. Such an approach also ensures the generated HoneyThings are indistinguishable from the real system. The *analysis agent* stores these generated HoneyThings derived from the deception candidates into the *Deception Playbook* with a mapping to the key representative samples. Therefore, the deception playbook holds details of HoneyThings with reference to a malware sample indicating how to deceive the particular malware or similar malware and thus the adversary with the same intent; we design deception schemes with corresponding HoneyThings.

## 2.5 Dynamic Deception Scheme Creation (online) phase

During the *dynamic deception scheme creation phase*, DodgeTron detects zero-day malware and performs *light analysis* on the malware. By utilizing the clustering model we built in the offline, *analysis agent* classifies the new malware based on *light analysis* traces and maps the new malware to its key representative. The *planning agent* selects the deception scheme associated with the key representative and creates actuation task. The *actuating agent* executes the actuation tasks to deploy the deception scheme. Fig 2.4 shows the workflow of DodgeTron in the online phase.

### 2.5.1 Detection Agent in the online phase

The *detection agent* is the entry point of DodgeTron during the online phase and part of a production server. As DodgeTron is capable of designing deception schemes for zero-day malware variants, our *detection agent* must be competent in detecting zero-day malware. However, the main focus of this research is not to detect malware samples. Instead, we are solving the research problem of designing an automated deception system to deceive the attackers without human intervention. Therefore,

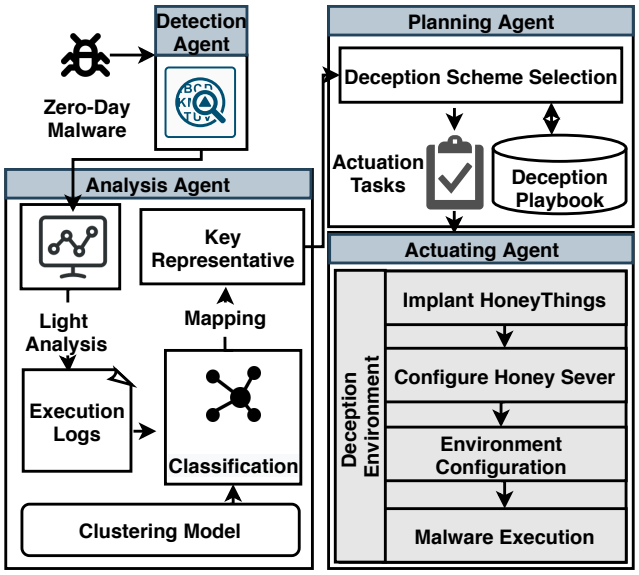



Figure 2.4: Workflow of DodgeTron during the Dynamic Deception Scheme Creation (online) phase

we can integrate zero-day malware detection mechanisms such as [39] that can have high accuracy (97.09%). Once the *detection agent* identifies a malware, it pushes the malicious executable to the *analysis agent*.

### 2.5.2 Analysis Agent in the online phase

Once the *analysis agent* receives a malware from the *detection agent*, it performs the *light analysis* on the sample. Based on the *light analysis*, the *analysis agent* classifies the new malware sample and identifies the key representative sample for it.

**Light analysis** The primary goal of *light analysis* is to perform a quick analysis so that the classifier can classify the malware according to its behavior. One state-of-the-arts is to implement malware classification based on execution traces retrieved from a sandbox like Cuckoo. Cuckoo reports detailed information about 318 Windows APIs along with parameter values. On the other hand, our *light analysis* provides two advantages over Cuckoo. Firstly, it reports all the invoked APIs during the execution. Secondly, we implemented annotation-plugins for 390 Windows APIs to retrieve parameter values which is 78 more than Cuckoo. We believe these APIs are significant to malicious behaviors. Such advantages help classifiers to achieve higher

accuracy.

**Classification** After the light analysis, the analysis agent uses the execution trace to classify the new malware sample with the trained clustering model acquired in the offline phase. In the next phase, a key representative sample is identified for the new malware sample and the actuating agent proceeds with the deception playbook created by the planning agent for the selected key representative sample.

### 2.5.3 Planning Agent in the online phase

Once the *analysis agent* notifies the *planning agent* about the closest key representative of the new malware sample, the *planning agent* selects HoneyThings from the *Deception Playbook* associated with the selected key representative and creates actuation tasks that are specific instructions about how to implant HoneyThings. Created actuation tasks are forwarded to actuating agent for further action.

### 2.5.4 Actuating Agent in the online phase

At first, accordingly to actuation tasks, the actuating agent downloads all the necessary honey-files and copies them into the referred locations. These honey-files contain honey-credentials, honey-financial information, etc. If the *Deception Playbook* includes registry values, the *actuating agent* modifies the necessary registry values and adjusts necessary configurations. These honey-registries contain honey-credentials and configuration values to make the system consistent from the attacker's point of view. Once download-copy-reconfiguration processes are completed, it executes the malware samples so that the malware collects and exfiltrates false information to the C&C server.

Other than creating actuation tasks, the *actuating agent* performs two major automated configurations: (1) configure honey server and (2) configure necessary settings so that the network-packets containing HoneyThings can flow towards the remote host (attackers). The honey server is instrumented with monitoring tools so that we

can capture an attacker's activities when the attacker logs into our honey server. The honey server is configured with honey-credentials placed in the *actuating agent*. For acquiring randomness, we randomize some of the values within the HoneyThings such as IP addresses, username, etc.

## 2.6 Implementation and Evaluations

We developed a prototype of DodgeTron using several technologies. (1) To automate the agents, DodgeTron contains more than 2,000 lines of python and bash scripts. (2) To develop 390 annotation plugins, we wrote around 18,000 lines of LUA code. (3) To create optimal deception parameter selection, we developed a heuristic-based approach consisting of 150 lines of Python code. We plan to put the codes in Github in the near future and a demonstration of DodgeTron can be found at [40].

For our evaluation measurements, we assess DodgeTron with different types of InfoStealers aging less than two years (2017-19). In the following section, we explain the outcomes of our experiments intended to verify the accuracy and effectiveness of DodgeTron. We evaluated the precision of DodgeTron in terms of creating the deception playbook in the offline phase. This metric is vital as the accuracy of designing deception for the new malware variants depend on it. We also evaluated the effectiveness of DodgeTron in terms of creating accurate automated deception schemes. We also performed a time evaluation of DodgeTron and showed time optimization we could gain due to hybrid analysis. As DodgeTron makes crucial decisions based on the clustering algorithm, we provide a rigorous performance outline of our classifying module.

### 2.6.1 Dataset

We collected malware samples and their AV labels from four sources: VirusShare, MalShare, Hybrid-Analysis and VirusTotal. As each AV has its own methodology of labeling malware, it is difficult to extract one unique label for each malware. Further-

more, many antivirus use generic labels such as *trojan* for InfoStealers. To verify the accuracy of our approach, we need labeled data. We construct an automated labeler to label samples based on keyword and a threshold value. Our strategy is to search a keyword (for example, LokiBot, Stealer, Khalesi, Emotet) within the VirusTotal provided AV signatures, if matches, we consider the label if the number of AV that labeled the sample with the given keyword is higher than a threshold value. For example, if the keyword is 'ramnit' and the threshold value is 10, then our automated labeler labels the sample as ramnit if at least 10 AV engines label this sample as 'ramnit'. Our dataset includes around 5000 malware samples consisting of InfoStealers from different families (e.g., Emotet, LokiBot, Generic.Password.Stealers, Trojan.Delf.Agent, Zeus, Khalesi, Dridex, Ramnit, etc). Out of 5000 malware samples, we randomly picked 4000 to train the clustering model and create Deception Playbook and used the remaining 1000 for evaluating DodgeTron. Therefore, both our training dataset and testing dataset are consistent in terms of containing a variety of InfoStealers types.

### 2.6.2 Experiment I: Evaluating the precision of DodgeTron in extracting deception parameters

We performed this experiment with 122 malware samples containing InfoStealers such as Azorult, Raccoon, Khalesi, Vidar, LokiBot and Ramnit. The ground truth of these samples for the verification are the technical analysis reports provided by security analysts such as [41, 42]. For simplicity, we explain the experiment in this section using a single sample, $m_1$ (MD5: eccad903b4c27d149e159338f58481a9).However, our experiment is repeated for all 122 malware samples. On average the precision of our deception parameter extraction is **91.45%**. Fig 2.5 summarizes the results of experiment I.

**Ground truth for verifying precision:** We uploaded the sample to ANY.RUN. It identified it as LokiBot. According to the SANS [42] report, $m_1$ steals information

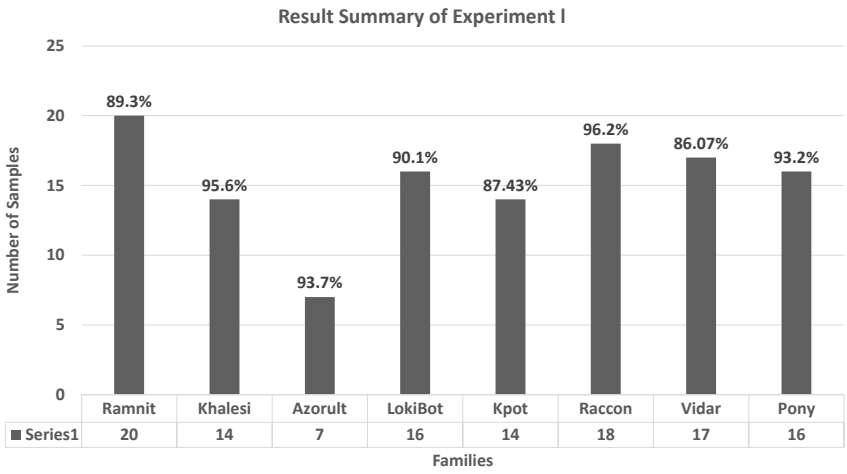



Figure 2.5: Precision of DodgeTron across different malware families in term of deception parameter extraction

from around 120 various applications that store sensitive data at the Windows registry and the local file system and later exfiltrates stolen data to the C&C server. Therefore, to verify the precision of our deception parameter extraction and deception playbook creation, we compared our result with the SANS report which is the ground-truth here.

**Verifying the precision of deception parameter extraction:** We observed the filtered traces after DPE extracted all possible deception candidates that satisfy T1, T2, T3 criteria. We found the malware tried to collect sensitive information such as credentials of the browsers, FTP clients and other applications from different registry entries and files. For example, to steal *stored credentials* from *Google Chrome*, $m_1$ retrieves the version and installation directory of Google Chrome by performing queries using RegQueryValueExW and SHGetValue APIs on *HKEY_LOCAL_MACHINE* registry. Then it calls NtQueryAttributeFile with *"C:\Users\USERNAME\AppData\Local\Google\Chrome\User Data\Default\Login Data"* as "ObjectAttributes" to check the existence of the file. "Login Data" is an *SQLite DB* file where *Google Chrome* stores credentials. Later, $m_1$ opens it and performs SQL queries to read site URL, username and password and decrypts the passwords using CryptUnprotectData API and stores them into allocated heap. Later, it exfiltrates the data from the heap.

Therefore, the "Login Data" satisfies T2 and is considered as one of the deception candidates. Another victim of $m_1$ is *BlazeFTP*; $m_1$ identifies the path that contains the credential file "site.dat" from the Windows registry under the key: "Software\FlashPeak\BlazeFtp\Settings." On success, it recursively enumerates the values such as hostname, username and password from the "site.dat" and exfiltrates the data to the C&C server. Here, selecting all these aforementioned APIs comply with T1, the "site.dat" file satisfies T2 and the "Settings" satisfies T3. These registries and files (e.g., "site.dat" and "Login Data") are potential deception candidates as they can be substituted by HoneyThings (honey-registry and honey-files). DodgeTron was able to extract 109 candidates out of 120 mentioned in the SANS report from $m_1$, which is almost **91%** consistent. In addition, DodgeTron identified 195 other dependent candidates that needed to be replaced to comply with T3 which keeps the system consistent from the attackers point of view.

### 2.6.3 Experiment II: Evaluating the accuracy of DodgeTron in the online phase

We performed this experiment with 953 malware samples randomly picked from our dataset. We observed and confirmed 869 of these 953 samples were successfully deceived by DodgeTron with an accuracy of **91.18%**. However, in many cases, we could not confirm the malware got deceived or not due to unknown encryption algorithm. If we could verify them, our accuracy could have gone higher to **95.9%**. Fig 2.6 summarizes the results of experiment II. For simplicity, we explain the experiment using two infoStealers, where $m_2$ (MD5: *68119dd7fb9ecb099de50227162bd82f*) is a new malware that we didn't use while training our classifier. On the arrival of $m_2$, the classifier performs fine-grained classification based on *light analysis* to classify it. Our classifier identifies a key representative $m_3$ (MD5: *ea722bea2a44cd06d797107d5ff9da92* that can represent $m_2$.

**Ground truth about $m_3$:** We have already performed in-depth analysis and deception parameter extraction on m3 during the offline phase and identified $m_3$ is

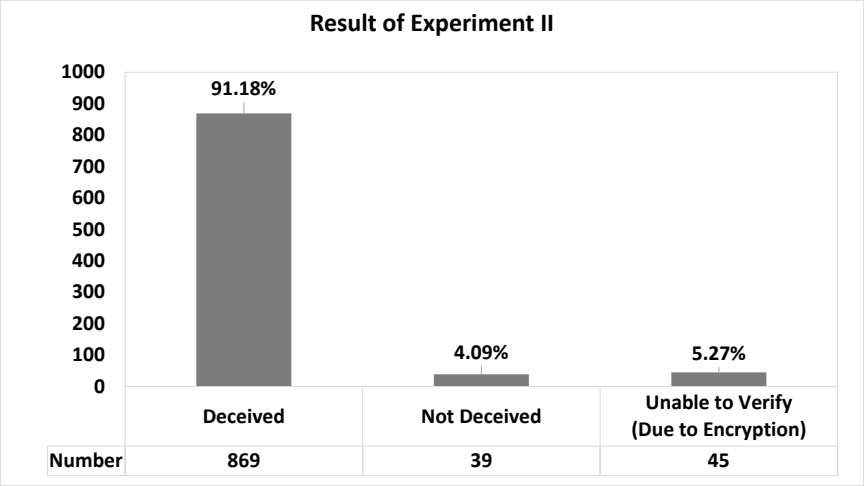



Figure 2.6: Overall accuracy of DodgeTron across different malware families in term of deceiving the malware

capable of retrieving five distinct digital wallets: Ethereum, mSIGNA, Electrum, Bitcoin and Armory. It queries subkey values using *RegQueryValueExW* to locate the path of digital wallets client. In the next step, $m_3$ targets one by one each client directory and iterates to find specific files such as *.dat using *NtQueryAttributeFile* API. For example, in the case of Ethereum, $m_3$ tried to capture information from C:\Users\sajid\AppData\Roaming\Electrum\wallets \wallet.dat. Therefore, we created a honey wallet.dat file containing a fake private key in the retrieved location and a honey-registry containing honey values at HKEY\appdata\Ethereum\wallets.

**Verification set up:** To verify that our deception worked, we set up a proxy C&C server for redirecting the traffic towards our proxy server using ApateDNS. In this case, DodgeTron implanted a honey-file named *wallet.dat* at C:\Users\sajid\AppData \Roaming\Electrum\wallets and a honey-registry values in the HKEY\appdata\Ethereum\wall ets registry. At this stage, we let the malware run and steal information. Within a few minutes, our proxy C&C server started to receive packets from the infected machine. We found that a zip file had been uploaded to our proxy server. We extracted this zip file and found it contained a folder named "Ethereum". We observed and found the exfiltrated file contained the contents of honey-file that DodgeTron implanted.

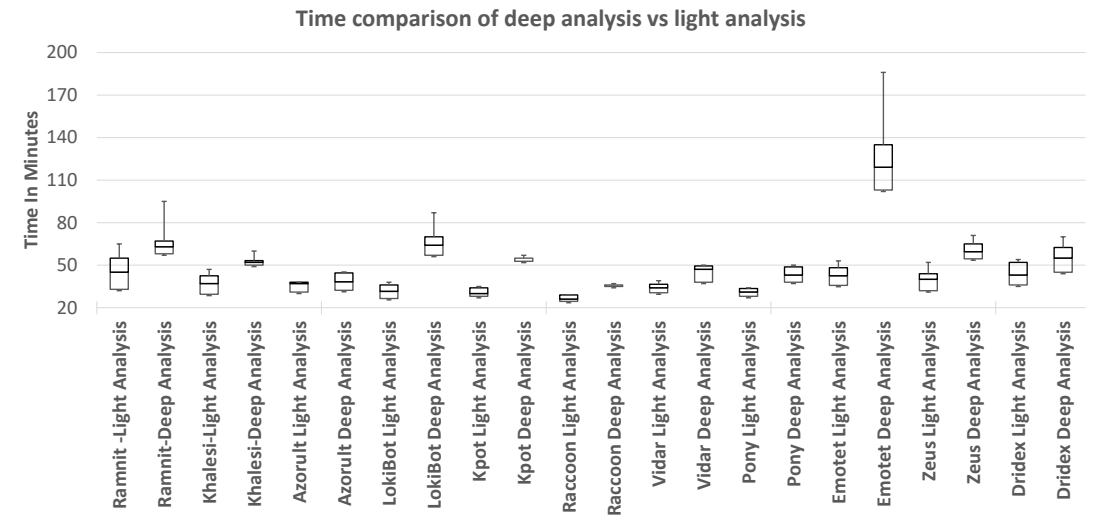
**The explanation for the failed cases:** If malware does not send any traffic to the malicious C&C server, we consider such cases as "Not Deceived" by DodgeTron. However, in some cases, we observed malware samples forwarded traffics to the C&C server. However, as the traffic was encrypted with an unknown algorithm, we could not verify whether the traffic contained the content of honey-files or not.

### 2.6.4 Performance analysis of DodgeTron in terms of execution time

In the offline phase, DodgeTron performs costly operations such as *deep analysis* to train the clustering model. On average, for each sample, the *deep analysis* was performed in 72 minutes. Therefore, to minimize the analysis time, we perform light analysis during the online phase to classify the new malware, which takes 27.7 minutes on average. The clustering in the offline phase is completed within 19.7 minutes, which is very small compared to the total time required by the *deep analysis*. Therefore, to minimize the analysis time, we perform light analysis during the online phase to classify the new malware, which takes 27.7 minutes on average. The clustering in the offline phase is completed within 19.7 minutes, which is very "small number" compared to the total time required by the *deep analysis*. Therefore, we ignore it in performance analysis. In addition, the time required to classify new malware in the online phase is also ignored as it takes only a few seconds to classify the new malware. Hence, by utilizing the hybrid analysis, DodgeTron speeds up the analysis process and saves time. Fig 2.7 shows the speedup DodgeTron gained across different malware families due to hybrid analysis.

### 2.6.5 Performance analysis of our classifier

We extract API names and their parameters from the execution traces and encode them as numerical vectors. The embedder produces 32-dimensional vectors representing each trace. These vectors are then grouped into multiple clusters according to their cosine similarities using agglomerative hierarchical clustering technique.




| | Ramnit | Khalesi | Azorult | LokiBot | Kpot | Raccoon | Vidar | Pony | Emotet | Zeus | Dridex |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Speed UP | 1.4 | 1.41 | 1.1 | 2.03 | 1.83 | 1.38 | 1.38 | 1.38 | 2.8 | 1.49 | 1.28 |

Figure 2.7: Time Comparison of *deep analysis* vs *light analysis* across different malware families and the speedup factor

In order to find the optimal number of clusters, we calculate the homogeneity score at each level of hierarchical clustering. In a perfectly homogeneous clustering case, every cluster contains only data points belonging to a single class, which indicates zero entropy in the clusters. Hence, the objective is to maximize the homogeneity score by increasing the number of clusters. Given $N$ data points, a set of clusters $K = \{k_i | i = 1, ..., m\}$, and a set of classes $C = \{c_j | j = 1, ..., n\}$, we calculate the homogeneity score $hs$ as follows:

$$hs = \begin{cases} 1, & \text{if } H(C|K) \text{ or } H(C) = 0 \\ 1 - \frac{H(C|K)}{H(C)}, & \text{otherwise.} \end{cases} \tag{2.5}$$

In fact, as the knowledge of $H(C)$ reduces, the uncertainty of $H(K)$, the conditional entropy $H(H(C)|H(K))$, becomes smaller. Fig 2.8 shows the trade off between homogeneity score and the number of clusters generated by agglomerative hierarchical clustering technique. It demonstrates that increasing the number of clusters results in higher homogeneity score until the cluster count reaches 275. From that point on,

the homogeneity score does not change. Therefore, the total number of 275 clusters is maintained as the optimal cluster count. The 91% homogeneity score means that, in each cluster, slightly 91% of the samples belong to a common class. The approach showed good results for the following reasons:

1. Malware from the same family usually follows analogous execution patterns. Thus, simple clustering failed to precisely cluster different families. However, this approach mainly focuses on representing mutually exclusive groups where each one demonstrates a unique behavior regardless of the associated family.

2. In simple clustering, key representatives are selected randomly. Here prototypes are selected more systematically in which it takes samples from various distributions where each one displays a unique pattern.

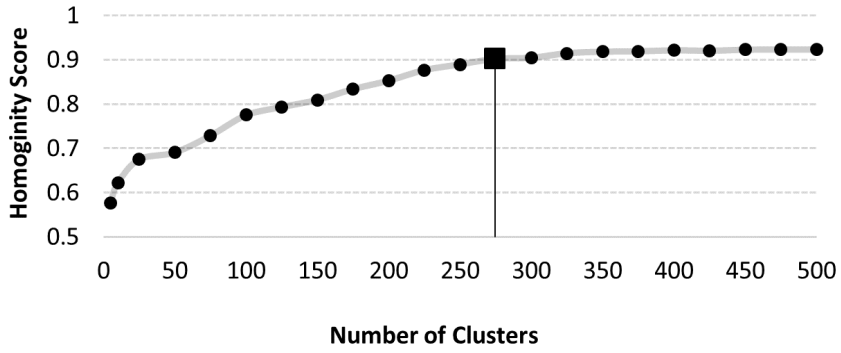



Figure 2.8: The trade-off between homogeneity score and number of clusters in hierarchy

## 2.7 Related Work

To the best of our knowledge, only a few research works have focused in obtaining useful information from the InfoStealers through automated analysis. In [43] and [44] the authors proposed automated systems capable of analyzing banking-trojans based on their web-injection behaviors. However, these approaches only considered banking-trojans, which are a subset of InfoStealers. In addition, they assumed that the attack

mechanism is web-injection based, which is not true for all types of InfoStealers.

Honeypots [11] and honeynets [45] have been widely adopted to complement traditional detection and prevention mechanisms. Such methods offer attractive artificial information (i.e., baits) to attackers to divert them from the real targets. However, lack of randomness is a fundamental hurdle for these approaches. A few studies such as [32] proposed alternative solutions to formulate honeypots that are indistinguishable from the real system. Unfortunately, these approaches work only theoretically and are not directly applicable to real world scenarios as an attacker can adapt techniques such as [46] to detect both low-interaction and high-interaction honeypots. In addition, these techniques assume the malware will take the baits as they are lucrative. However, malware may not take the planted baits.

Advanced honeypot and honeynet techniques such as shadow honeypot [7] hinders information on the production system from the attackers. In this approach, both the real production system and an instrumented shadow version of the actual system are deployed separately. Anomaly detection sensor forwards suspicious network flow to the shadow copy, while legitimate flows are directed to the real system. Thus the precision of such a mechanism solely relies on the accuracy of the anomaly detection sensor to be able to detect the malicious payload, which is not always accurate [47]. In [22], the authors proposed to instrument production systems with fake services and mock vulnerabilities to attract attackers. However, it is possible the attackers might launch attacks with specific services or vulnerabilities that are not fabricated. Besides, the approach has a dependency on the sysadmin, so it requires manual assistance.

An orthogonal line of works utilized decoy files or honey accounts to detect ransomware [23, 24], general malware [25, 26], or DDoS attacks [27]. In [28] and [29], researchers employed honeypots and honeytokens to detect and prevent web-based attacks. Such strategies are out of the scope of our work as they mainly focus on detection where we are interested in deception. Moreover, these techniques are designed

to detect only a particular type of malware where our approach is intended for the general type of InfoStealers.

DodgeTron is built upon and can benefit from an extensive body of research on dynamic analysis of malware [48, 49], such as forced execution [50], multipath exploration [51] and symbolic execution [35]. While our work needs to analyze malware, we have a different goal: to automatically discover system parameters that can be leveraged to deceive, rather than detect, malware.

## 2.8 Discussion & Conclusion

We acknowledge a few technical challenges concerning our approach. As we intend to maintain the honey-files close to the real system, we generate all the honey-files by installing the actual software, which requires a lot of manual effort. Since we utilize symbolic execution, we inherit the limitations (e.g., state-space explosion) of symbolic execution. To minimize the state space explosion, we limit path exploration. Limitation of path exploration is not effective in discovering interesting malware execution code. However, we plan to generate honey-files automatically by implementing machine learning algorithms. Currently, we do not have an implementation to counter malware evasion. We plan to incorporate strategies in future to address this issue.

We present the first autonomous cyber deception framework that is capable of creating a deception scheme through rigorous dynamic malware analysis, automated reasoning and decision making. We execute malware samples through a selective symbolic execution engine to classify the malware samples based on their interactions with the system. Based on classifier results, we perform deception oriented analysis to extract cost-effective deception candidates to design consistent and resilient deception schemes. We have experimentally verified the effectiveness of DodgeTron against recent malware.

# CHAPTER 3: symbSODA: Configurable and Verifiable Orchestration Automation for Active Malware Deception

## 3.1 Introduction

Active cyber deception (ACD), which complements and overcomes the shortcomings of traditional detect-then-prevent strategies, has emerged and developed into an effective defense technique. ACD creates doubts and ambiguity in the adversary's mind by intentionally, actively and deliberately concealing or falsifying actual system configurations (such as the keyboard layout, registry values, and IP address). In other words, ACD corrupts the adversary's perception of the victim system and decision-making processes. Best practice cyber deception systems have been primarily high-interaction decoy systems that contain fake files, user accounts, credentials, and other components. However, cyber deception has a wider range of benefits than only catching attackers in a controlled environment [6, 8, 23, 24]. An effective cyber deception can achieve 4D goals: (1) *deflect* adversaries to false targets, (2) *distort* adversaries' perception about the environment, (3) *deplete* adversaries' resources and (4) *discover* adversaries' motives, tactics and techniques [21].

Although cyber deception has been applied effectively in numerous settings, existing deception approaches lack agility, resilience, and automation. Furthermore, these approaches fall behind with static deployment and configurations that are easily distinguishable from the real systems [1], which skilled attackers can quickly discover and circumvent [2–5]. Existing deception techniques are largely intended to thwart attackers at a certain stage of the kill-chain phase. For instance, some well-known deception techniques use honeypots [1,6,7], honeyfiles [8–10], honeypatches [11], honeybugs [12] and decoys [13] to mislead attackers during the collection and credential

access phase. On the other hand, others use malicious traffic rerouting, such as [17], to deter attackers at the command and control phase. Only a few approaches take into account every kill chain phase [13,19,20]; however, they cannot provide programable and configurable deception plans to their users, which enables them to create and design their deceptive playbook based on their own requirements. Network randomization [18] and moving target defense [14–16] are other well-known deception strategies.

Existing sandbox-based malware analysis agents suffer from two major drawbacks: 1) malware evasion mechanisms against the well-known systems and 2) inability to execute malware in multipath exploration settings. The anti-malware systems such as detection, investigation, response, prevention, and deception-oriented systems depend on these analyses to design countermeasures against the malware, while the malware implements evasion techniques, trigger-based conditions and logic bombs to corrupt the analysis results. If the analysis data is corrupted, the countermeasure is no longer effective. Besides, the malware can have environment sensitivity checks to target only specific systems. Therefore, it is essential to discover these conditions to uncover the interesting malicious behaviors and learn from them to design an accurate anti-malware system. Multipath exploration techniques such as symbolic execution can explore both sides of the conditional branching within the malware. Therefore, they can generate comprehensive execution logs and provide more insights into how multiple paths can lead toward malicious goals. In summary, accurate and more malicious sub-graph (MSG) extraction is a key for designing a precise deception-oriented system. Therefore, a multipath exploration-based analysis system can observe more MSGs than a single path exploration-based analysis system like Cuckoo Sandbox. Hence, such an analysis agent provides us with a more exciting malware execution path to play the deception game.

This work addresses these limitations by introducing an autonomous cyber decep-

tion system called symbSODA that provides deception as a service and orchestrates a deceptive environment at run-time. First, symbSODA execute malware symbolically and extracts malicious sub-graphs from the execution traces. A malicious subgraph (MSG) represents a sequence of WinAPI calls that work together to perform a malicious task. During extraction, we traverse different execution paths within the malware in order to extract comprehensive number of MSGs. Next, symbSODA maps these extracted MSGs to the MITRE ATT&CK framework to determine the malware's behaviors at the kill chain tactical level. Later, this knowledge base is utilized to create the Deception Playbook, a set of deception course-of-actions to be performed to deceive specified malicious behavior with a given deception goal and strategy. Users can weaponize their systems using these pre-built deception playbook profiles or can create their own. Based on deception playbook profile selection/creation, symbSODA provides deception as a service where orchestration is performed automatically.

This chapter is a major extension of SODA, presented in a previous conference paper [52]. We extend SODA in the following ways: (1) **Adding symbolic execution capabilities**. In our conference paper, we used concrete execution analysis based on Cuckoo sandboxing. The analysis was limited due to a lack of multi-path exploration of malware code. Thus, in this version we developed a significantly improved API Call Tracer (discussed in Section 3.3.1.1), which employs symbolic execution capability to provide us with better system call monitoring via kernel-level hooking. In addition, symbolic execution allows the API Call Tracer to explore multiple paths within the malware, which helps extract more MSGs that were not explored with SODA. (2) **Adding verifiable orchestration**. We developed a Deception Planning Verifier using the Assume-Guarantee technique to ensure consistent and conflict-free deception actions (Section 3.3.3.5). In our conference paper, SODA deployed deception actions without considering potential conflicts among them. Such conflicts can cause incon-

sistency among the deceptive actions leading to malfunctioning deception plans that can be leveraged by attackers. Therefore, it is important to keep the system consistent from the API and the attackers' point of view [53,54]. We developed a deception planing verifier that uses the assume-guarantee technique to identify conflicts among the deception actions and remove them during user configuration. This process can also guide the deception developers to implement deceptive programs that are consistent and valid. (3) **Enriching honey factory**. We enhanced the implementation of the SODA honey factory with a more comprehensive set of capabilities; previously, we had only a few honey files and registry values inside the honey factory. (4) **Improving evaluation and case studies**. We improved the evaluation by including the comparison of symbSODA with other related tools in terms of detecting different MITRE ATT&CK techniques within malware execution. We also added new case studies (Section 3.5.1) on three different types of malware to demonstrate how symbSODA can advance SODA by overcoming the triggering conditions within the malware and extracting new MSGs.

Specifically, we make the following contributions:

- We propose a dynamic security orchestration, automation, and deception system, symbSODA, which enables users to orchestrate deception ploys with appropriate strategies and goals dynamically.

- We propose a symbolic execution driven analysis agent that satisfies malware condition checkings to extract comprehensive MSGs. The analysis agent in our earlier work, SODA, was developed on top of the Cuckoo Sandbox, which could only perform concrete execution and had limitations against evasive/environment-aware malware.

- We propose a systematic way of removing conflicting deception actions by performing assume-guarantee analysis on selected deception actions to detect po-

tential conflicts and warn the users and deception developers to resolve them.

- We propose automated MSG extraction and MSG-to-MITRE mapping, allowing symbSODA to understand malware behaviors at runtime to activate relevant deception ploys.

- We propose an embedded deception technique based on API hooking, allowing symbSODA to execute deception ploys in real time.

- We evaluated symbSODA with recent malware to determine the accuracy and the scalability of our approach. We observed an accuracy of 95% in deceiving malware with negligible overhead and deployment time. Furthermore, our approach successfully extracted MSGs with a 97% recall value and MSG-to-MITRE mapping achieved a top-1 accuracy of 88.75%.

The remainder of the chapter is organized as follows: In Section 4.2, we discuss the threat model and assumptions. Section 4.3 explains how MSG extraction and MSG-to-MITRE mapping are used to develop deception playbooks offline. Section 4.4 presents how symbSODA provides deception as a service in real-time. The outcomes of the evaluation are presented in Section 4.5. Related work is discussed in Section 4.6. Finally, limitations, future work and conclusions are presented in Section 4.7.

## 3.2 Threat Model and Assumptions

We used symbSODA to analyze thousands of malware samples and extract relevant MSGs (malicious sub-graphs) that are readily usable by users to define deception ploys. In addition, users can extend the collected library of MSGs in sympSODA assuming there exist sufficient and representative malware repositories for each major malware type (e.g., RATs, ransomware, and InfoStealers) to be used to extract MSGs. This will enable users to effectively define custom-made deception ploys for new malware types. Users can obtain a repository of malware samples from public malware

sharing sites (such as VirusTotal [55] and MalShare [56]), which provide both malware samples and their labels that can be used to infer malware type.

Users can use symbSODA to define fixed ploys for each MSG applied for all malware. In this scenario, users will enable all defined ploys simultaneously and only the ploys that belong to an MSG related to analyzed malware will be activated at run-time. On the other hand, users of symbSODA can define various profiles based on the anticipated types of malware (e.g., ransomware). In this case, the same MSG can be associated with different ploys in each profile according to the malware type and user selection. We assume that the malware type can be obtained by the malware detector or analyzer [10]. Thus, each profile can have its specific ploys designed based on the deception objective of this malware type. As discussed in Section 4.4, we can make these kinds of custom-built deception profiles free of errors.

## 3.3 Deception Playbook Creation

A single malware can contain multiple malicious behaviors. These malicious behaviors can be deceived in multiple ways depending on different deception goals and strategies. Depending on the deception goals and strategies, each malicious action can be deceived in various ways. For example, suppose a malware behavior is to steal a credential file. One way to deceive the malware is to supply honey credentials in the form of a honeyfile. Another option is to pretend the credential file doesn't exist on the system. Each action is referred to as a *Deception Ploy*. Therefore, *Deception Ploys* or simply *ploys* are the actions or measures we can perform to deceive malicious behaviors with given strategies and deception goals. If a malware ($m_1$) has the following behaviors: $B_1$, $B_2$ and $B_3$ and $T_1$, $T_2$ and $T_3$ are the ploys used to deceive these behaviors respectively, then *Deception Playbook* of $m_1$, $DP(m_1) = \{T_1, T_2, T_3\}$. Therefore, we can say that the *Deception Playbook* is a collection of deception ploys intended to deceive a certain set of malicious behaviors.

The *Deception Playbook* Creation phase aims to create Deception Playbooks for

given malware and store them so that if the same/similar malware compromises our system in the future, we can deceive it using the stored Deception Playbooks. The Deception Playbook Creation phase can be divided into three parts: *Malicious Sub-graphs (MSGs) Extraction*, *MSG-to-MITRE Mapping* and *Deception Playbook Synthesis*. Firstly, we extract MSGs from real-world malware. Secondly, we map these retrieved MSGs to MITRE techniques as well as to our defined malicious behaviors. As MSGs represent the malware's low-level implementation details, they might be complicated or ambiguous for those who are not experts or familiar with WinAPIs. Behaviors are high-level descriptions of malware actions or capabilities. We perform MSG-to-MITRE mapping to assist the users in understanding what and how the malware is trying to execute malicious action and how the deception mechanism will combat that action. Finally, we develop deception ploys for various malware behaviors in the Deception Playbook Synthesis. Then, using the deception techniques, we create and store *Deception Playbook*. Finally, we implement the Deception Factory, which includes developing hooks and REST APIs and creating HoneyFactories (HF) to perform the actions outlined in Deception Playbooks. The whole procedure of this phase is depicted in Figure 3.1 and explained in the following sections.

### 3.3.1 Malicious Sub-graphs (MSG) Extraction

The malware must call a sequence of WinAPIs to achieve a particular malicious objective/goal/behavior. If each WinAPI in this sequence is represented as a node and the data flow between two WinAPIs is represented as an edge, then this WinAPI sequence can be represented as a graph. These graphs are defined as malicious sub-graphs (MSGs). We note that an MSG is related to, but not strictly a sub-graph of, the malware's control flow graph. Specifically, nodes in an MSG represent Windows API calls, not basic blocks as in a control flow graph; and edges in an MSG represent data flows among API calls, not jumps (or control transfers) as in a control flow graph. A control flow graph can contain API call information in its nodes, but it

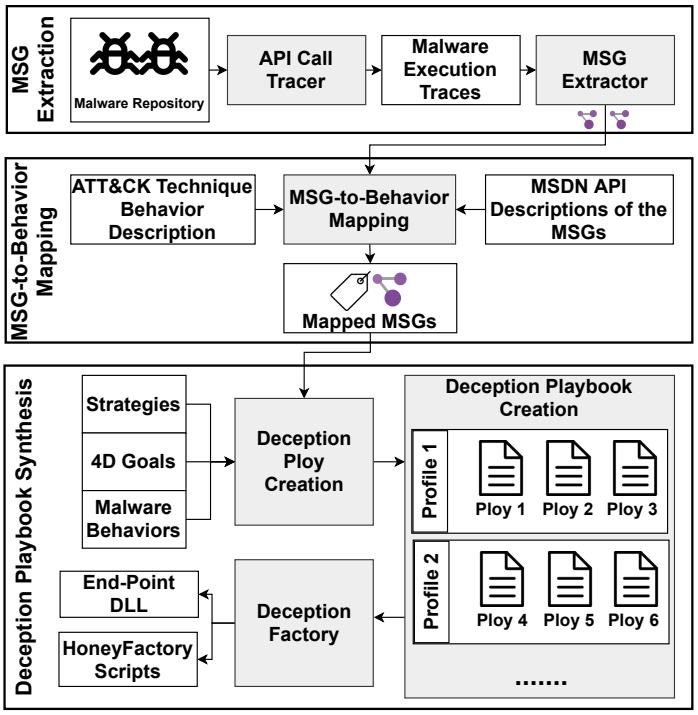



Figure 3.1: Deception Playbook Creation

does not directly represent data flows.

Identifying these MSGs from malware traces is necessary to understand malware execution flow, and these execution flows will lead us to design an accurate deception plan. We need an environment where we can execute hundreds of malware and collect their execution traces to extract MSGs. We build the API Call Tracer (also referred to as the Analysis Agent) by utilizing the sandboxing idea and it's built on top of a selective symbolic execution engine [9, 10, 35]. We also automated the process of uploading malware, collecting traces, and extracting MSGs.

#### 3.3.1.1 Symbolic execution based API Call Tracer

We use the sandboxing technique to collect malware execution traces. The API Call Tracer consists of three modules: a) Template Code Generator, b) Hooking Engine and c) Symbolic Execution Engine. The Template Code Generator automates the generation of the corresponding C++ source code for the Hooking Engine, which is required for basic API monitoring. The Hooking Engine is used to intercept the

selected APIs and then, based on different reasoning, switch to the Symbolic Execution Engine. The Hooking Engine is implemented using EaskHook [57] and is used to intercept the desired API calls and then switch to symbolic execution mode based on necessity. The Symbolic Execution Engine is built on top of S2E, a selective symbolic execution engine [35]. Inside API call traces, our combined EasyHook-S2E plugins can intercept and change the original API calls with both concrete and symbolic parameter values. This mechanism facilitates us in collecting execution traces on both sides of the conditional branching. For example, if the malware is searching for a specific file, conditional branching would go either to the success branch (if the file exists) or to the failed branch (if the file does not exist), but not both. As the API call tracer employs symbolic execution, it can traverse both sides of the branching by assigning symbolic values to the conditional variables during the analysis. Thus, it explores multi-paths within the malware; as a result, our execution traces are rich and enhance the quality of MSG extraction.

As MSGs are the first building block of our system, our MSG extraction needs to be comprehensive, i.e., ideally we want our API Call Tracer to log all possible WinAPIs with all parameter values. Unfortunately, doing that will impact the performance of both the API Call Tracer and the MSG Extractor. Therefore, we choose the middle ground where the API Call Tracer will have the maximum WinAPI coverage with a reasonable performance overhead. Hence, we select only those WinAPIs that are significant among the malware. To find the significant WinAPIs, we leveraged gExtractor [9], a dynamic analysis tool that can report all WinAPIs that are invoked by a given piece of malware. We ran 1,000 representative malware samples using gExtractor and identified 516 unique WinAPIs. Therefore, we decide to cover these 516 WinAPIs with all parameter values in the API Call Tracer.

However, we still need to write the template codes inside the Hooking Engine for each of these 516 WinAPIs that we want to monitor, which is a non-trivial task.

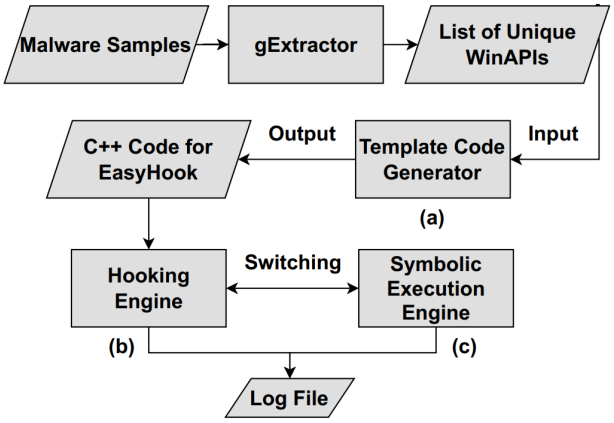



Figure 3.2: API Call Tracer: (a) Our implemented approach automates the template code generation process, (b) The Hooking Engine is responsible for intercepting API calls, and (c) The Symbolic Execution Engine is responsible of executing the malware symbolically. (Rectangular shapes indicate agents (scripts) capable of processing, while parallelograms indicate I/O data for respective agents.

Therefore, we automated the process of C++ template code generation for the hooking engine. Firstly, we provide a list of WinAPIs to be monitored to our tool. For each WinAPI, our tool retrieves its definition from the corresponding MSDN website. Secondly, using the scripts and retrieved definitions, our tool generates the bare-minimum necessary C++ code for each API that is required for monitoring individual API. We implemented Python scripts (we refer to it as "Template Code Generator") to automate this code generation process. Finally, some of these API codes are changed based on the condition of symbolic exploration.

Each module within the API Call Tracer and its workflow is depicted in Figure 3.2. Let's explain the workflow of the API Call Tracer with an example. At first, the Hooking Engine starts the malware in a suspended state, injects a DLL and then resumes the suspended process. Now, let's assume the malware calls *GetLocalTime* API, and we have a hook function for this API inside the Hooking Engine, which intercepts the original API call. Again, it's unnecessary to execute every single API called by the malware symbolically; we selectively enable the symbolic execution based on our domain knowledge. Now, let's say we want to enable the symbolic execution for the *GetLocalTime* API. We can call *S2EMakeSymbolic* (a function offered by S2E for

```
1  RegOpenKey {'key_handle': '0x000000bc', 'regkey': 'HKEY_LOCAL_MACHINE\\SOFTWARE\\Mozilla\\Mozilla Firefox'}
2  CreateToolhelp32Snapshot('out_handle': 0x000000f8')
3  RegQueryValue {'key_handle': '0x000000bc', 'value': '41.0.2 (x86 en-US)', 'regkey': 'HKEY_LOCAL_MACHINE\\SOFTWARE\\Wow6432Node\\Mozilla\\Mozilla
   Firefox\\CurrentVersion'}
4  RegCloseKey {'key_handle': '0x000000bc'}
5  ........
6  Process32First {'process_name': '[System Process]', 'snapshot_handle': 0x000000f8'}
7  Process32Next {'process_name': 'System', 'snapshot_handle': '0x000000f8'}
8  Process32Next...
9  ........
10 FindFirstFile{'filepath': 'C:\\Users\\Administrator\\AppData\\Local\\Google\\Chrome\\User Data\\Default\\Login Data'}
11 CreateFile{'out_file_handle': '0x000000de' 'filepath: 'C:\\Users\\Administrator\\AppData\\Local\\Google\\Chrome\\User Data\\Default\\Login Data'}
12 GetFileType {'file_handle': '0x000000de'}
13 ........
14 ReadFile {'file_handle': '0x000000de' buffer,: 'FileContent', 'length': 260, 'offset': 0}
15 SetFilePointer {'file_handle': '0x000000de' 'move_method': 1, 'offset': 260}
16 ReadFile {'file_handle': '0x000000de' 'buffer': 'FileContent', 'length': 4096, 'offset': 0}
17 CloseHandle {'handle': '0x000000de'}
18 ........
```

Figure 3.3: Execution trace of a malware sample generated in Cuckoo Sandbox (N.B. The size of the parameter“buffer" is large hence denoted as “FileContent" instead of the original content)

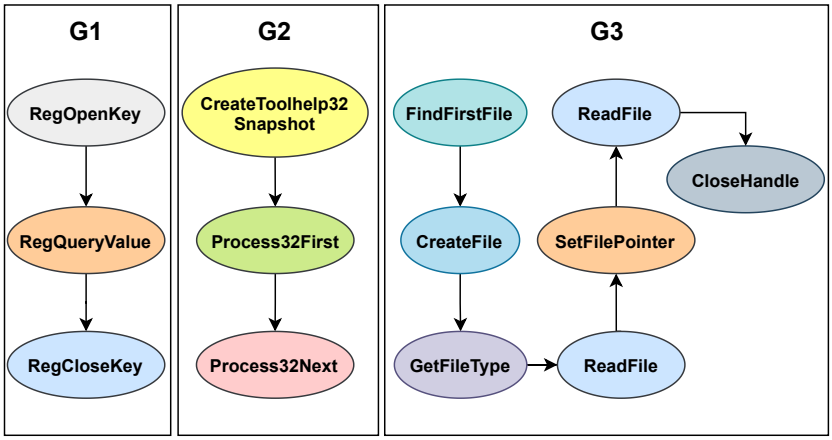



Figure 3.4: MSGs generated from malware traces

marking a particular variable/set of variables symbolic) with the variable we pass to *GetLocalTime*, which is *lpSystemTime*. By doing so, the selective symbolic execution engine will mark *lpSystemTime* symbolic and will execute the *GetLocalTime* API call. Suppose the malware performs any checking on this variable to make a decision. In that case, we can find out the concrete value of this variable that satisfies the conditional check using the SMT solver within the Symbolic Execution Engine. The evaluation section provides more insights about the API Call Tracer.

#### 3.3.1.2 MSG Extractor

Once we have run malware samples via the API Call Tracer and collected traces, we'll utilize them to extract MSG. MSG extractor extracts MSGs from traces where

node denotes WinAPI and edge represents the data dependency between APIs' parameters. Figure 3.3 is an example trace generated by the API Call Tracer. Please note that the traces depicted in Figure 3.3 are not sequential but instead picked from three separate locations inside a single log file to demonstrate how the graph generation and extraction work.

In Figure 3.3, lines 1,3,4 illustrate that the malware opens a registry, executes a query and then closes the registry after the query is complete. Lines 2,6-8 show that the malware enumerates currently running processes. The malware scans, harvests and steals credentials from the Google Chrome browser on lines 10-17. In Figure 3.4, G1, G2, and G3 are three MSGs created from lines 1,3,4, lines 2,6-8, and lines 10-17 in Figure 3.3, respectively.

The pseudocode of our MSG extraction is listed in Algorithm 1. It takes a log file as input and generates MSGs. It uses API calls from the execution trace to construct graph nodes and the arguments/parameters to link API calls based on data dependencies. It uses variable G to represent the MSG and four lists of (key, value) pairs to find the data dependencies among API calls. In each (key, value) pair, the key represents a concrete parameter (such as 0x000000bc), and the value represents an API node in the MSG. The four lists (i.e., Active_ids, Active_files, Active_regs and Active_bufs) correspond to four common types of parameters: handler, file path, registry path, and buffer.

Let's see how our algorithm extracts MSG G1 from the log file in Figure 3.3. When the function MSG_Extractor processes line 1 of Figure 3.3, it recognizes the parameter value 0x000000bc as an identifier (in line 21) and therefore inserts a (key, value) pair (0x000000bc, RegOpenKey) into Active_ids (in line 32). Next, when it processes line 2 of Figure 3.3, because the parameter value 0x000000f8 does not belong to the keyset of Active_ids (line 29), it inserts a new pair (0x000000f8, CreateToolhelp32Snapshot) into Active_ids (line 32). When it processes line 3 of Figure 3.3,

**Algorithm 1** MSG Extraction Algorithm

```
1: function ADD_EDGE(API, item, action, targetList)
2:     Add Edge into G from node2 to node1
3:         where: node1 = API
4:             node2 = targetList[item]          ▷ Add the closure API to the MSG before removing
5:     if action=="remove" then
6:         Remove item from targetList
7:     else
8:         Update targetList
9:     end if
10: end function
11: function MSG_EXTRACTOR (logFile)
12:     Variable Initialization
13:     G = {}                                                          ▷ Empty Directed Graph
14:     Active_ids = {}      ▷ Key holds identifier and value holds latest node to use this identifier
15:     Active_files = {}          ▷ Key holds file name and value holds latest node to use this file
16:     Active_regs = {}  ▷ Key holds registry key and value holds latest node to use this registry
17:     Active_bufs = {}  ▷ Key holds starting address of buffer and value holds latest node to use
    this buffer
18:     closure = [CloseHandle, RegCloseKey, FindClose, VirtualFree, ...]
19:     for <each line of the input logFile> do
20:         API = the API call on this line
21:         returnValue = Return value of the API call on this line
22:         param = [Param1_value, Param2_value,..., Param_n_value]
23:         Identify Files as f, Registry as reg, Identifier as id and Buffer as buf from param and
    insert into RELATION                                  ▷ Identification is performed using regex
24:         if API in closure then
25:             for each param do
26:                 if item in Param belongs to TARGET: {Active_ids or Active_files or Ac-
    tive_regs or Active_bufs}
27:                 ADD_EDGE (API, item, "remove", TARGET)
28:             end for
29:         else
30:             if item in Param belongs to keyset of TARGET: {Active_ids or Active_files or
    Active_regs or Active_bufs} and API not in TARGET[item] then
31:                 ADD_EDGE (API, item, "update", TARGET)
32:             else
33:                 for each item in RELATION do
34:                     if item is not Null then
35:                         Insert into (Active_ids if item: id) or (Active_files if item: f) or (Ac-
    tive_regs if item: reg) or (Active_bufs if item: buf), where (Key, Value): (item, API)
36:                     end if
37:                 end for
38:             end if
39:         end if
40:     end for
41:     return Directed Graph: G
42: end function
```

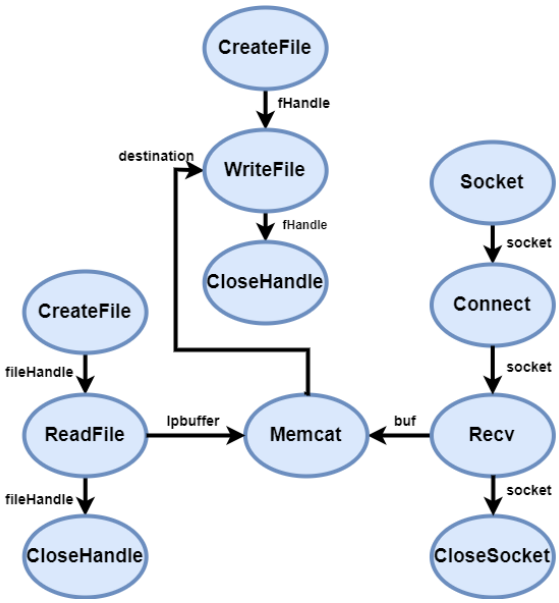



Figure 3.5: An MSG that corresponds to malware actions in a real-world attack scenario

because the parameter value 0x000000bc belongs to the keyset of Active_ids and the API RegQueryValue is not equal to Active_ids [0x000000bc], i.e., RegOpenKey (line 27), it invokes ADD_EDGE with (RegQueryValue, 0x000000bc, "update", Active_ids) as parameters in line 28. As a result, a new edge RegOpenKey → RegQueryValue is inserted to the MSG (lines 2-4) and the (key, value) pair in Active_ids is updated to (0x000000bc, RegQueryValue) in line 8. When MSG_Extractor processes line 4 of Figure 3.3, because the API RegCloseKey is in the "closure" set (i.e., the condition checked in line 22 is *true*) and the parameter value 0x000000bc belongs to the keyset of Active_ids (line 24), it invokes ADD_EDGE with (RegCloseKey, 0x000000bc, "remove", Active_ids) as parameters in line 25. As a result, a new edge RegQueryValue → RegCloseKey is inserted to the MSG (lines 2-4) and the (key, value) pair (0x000000bc, RegQueryValue) is removed from Active_ids (lines 5-6).

Note that MSGs can be created using multiple dependencies too. For example, for lines 10 to 17, graph G3 is formed where the first two nodes, *FindFirstFile* and *CreateFile*, are connected using *filepath*. However, *CreateFile* and the next five nodes are linked with the handle value *0x000000de*. Moreover, MSGs that describe the

real-world attack scenarios can be more complex than the ones in Figure 3.4. For example, the MSG in Figure 3.5 corresponds to malware actions that (1) read the content from a file, (2) call a network API to get some malicious code/data from a remote server, (3) add the malicious code/data to the content, and (4) write the manipulated content to another file. There are multiple parameters associated with this MSG but each one is used independently to infer the dependency among APIs. For example, the parameter *lpbuffer* of *ReadFile* is a reference to the memory block to hold the read-in content in step 1, and it becomes one input parameter of the *memcat* API in step 3. Thus, using this parameter (which is of "*Active_bufs*" type), we can discover the link between *ReadFile* and *memcat* in the extracted MSG."

Finally, if a WinAPI call is followed by the same WinAPI call and parameter values, the MSG extractor considers these multiple calls as a single call and creates only one node. In the case of G2, *Process32Next* is followed by multiple *Process32Next* calls with the identical parameter values; as a result, the MSG extractor ignores the following calls and creates only one *Process32Next* node. The MSG extractor extracts all these possible malicious sub-graphs and stores them for MSG classification.

### 3.3.2 MSG Classifier

To implement a successful deception framework, it is inevitable to understand what the malware is trying to achieve at the time of deception, hence, enabling us to design a deception ploy for the malware strategically. MSGs are a low-level characterization of malware that is semantically distant from high-level descriptions of malware behaviors and goals. This semantic gap needs to be bridged in order to identify the adversary's actions achieved by the malware. To attain this understanding, we map MSGs to their corresponding MITRE ATT&CK techniques. MSGs' mapping to MITRE ATT&CK techniques provides insight into the malware's various characteristics, thus enabling us to design an appropriate deception ploy. MITRE ATT&CK is a publicly accessible, structured knowledge base that contains adversaries' various tactics and

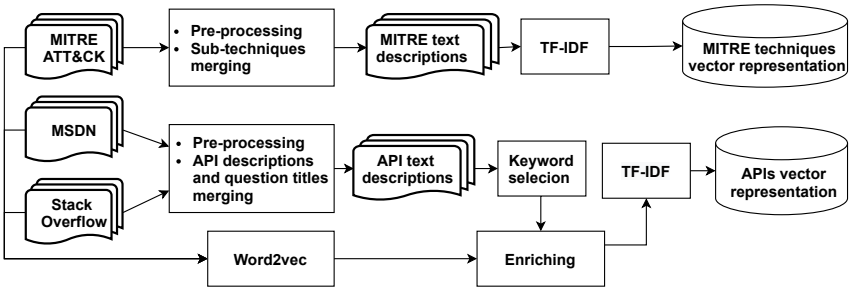



Figure 3.6: MITRE techniques and APIs vector representation extraction.

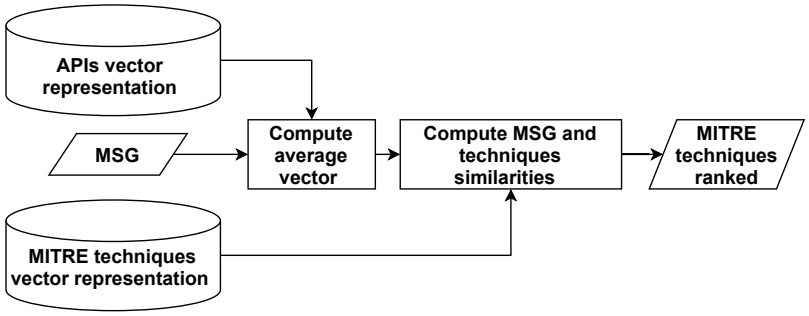



Figure 3.7: MSG to MITRE technique classification

techniques illustrating the attack lifecycle of the adversary. Each MITRE ATT&CK technique has a text description that explains the attack's procedures and objectives. As MSGs consist of a set of Windows APIs, and since rich information about each API can be collected from online resources, MSG Classifier performs MSG-to-MITRE ATT&CK techniques mapping by analyzing descriptions of APIs from MSDN, online API-related data, and descriptions of MITRE ATT&CK techniques. MSG Classifier consists of two main components. First is vector representation extraction, where the MSG Classifier generates a vector representation of each API and MITRE technique based on the collected related text. The second is MSG to MITRE classification, where MSG Classifier uses the generated vectors for real-time MSG to MITRE classification.

Figure 3.6 and Figure 3.7 show the process of the vector representation extraction and MSG to MITRE classification, respectively. This section describes the MSG Classifier's data, components, and classification process.

**Data Collection**: MSG Classifier maps MSG to a MITRE technique using MITRE

techniques' descriptions, APIs' descriptions, and Stack Overflow Windows-related question-answer pairs. Stack Overflow is a website where users discuss programming-related problems [58]. Each Stack Overflow post consists of a question and multiple answers provided by users. Since APIs' are low-level malware characteristics, whereas MITRE techniques are high-level attack descriptions, Stack Overflow was leveraged to bridge this knowledge gap. For example, a MITRE technique description can mention the "taking a screenshot" behavior, which is a high-level action that can be achieved in Windows using several APIs. Stack Overflow questions regarding taking a screenshot are leveraged by extracting APIs from users' highly rated answers.

The accuracy of our MSG to MITRE technique mapping critically depends on the quality of the StackOverflow dataset because StackOverflow is an open forum where users talk about many different things. To this end, we preprocess the original question-answer pairs to filter noise. Specifically, we include only StackOverflow questions with Windows-related tags such as winapi and win32. We extract APIs from the answers by first tokenizing the code part of the answers and only retrieving tokens that match an API from our predefined API list. Additionally, we consider an answer only if it mentions at least one APIs. To address the potential inaccuracies inherent in Stack Overflow content, such as incorrect answers or outdated information, we have implemented a set of rigorous criteria and mechanisms. These measures leverage the quantitative evaluation provided by Stack Overflow for each question and answer, ensuring that any noise introduced is statistically insignificant and does not affect the accuracy of our model. Firstly, we employ a filtering process to eliminate outdated answers. This involves verifying the validity of the mentioned APIs by cross-referencing them with the current API documentation. Answers that reference deprecated or non-existent APIs from MSDN are promptly removed from consideration. This meticulous approach guarantees that the data collected for our purposes remain entirely relevant and up-to-date. Secondly, a significant majority

(approximately 80%) of the collected data we consider has undergone validation and approval by the Stack Overflow user community, as indicated in the posts themselves. This validation is essential as it reflects the consensus reached by experts and knowledgeable individuals within the community. For the remaining data, we adopt a selective approach by only considering answers that have received a positive score. This implies that the number of positive reviewers significantly outweighs the number of negative reviewers. By incorporating this criterion, we further ensure that the information included in our dataset is not only validated but also favorably acknowledged by the Stack Overflow community. Through the careful application of these criteria and mechanisms, we have effectively filtered out any outdated or highly inaccurate information that could potentially compromise the integrity of our collected data. As a result, the noisy information is statistically insignificant in terms of its potential impact on our overall data accuracy.

After performing pre-processing and filtration, we ended up with a dataset consisting of 525 MITRE enterprise techniques, 7,241 APIs, and 10,289 StackOverflow question-answer pairs. We consider this dataset sufficient for the classification task described in the chapter. Our evaluation (Section 3.5.5) shows that our API representation is robust, as it can classify MSGs from real-world malware to the correct MITRE ATT&CK techniques.

**Preprocessing:** MITRE techniques' descriptions, APIs' descriptions, and Stack Overflow questions contain many unessential words called stop words, which are words that frequently occur in a text but with no contribution to its meaning. Examples of stop words are to-do verbs and prepositions (e.g., about, in, and below). We adopt the stop words list introduced by NLTK [59] in our analysis. We also added to the list some Windows-specific words such as "windows" and "c." We filtered out stop words from our data prior to any analysis. We also filter out low occurring words that occur less than a predefined number of times that we denote as minimum word

frequency. Low occurring words expand the space complexity of our task while not contributing to the overall results. Finally, we filter out punctuations and numbers, lemmatize verbs to their base form, and convert all text to lowercase.

**MITRE and API text representation:** We represent each MITRE technique by its main description and its procedure examples' description. A technique's procedure examples are a brief description provided by MITRE of how different malicious entities use the corresponding technique. For techniques that have sub-techniques, we merged all the sub-techniques descriptions and appended them to the parent technique's description.

We represent each API by three types of text: the API description taken from MSDN, all Stack Overflow questions' titles in which the API exists in the answers, and words that form the API name. For example, words that form the API `CreateFile` are Create and File. We merged all these texts for each API and used them as API text representation. We denote the new merged API description as API full description.

**MITRE vector representation:** We represent each MITRE technique by a vector. Each index of the vector corresponds to a word and its value is the TF-IDF score of the word. A TF-IDF score reflects the importance of a word in a document in comparison to other documents. This importance is measured based on the word's frequency in the current document compared to other documents. The TF-IDF score of a word $w$ in a document $d$ from the set of documents $S$ is:

$$\textit{tf-idf(w,d,S)} = \textit{tf (w,d).idf(w,S)}$$

Where:

$$\textit{tf (w,d)} = log(1 + freq(w, d))$$

$$\textit{idf (w,S)} = log(\frac{N}{count(d \in S : w \in d)})$$

Here, *freq(w,d)* is the number of times the word $w$ appears in document $d$, *count(d*

$\in S : w \in d)$ is the number of documents in which the word $w$ appears and N is the number of documents in S. Therefore, the TF score corresponds to the frequency of a word occurrence in a document, and the IDF score corresponds to the number of documents that contain the word. The TF score is directly proportional to the frequency of occurrence, while the IDF score is inversely proportional to the number of documents that contain the word.

The vector representation $V$ of a MITRE technique $m1$:

$$V_{m_1} = [\textit{tf-idf}(w_1, d_{m_1}, S), \textit{tf-idf}(w_2, d_{m_1}, S), ..., \textit{tf-idf}(w_n, d_{m_1}, S)]$$

We only keep the highest-scored words in each technique representation and set other words' scores to zero.

**API vector representation:** We extract each API vector representation in three steps: keywords extraction, enriching, and TF-IDF extraction. First, we calculate the TF-IDF score of all words in each API full description and extract nouns with a score above a predefined threshold. We denote these words as keywords. Second, we enrich each description using a Word2vec model we trained on MITRE technique descriptions, API descriptions, and Stack Overflow questions. Word2vec [60] is a word embedding model that represents words by vectors of multiple dimensions based on their co-occurrence. In Word2vec vector representation, words that highly co-occur in the training text will get similar vectors. This Word2vec behavior helps us to find words similar to words in API full description. We enrich each API full description by adding words similar to the description's keywords with a similarity above a predefined threshold. Third, we extract the TF-IDF score from the enriched descriptions, which will output a vector for each API similar to the MITRE technique vector we mentioned before.

**Classification:** When MSG Classifier gets an MSG as an input, it generates a

vector representation of the MSG by computing the average of the APIs' vectors of the MSG. When computing the average vector, all elements with a value of zero are discarded. For example, if an MSG containing 3 APIs and the values of the first element of each API's vector are 20%, 30%, and 0, the first element of the generated vector will have a value of 25%. MSG Classifier then calculates the cosine similarity between the average vector and all MITRE techniques' vectors. The higher the similarity, the more likely the MSG belongs to a technique. MSG Classifier outputs all MITRE techniques ranked by their similarity to the MSG. The MSG is then mapped to the highest-ranked technique.

### 3.3.3 Deception Factory Synthesis

In this section, we enumerate all possible Deception Ploys for different malicious behaviors, strategies and 4D deception goals (Section 3.3.3.1). It is possible to create a Deception Playbook covering all the ploys we listed. However, we choose to break down these ploys and build profiles based on the co-occurrence of certain behaviors. After building these profiles, we use the mapped MSGs to develop WinAPI hooks. We can use WinAPI hooks to intercept malware execution and change the response as we see fit. The specifics are provided in the following sections.

#### 3.3.3.1 **4D Deception Goals**

An attacker can be deceived in multiple ways. To design a deception framework, we must consider 4D deception goals: diversion, distortion, depletion, and discovery. Details are given as follows:

- **Diversion:** Diversion means leading an adversary to a fake target instead of a real one by giving deceptive responses to any adversary queries or providing honey information. For instance, diversion can be achieved by redirecting the adversary's exfiltration traffic to a decoy server.

- **Distortion:** Distortion creates confusion in the adversary's mind by providing

too many options and ambiguity in cyberspace. For instance, while an adversary searches for user credentials, providing multiple yet honey credentials will distort the adversary's view of the system.

- **Depletion:** Depletion means delaying the adversary's propagation in the attack kill-chain by consuming his power or resources. As an example, we can deplete a Ransomware by creating a delayed response without doing any encryption, but responding that encryption is not done yet.

- **Discovery:** Discovery means figuring out new attack techniques by letting an adversary executes its desired action but in a contained environment. It can be done by running malware inside a deception framework such as symbSODA.

#### 3.3.3.2 Deception Strategies

Since we implement the deception through API hooking, there are four ways we can respond to the malware: FakeFailure, FakeSuccess, FakeExecute, and NativeExecute. We call these approaches *Deception Strategies* and more details are given as follow:

- **FakeFailure:** By FakeFailure, we mean denying something that exists or the falsification of a true property or fact. Under this strategy, symbSODA responds to the malware as if the operation failed while the actual WinAPI has never been called. For example, if the malware calls `CreateProcess` to create a child process, symbSODA responds back to the malware that the child process creation failed without invoking the actual `CreateProcess` call.

- **FakeSuccess:** By FakeSuccess, we mean confirming something that doesn't exist or the confirmation of an invalid property or fact. Under this strategy, symbSODA responds to the malware as if the operation was successful but the actual WinAPI has never been called. For example, the malware calls ReadFile to read the content of a file, symbSODA can respond back to the malware that

Table 3.1: Deception ploy creation and verification ($D_1$ = diversion, $D_2$ = distortion, $D_3$ = depletion, $D_4$ = discovery).

| Malware Behavior | Mapped MSG (API Sequence) | Strategy | Deception Goal | | | | Deception Actions |
|---|---|---|---|---|---|---|---|
| | | | $D_1$ | $D_2$ | $D_3$ | $D_4$ | |
| Stealing from credentials files | 1) Search file and steal: FindFirstFile, PathFileExist, CreateFile, GetFileSize, VirtualAlloc, ReadFile, CloseHandle, VirtualFree, FindNextFile, FindClose<br>2) Read sensitive file (known file). CreateFile, ReadFile<br>3) Steal from the browsers. CreateFile, GetFileSize, VirtualAlloc, ReadFile, CryptUnprotectData, CreateFile, WriteFile, CloseHandle | FakeFaliure | ✓ | - | - | - | Diversion: pretend the File doesn't exist by returning false when PathFileExist is called |
| | | FakeSuccess | - | - | ✓ | ✓ | 1) Depletion: replace sensitive file reading with static HoneyFile containing Honey Credentials.<br>2) Discovery: watch out for Exfiltration. |
| | | FakeExecute | ✓ | - | ✓ | ✓ | 1) Diversion: forward the execution to HoneyFactory (false target).<br>2) Depletion: communicate with HoneyFactory. Ask for HoneyFile containing Honey Credentials. Replace sensitive data with the content of HoneyFile.<br>3) Discovery: watch out for Exfiltration. |
| | | NativeExecute | - | - | - | ✓ | Discovery: watch out for Exfiltration |

the file is opened and content is read without invoking the actual ReadFile call. In addition, static content can be provided by symbSODA for this ReadFile call. For instance, if the malware wants to read the content of a file named "input.txt," where the content is "It's a text file", symbSODA pretends the ReadFile was called and content of "input.txt" is read; however, it returns "It's **not** a text file" instead of "It's a text file" to the attacker.

- **FakeExecute:** By FakeExecute, we mean executing (on a remote honey factory) something that doesn't execute (on the victim machine). Under this strategy, the action of the malware is performed on a remote machine and symbSODA sends back the response it received by running the action on a remote machine. For example, the attacker wants to know the IP address of the victim machine; hence the malware is designed to run the "ipconfig" command on the victim machine. In this scenario, symbSODA intercepts this call and runs the "ipconfig" command on a remote machine (HoneyFactory) and sends the response back to the attacker. Therefore, the attacker receives the IP address of the remote machine (HoneyFactory).

- **NativeExecute:** Under this strategy, symbSODA lets the malware run so that it can discover the current/future actions of the malware.

#### 3.3.3.3 Deception Ploy Creation

Given 4D deception goals and four strategies, we can design sixteen possible deception techniques (or deception ploys) for each malicious behavior. However, not all of these techniques will make sense or applicable to deploy. As a result, we start by devising all possible/feasible deception techniques, then verify each technique and eliminate those that do not make sense. Even though multiple deception actions can be taken for a particular malware behavior, we only allow one ploy for each malicious behavior to be selected by users. For better understanding, we explain the creation, verification and filtration process in Table 3.1.

Assume the malware behavior is "Stealing from credential files." If we want to deceive this behavior with the `FakeFailure` strategy and `Diversion`, the deception action would be to pretend the sensitive file doesn't exist on the system; hence, the malware is diverted to `no target`. Now, if the deception strategy is `FakeSuccess`, we can achieve both `Depletion` and `Discovery`. In the instance of `Depletion`, the credential files can be leveraged to provide the attacker with fake login credentials and deplete her resources and effort. An even better strategy is to generate an encrypted version of an invalid password and forward it to the attacker, who must then decrypt the password, further depleting the attacker's resources. Since we provide the fake credentials and the malware performs read operations, the malware will likely try to exfiltrate the data to the attacker. As a result, this strategy also allows us to `Discover` another malicious behavior of the malware. This is how we identify all possible combinations of Deception Strategies and Goals that are valid and create deception actions for them. These valid deception actions are considered deception ploys and then utilized to develop `Deception Playbook`.

#### 3.3.3.4 Deception Playbook Creation

We perform frequent itemset mining to identify highly associated MSGs, which we then utilize to create Deception Playbook profiles. For example, let's say deception ploys (or deception techniques) $T_1, T_2, T_3, ..., T_6$ are designed for malicious behaviors $B_1, B_2, B_3, ..., B_6$, respectively. If $B_1, B_2$ and $B_3$ are in a frequently used itemset, we can create a profile $P_1$ that consists of $T_1, T_2$ and $T_3$. If the victim is infected with malware with the behaviors $B_1, B_2$ and $B_3$, we can deploy P1 to deceive the malware.

Frequent itemset mining is a data mining approach that identifies items that frequently appear together. In this scenario, a set of items (malicious behaviors) is considered frequent if it satisfies a minimal threshold value for *support* and *confidence*. The support indicates how frequently the set of malicious behaviors appears in the dataset, and confidence is the likelihood that if one behavior appears, the other behavior also appears. Let $B = \{B1, B2, B3, ..\}$ be a set of $n$ binary attributes called Malware Behaviors (items), then Support and Confidence can be defined as:

$$Support(B) = \frac{Number\ of\ malware\ in\ which\ B\ appears}{Total\ number\ of\ Malware\ in\ the\ dataset}$$

$$Confidence(B1 \rightarrow B2) = \frac{Support(B1 \cup B2)}{Support(B1)}$$

The frequent pattern mining technique is used to discover relationships between different items (malware behaviors) in a dataset (malware dataset). These relationships are represented in the form of association rules. Let $L = \{L1, L2, L3, ...\}$ be a set of log files collected from malware as execution traces. Each log in L represents a unique malware from our dataset and contains a subset of the Malware Behaviors (items) in B. An association rule is defined as an implication of the form $B1 \rightarrow B2$ where $B1, B2 \in B$, and the strength of the association between $B1$ and $B2$ can be measured by $Confidence(B1 \rightarrow B2)$ defined above. To discover frequent itemsets (malware behaviors) and association rules, we leveraged a well-known algorithm called *APRIORI*.

We calculate support and confidence for extracted MSGs and identify frequent malware behavior sets above the threshold value. Deception ploys for those MSGs with a high association are grouped and stored as profiles.

Deception Playbook stores all the profiles that have been developed, and only the relevant profile is deployed. Furthermore, symbSODA allows users to create their own custom profiles. Profiles indicate which deception ploys to be activated. A configuration file (in JSON format) is generated when a profile is created or selected, and it is then utilized by WinAPI hooks to determine which ploys to activate.

#### 3.3.3.5 Assume-guarantee analysis to ensure non-conflicting deception actions selection

As we let the users create their profiles with desired deception actions, users may select conflicting deception actions. Therefore, we introduced an *Deception Planning Verifier*, which utilizes assume-guarantee analysis to ensure the users select valid deception actions while creating their profiles. In this way, we keep the deception actions consistent to the attacker throughout the execution of the malware.

Assume-guarantee reasoning [61] is used to address the issue of compositional system analysis by applying verification to individual components and merging the results without analyzing the whole system. In checking components individually, it is often necessary to incorporate some knowledge of the context in which each component is expected to operate correctly. Assume-guarantee reasoning tries to capture the expectations that a component has about its environment by using assumptions. Therefore in our case, if we have N number of deception actions such as $\{DA_1, DA_2..., D_N\}$, if a user selects $DA_1$ at first, then $DA_2$, while selecting $DA_2$, we perform the assume-guarantee analysis with the context of prior selections (in this case, $DA_1$) to ensure the selected deception actions do not conflict hence operate correctly.

Now, let's rephrase this idea at the API level. The deception actions are focused on

the extracted MSGs, and multiple MSGs might share a common subset of APIs. Since the hooking is done at the API level, it is essential to keep track of the previously visited APIs to understand the exact context. We use flags across these API hooks to track the context as well as relevant system values, such as handles, filenames, and registry names. These system values indicate the state of the system before and after the malware/application calls a particular API. We call these before and after state values "pre-condition" and "post-condition," respectively. Therefore (a) the pre-condition of the current API depends on the historical data (the APIs that are called before the current API), and (b) the post-condition relies on identifying the current state (it belongs to which MSG) and the deception action taken based on user selection. Therefore, the values of the pre-conditions and the post-conditions can be different. The common APIs among different MSGs might be subject to create conflicting pre-condition and post-condition values.

We apply Assume-guarantee analysis to automatically validate the chosen deception actions by the users and guide them in selecting non-conflicting deception actions. For all the API calls (attack actions), we generate the pre-conditions and post-conditions of the four possible deceptions. For every pair of attack actions $A_1$ and $A_2$ that may happen consecutively, we check the post-condition of deception actions $NativeExecution$, $FakeExecute$, $FakeSuccess$, $FakeFailure$ of $A_1$, and pre-condition of all other deception actions for $A_2$, $A_3$, ..., $A_n$ pairwise. For simplicity, we formalize the following equations in a way that we check the pre-conditions and the post-conditions of $A_1$ and $A_2$. However, in reality, we repeat this for each possible attack action pair. Since for each API, we can take four possible deception actions, we have 16 possible combinations of deception actions for an API pair ($A_1$ and $A_2$), which can be denoted as:

$$(POST_{1i}, PRE_{2j}), 1 \leq i, j \leq 4$$

For every such combination, we verify if the following logical condition is satisfied

$$POST_{1i} \rightarrow PRE_{2j}, 1 \leq i, j \leq 4$$

Note that here $POST_{1i}$ and $PRE_{2j}$ can be formulated as:

$$a_1 = 1|2|3, a_2 > 10, a_3 ! = 2$$

where comma means logical AND, | means logical OR. If the post-condition $POST_{1i}$ is

$$a_1 = 1, a_2 = 1|2, a_3 < 2$$

and the pre-condition $PRE_{2j}$ is

$$a_1 = 1|2|3, a_2 > 0, a_3 < 2$$

the logical condition can be satisfied and and the two deception actions can be composed.

As another example, if the post-condition $POST_{1i}$ is

$$a_1 = 1, a_2 = 1|2|3, a_3 = 2$$

and the pre-condition $PRE_{2j}$ is

$$a_1 > 0, a_2 = 2, a_3 > 1$$

the logical condition is not satisfied, since $a_2$ must be 2 but in the post-condition in $POST_{1i}$, $a_2$ can be 1, 2, or 3. This means the two deception actions cannot be composed. With this analysis, we can provide users the guidance of choosing feasible

Table 3.2: List of the Pre and Post conditions for each API(selective cases, the actual table consists of all possible combinations)

| Pre-Conditions | API | Strategy | Post-Condition |
|---|---|---|---|
| 1. - | CreateFile (Any) | FakeFailure | fileHandle = 0 |
| 2. fileExist=True | | NativeExecution | fileHandle = 1 |
| 3. fileExist=False | | NativeExecution | fileHandle = 0 |
| 4. fileExist=True\|False, searching_flag=1, exist_path_flag=1, internetHandle=0 | CreateFile (Credential Stealing) | FakeSuccess/FakeExecute | fileHandle = 1, credential_flag=1, uploadfile_flag=0 |
| 5. fileExist=True\|False, searching_flag=0, exist_path_flag=0, internetHandle=1 | CreateFile (Upload File) | FakeSuccess/FakeExecute | fileHandle = 1, credential_flag=0, uploadfile_flag=1 |
| 6. - | ReadFile (Any) | FakeFailure | - |
| 7. createprocess_flag=0, searching_flag=1, exist_path_flag=1, internetHandle=0, uploadfile_flag=0, credential_flag=1, remote_flag=0, fileHandle=1 | ReadFile (Credential Stealing) | FakeSuccess | Buffer=StaticBuffer |
| 8. createprocess_flag=0, searching_flag=1, exist_path_flag=1, internetHandle=0, uploadfile_flag=0, credential_flag=1, remote_flag=0, fileHandle=1 | | FakeExecute | Buffer=HoneyBuffer |
| 9. createprocess_flag=0, searching_flag=1, exist_path_flag=1, internetHandle=0, uploadfile_flag=0, credential_flag=1, remote_flag=0, fileHandle=1 | | NativeExecution | Buffer=DefaultBuffer |
| 10. createprocess_flag=0, searching_flag=0, exist_path_flag=0, internetHandle=1, uploadfile_flag=1, credential_flag=0, remote_flag=0, fileHandle=1 | ReadFile (Upload File) | FakeSuccess | Buffer=StaticBuffer |
| 11. createprocess_flag=0, searching_flag=0, exist_path_flag=0, internetHandle=1, uploadfile_flag=1, credential_flag=0, remote_flag=0, fileHandle=1 | | FakeExecute | Buffer=HoneyBuffer |
| 12. createprocess_flag=0, searching_flag=0, exist_path_flag=0, internetHandle=1, uploadfile_flag=1, credential_flag=0, remote_flag=0, fileHandle=1 | | NativeExecution | Buffer=DefaultBuffer |

deception actions. Note this analysis can be extended to attack action sequence longer than 2. We implemented this algorithm to automate the deception feasibility verification process. Table 3.2 depicts the pre-conditions and the post-conditions that we list out for representative APIs.

In addition, the assume-guarantee not only helps the users in selecting correct non-conflicting deception actions but also helps the deception action developers to identify the conflicts they may create while developing the deception actions. Let's explain the process of removing conflicts using an example with reference to Table 3.2.

**For the users:** An attacker may use malware that directly reads from the sensitive file and exfiltrates the credentials. Alternatively, the attacker can upload the credential file and perform the read operation later on the attacker's machine. If the user selects different deception actions for these two behaviors, the attacker can easily detect the inconsistency and realize our deceptive system. For example, in Table **??**, we can see the user selected the "FakeSuccess" strategy for the behavior "Credential Stealing"; as a result, honey static credentials are exfiltrated to the attacker. Now, if the user selects "FakeFailure" for the "Upload file," then the system will pretend that the credentials file doesn't exist, hence nothing is extracted. Unfortunately, the selection of these conflicting deception actions will reveal our deception to the attacker through inconsistent data (i.e., the existence of the credentials). We identify these conflicts using assume-guarantee during the deception action selection by the users and disable the conflicting actions. In Section 4.4, we discuss in details how the

users select deception actions through an interface and how we disable the conflicting deceptions by performing the assume-guarantee analysis.

**For the deception developers:** Let's say the user selects "FakeFailure" strategy to deceive the behavior "Upload File" and "FakeSuccess" strategy to deceive the behavior "Credential Stealing". As we can see from Table **??**, CreateFile and ReadFile are shared between two MSGs, and therefore, it is necessary to identify the MSG these shared APIs belong to at runtime. To do so, it is essential to set the correct flags and track the system properties. However, the developers might set the wrong flags or fail to track the system properties correctly. As a result, the deception system will not behave as expected. Based on the API invocation history, it is possible for the same API to have different flags as the post/pre conditions. For example, in Table **??**, we can see that ReadFile is part of two malware behaviors (Credential Stealing and Upload File); however, the sets of APIs being called before ReadFile are different. It is essential to understand which behavior a particular ReadFile invocation belongs to and exactly which deception action to deploy. To do so, developers can use the flags listed in the 1st column of Table 3.2 (such as uploadfile_flag and credential_flag) to keep track of the context. If any of these flags are set incorrectly, our system will become inconsistent or even crash. As a result, our deception might get revealed to the attacker.

In this paragraph, we discuss exactly how these conflicts arise and how they are identified using assume-guarantee. For example, in the case of the first row of Table **??**, where the malware behavior is "Credential Stealing" and the strategy is "Fake-Success", the corresponding MSG has both CreateFile and ReadFile. Now, let's assume the developer sets the uploadfile_flag=1 instead of the credential_flag=1 during the CreateFile invocation (cross reference to the 4th row of Table 3.2) by mistake, then the system won't be able to execute the correct deception action for "Credential Stealing" as it won't identify the ReadFile call as part of this behav-

ior. In such scenarios, the assume-guarantee checking will let the developer identify setting the wrong flags. The developer creates the pre and post conditions for each hook function and feeds them to the *Deception Planning Verifier* to identify potential mistakes. The expected outcome for the user selected "FakeSuccess" strategy for "Credential Stealing" is to fill the ReadFile buffer with "StaticBuffer" (row 7 in Table 3.2). To achieve that, the flag values in the precondition should be createprocess_flag=0, searching_flag=1, exist_path_flag=1, internetHandle=0, uploadfile_flag=0, credential_flag=1, remote_flag=0, and fileHandle=1. Since the developer by mistake sets the uploadfile_flag=1 and credential_flag=0, the *Deception Planning Verifier* notifies the developer that the expected outcome/post condition cannot be achieved because the required precondition is not satisfied.

We run the *Deception Planning Verifier* twice: 1) while the user selects deception actions and 2) after the developer writes new hook functions. Hence, we ensure the removal of conflicts in both cases.

#### 3.3.3.6 Deception Factory Implementation

The deception factory is the implementation of the actions of the deception playbook. We use WinAPI hooking (also known as API hooking) to implement deception actions. Our API hooks are created as DLL files (which we refer to as *End-Point DLL*) and injected into malware via the DLL injection approach. API hooking is a well-known technique for intercepting API calls made by the targeted executable, allowing us to monitor or alter API responses. We implemented API hooking using EasyHook [57].

**How API hooking works**: EasyHook [57] is a free, open-source hooking library for 32-bit and 64-bit Windows processes released under the MIT license in this project. EasyHook provides a generic template for APIs hooking and ensures thread safety by using a thread deadlock barrier. EasyHook utilizes the CreateRemoteThread function to create a thread in the target process and use it to load the desired DLL. We install

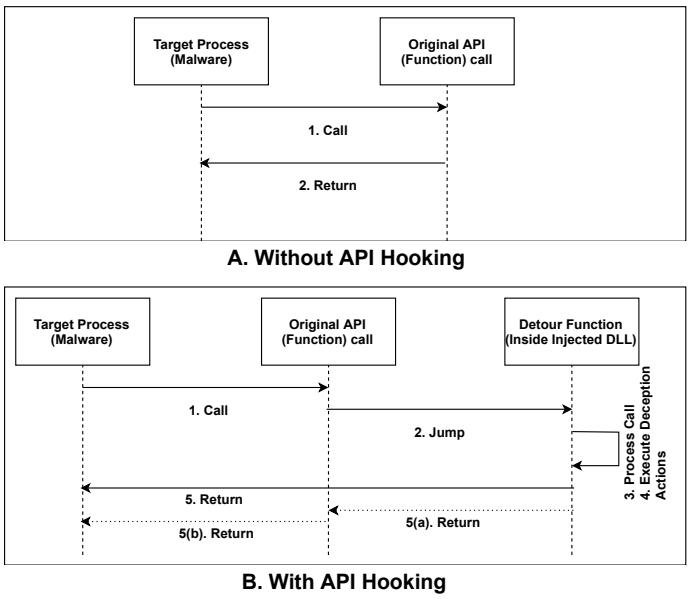



Figure 3.8: Call flow with and without API Hooking. In some case, Response is return via the original API (5a, 5b), otherwise, Detour function responds directly (5).

the hook by defining a detouring function for each original WinAPI call we want to intercept. As a result, the detouring functions take control of the malware and provide us with the option to enforce our code to modify the execution. Without API hooking, the API call flow is depicted in Figure 3.8A, where the original API is responsible for performing the task and returning a response. With API hooking, the call is jumped from the original API to our detour function (Figure 3.8B), where we process the call and execute our deception techniques and respond back to the malware.

**How deception technique works inside the hooks:** We hook some WinAPIs (MSG) mapped to each malicious behavior to deceive it. Inside these hooks, deception ploys are implemented and embedded. The following will discuss how the deception technique is implemented inside API hooks using Table 1, where the malicious behavior is Stealing from credentials files. To be more specific, we focus on the deception action "providing a HoneyFile containing Honey Credentials from the HoneyFactory" where the strategy is `FakeExecute` and Deception Goal is `Depletion`. Let's focus

```
1  known_sensitive_files_browsers = {"Login Data", ....}
2  malicious_behavior = 0
3  buffer_under_observation = NULL
4  HookFunction_CreateFile(){
5      Check if this Hook is in the activation_ploy_id List, If not exit without doing anything
6          Check lpFileName //The name of the file opened
7          If it's in known_sensitive_files_browsers {
8              //(for internal understanding), 5 indicates stealing credentials from browsers
9              malicious_behavior = 5
10             Handle_under_observation = CurrentFile_Handle
11         }
12         else{
13             .....
14         }
15 }
16
17 HookFunction_ReadFile(){
18     Check if this Hook is in the activation_ploy_id List, If not exit without doing anything
19         Check Handle //The name of the file opened
20         If it's same as Handle_under_observation {
21             //this buffer holds sensitive data
22             buffer_under_observation = buffer_under_observation.append(lpBuffer)
23         }
24         else{
25             .....
26         }
27 }
28
29 HookFunction_CryptUnprotectData(){
30     Check if this Hook is in the activation_ploy_id List, If not exit without doing anything
31         If malicious_behavior == 5{
32             //the malware is trying to decrypt the credentials
33             Check pDataIn->pbData and buffer_under_observation
34             If same{
35                 // change the buffer value that holds the decrypted data
36                 If the strategy is FakeExecute {
37                     //Communicate with HoneyFactory using REST API
38                     honey_file_content_from_HoneyFactory =
39                         AlterData("context": "5",
40                             "browser": "Google Chrome", "filename": "Login Data")
41                     pDataOut->pbData = honey_file_content_from_HoneyFactory
42                 }
43             }
44         }
45 }
```

Figure 3.9: A code snippet: How the deception technique is implemented inside API hooks

on the mapped MSG for this behavior: CreateFile-GetFileSize-VirtualAlloc-ReadFile-CloseHandle-CryptUnprotectData-CreateFile-WriteFile-CloseHandle. We don't need to hook all of these APIs to implement or achieve our deception strategy; only `CreateFile`, `ReadFile` and `CryptUnprotectData` need to be hooked. Let's explain how the deception is implemented inside the API hook using Figure 3.9.

The variable `known_sensitive_files_browsers` stores a list of known browser files that the attackers usually target for stealing credentials (at line 1). We used the variable `malicious_behavior` to flag what/which malicious behavior is likely to occur. We assign an ID to each malicious behavior, which we set to this variable for internal tracking. When the `CreateFile` API is called, we check which file the malware opened (at line 6). If the file is a sensitive file (in line 1), we set `malicious_behavior` to 5 (ID:5 indicates stealing credentials from browsers). It means there is a high possibility of malicious behavior "stealing credentials from browsers" to occur. We assign the current file handle to the variable `Handle_`

`under_observation` as any operation on this file handle is suspicious and should be under investigation. On the next `ReadFile` invocation, we check the file handle to ensure the malware performs a read operation on the target file (at line 20). The `ReadFile` operation assigns the data read from the sensitive file to `lpBuffer`. The data held by `lpBuffer` is critical because it contains the credentials. However, as the data is encrypted, the malware uses `CryptUnprotectData` to decrypt data that reveals the credentials as plaintext. Therefore, when `CryptUnprotectData` is called, we check if the data to be decrypted is the same as the `buffer_under_observation` (at line 33), and if it is, we interact with HoneyFactory through REST API and request for a HoneyFile containing HoneyCredentials (at line 38). `CryptUnprotectData` decrypts data and stores the data in `pDataOut->pbData`. We replace this `pbData` buffer with the content of the HoneyFile received from the HoneyFactory (at line 41). Ultimately, the credential the malware obtains is the HoneyCredentials instead of the actual user credentials. Note that we check if the API belongs to the activated ploys depending on a configuration file at the start of each detour function (lines 5, 18, and 30).

**Honey Factory (HF) Creation:** symbSODA often depends on external HoneyFactory (HF) to deceive malware, mainly when the Deception Strategy is FakeExecute. We develop HF offline based on the requirements of these actions/ploys since deception ploys are pre-defined and their actions are known. The HF consists of honey files, credentials, passwords, decoy user accounts, email accounts, web pages, decoy process lists, honey registry files etc. HF also hosts REST APIs to provide service to the API hooks.

## 3.4 Real-time Orchestration

This section discusses how symbSODA orchestrates and tackles real-world malware at run-time. This phase consists of four components: Orchestration Engine Server (OES), Orchestration Engine Client (OEC), Detection Agent and HoneyFac-

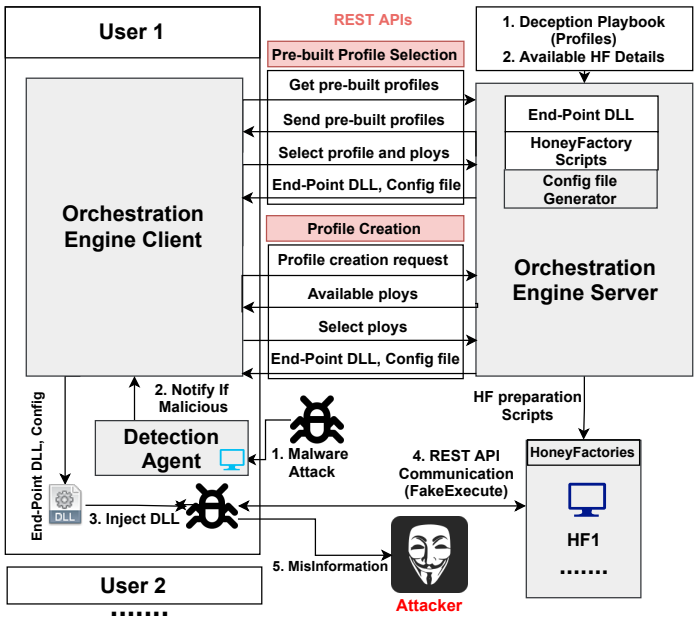



Figure 3.10: Communication among symbSODA's different segments, agents and the victim under attack.

tory (HF). The OES is the core component since it provides deception as a service and controls HF. The components use REST APIs to communicate among themselves. HF scripts, End-Point DLL, and deception profiles developed in the preceding stage are stored in OES. At first, the users install the OEC, which serves as an interface between the victims and the OES. Users can choose to use the pre-built profiles or create new profiles based on their needs. The whole procedure of this phase is depicted in Figure 3.10 and explained in the rest of this section.

**Profile Creation through interface:** The user sends a profile creation request to the OES through a user interface. First, OEC shows the user all the malicious behaviors and their corresponding valid deception ploys (actions). Next, the user selects the ploys she wants to use for orchestration. Our user interface is depicted in Figure 3.11. We carefully control the allowed combinations of deception ploys in the user interface to avoid potential conflicts. For each chosen ploy we perform the assume-guarantee based conflict checking and disable the actions that might cause conflict from the interface so that the user can not select them. Recall that in Table **??**, if the user selects "FakeSuccess" for the behavior "Credential Stealing", we

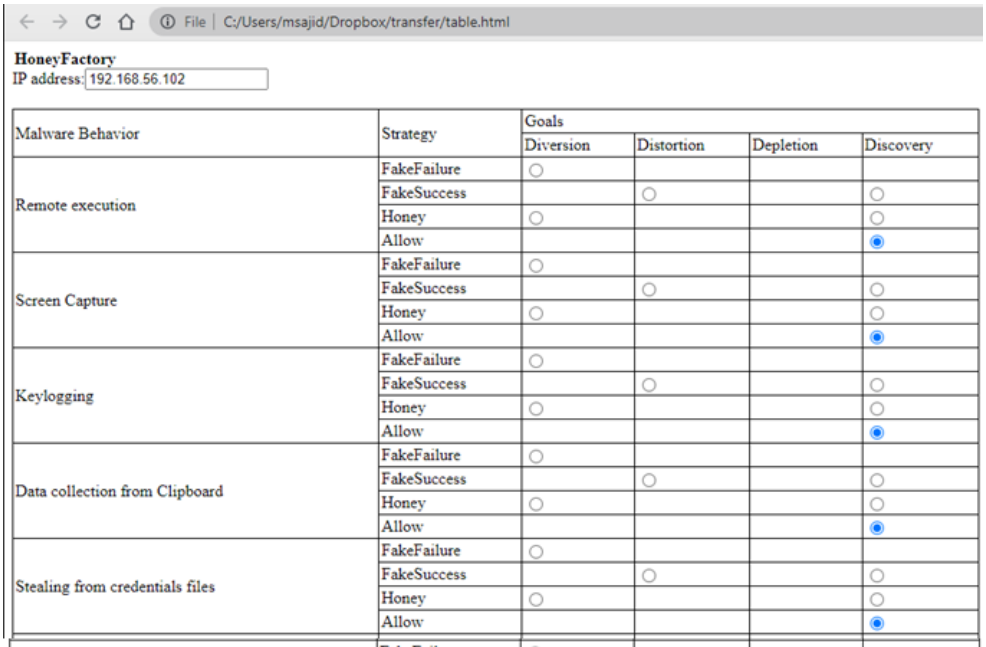



Figure 3.11: User interface to select deception actions for different malware behavior.

disable the "FakeFailure" strategy under the behavior "Upload file", hence avoid potential conflicts. We described the whole procedure in Figure 3.12. At first, for the selected deception action (A1) involving the behavior (B1), the *Deception Planning Verifier* identifies APIs representing the behavior (B1). Then, the *Deception Planning Verifier* analyzes the pre/post conditions of the involving APIs with all other APIs for different deception actions using Assume-Guarantee to identify conflicts and their involving APIs. Next, it determines behaviors (B2..Bn) involving conflicting APIs and corresponding Deception Actions (A2....An) and disables them from the interface so that the user can no longer select them. Based on the selection, OES prepares HF and generates a configuration file. This configuration file is an input to the End-Point DLL and is responsible for enabling and disabling ploys. Next, OES forwards the configuration file and the End-Point DLL to the user and deploys the required HF automatically.

**Pre-built Profile Selection:** Users can choose from one of our pre-built profiles

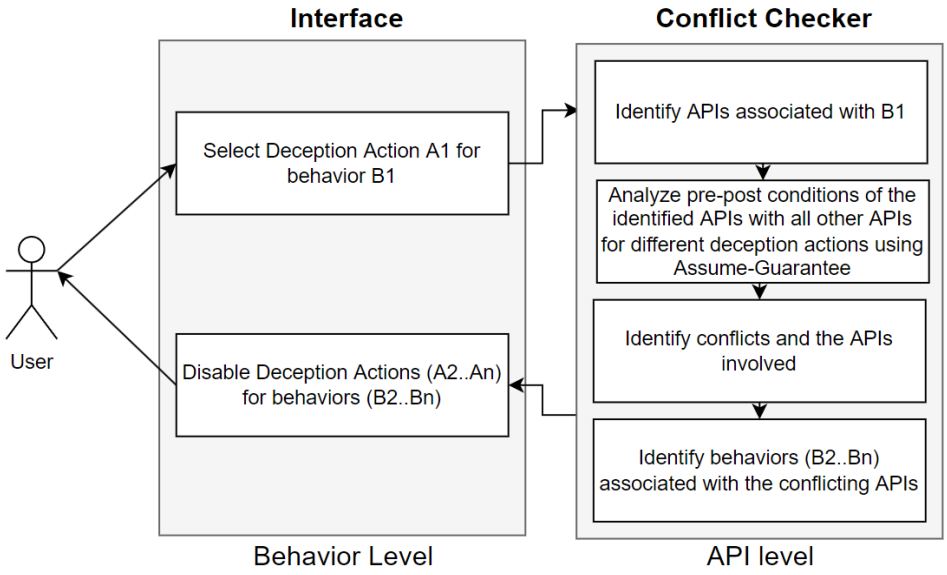



Figure 3.12: The process of identifying conflicting deception actions and disabling them on the interface level

rather than generating a new one. The user is shown suitable deception ploys for each pre-built profile. Once the user makes a selection based on her requirements, the rest of the process is the same: HF preparation, config file generation and deployment. At this point, symbSODA is equipped to deceive malware with its arsenals.

**Detection Agent:** The *detection agent* is the entry point of symbSODA, which detects malware and initiates the rest of the orchestration process. However, the primary goal of this research is not to detect malware samples; instead, we are solving the research problem of designing a cyber deception system that can provide dynamic orchestration at run-time. Therefore, symbSODA can leverage existing defense/mitigation systems and zero-day malware detection approaches as our detection agent [10, 39]. When the *detection agent* detects a malware process, it notifies the OEC. OEC injects the End-Point DLL into malware memory via the DLL injection approach, triggering real-time orchestration.

**Real-time deception using embedded API-hooking:** Once the detection agent confirms malware presence, OEC injects the End-Point DLL into the malware process. Such injections must happen in time to deceive the entire malware execution

period, which is feasible because malware detection systems usually suspend a process while checking and resume it if not malicious. For example, Windows Defender registers a process-creation callback routine (in its device driver WdFilter.sys [62]), which is called whenever a process is created. This callback routine checks if a new process is a malware before it runs. symbSODA can be integrated with Windows Defender's callback mechanism so that when a malware process starts, the OEC receives a notification and injects the end-point DLL into the malware process. Now, Let's describe how the orchestration works in this phase with an InfoStealer named LokiBot, which tries to steal credentials from the browsers. The user-selected strategy is FakeExecute and the 4D goal is Depletion. The deception action for the aforementioned adversarial action is redirecting the calls to a HoneyFactory and altering the actual credentials (Login Data) with honey credentials. We observed the malware tried to read from the file: (`C:\Users\Administrator\AppData\Local\Google\Chrome\UserData-\Default\Login Data`), then to decrypt credentials it calls `CryptUnprotect Data`. The injected DLL monitored the invocation of these APIs and determined the malware attempting to steal credentials from browsers. Hence, when the malware invoked `CryptUnprotectData`, the embedded hooking communicates with HF, asks for a HoneyFile containing HoneyCredentials and replaces the read result buffer with the content of the HoneyFile. As a result, ultimately, the malware gets the HoneyCredentials instead of the actual user credentials.

## 3.5 Evaluations

We assess symbSODA with different types of recent malware (InfoStealers, RATs, ransomware and spyware) created between 2019 and 2021. The objective of our experiments is to validate the accuracy, effectiveness, overhead and scalability of symbSODA, as well as the recall of MSG extraction and MSG-to-MITRE mapping. In addition, we performed case studies with three different malware to show how symbolic execution facilitates going past the malware triggering conditions, evasion

```
DWORD_PTR dwProcessAffinity, dwSystemAffinity;
GetProcessAffinityMask(GetCurrentProcess(), &dwProcessAffinity, &dwSystemAffinity);
if (dwSystemAffinity == 1)
    ExitProcess(0);
```

Figure 3.13: Malware source code: evasion check

technique within the malware, and extracting new MSGs that could not be retrieved without multipath exploration.

### 3.5.1 Case studies

#### 3.5.1.1 **Bypassing evasion techniques**

Malware authors use various techniques to identify a sandbox. For example, most sandbox solutions are designed to analyze multiple samples in parallel. To achieve this, they assign minimum hardware resources to each sandbox instance — including the least possible amount of RAM, disk space, and processing power. However, because very powerful systems are widely available today at inexpensive prices, real systems tend to be configured with maximum, or at least numerous, resources. Malware utilize these discrepancies in hardware to evade the analysis. Malware can call GetProcessAffinityMask API to obtain the CPU core count. The API returns the mask, which easily identifies the number of cores in use. If the returned core numbers is not as expected the malware will simply quit without showing any malicious behavior.

**Analysis** We perform analysis on a recent malware (MD5: 53f6f9a0d0867c10841b81 5a1eea1468) that utilizes this evasion technique. This is a cradle ransomware. Therefore, to uncover it's malicious behavior we want to go past this evasion check and let the malware reach a point where it starts the encryption. We ran the sample in both SODA and any.run [63] and the sample didn't show any significant behavior.

**Execution with symbSODA** Now we try to uncover the malware behavior using symbSODA. We hooked GetProcessAffinityMask API; as a result, when malware invokes this API, the call comes to our detour function. Before talking about the

symbolic execution, let's first focus on the malware source code and how the malware evades through this trick. In Figure 3.13, we can see the malware checks the affinity mask for the system, which indicates the number of cores the system has and if it is equal to one (1), then the malware process exits.

At this moment, we enable the symbolic execution and mark the variable that receives the affinity mask for the specified process as symbolic. As a result, we could observe a fork at the If condition checking and symbSODA keeps logging the execution traces of both paths. Now, one path doesn't show much as the malware quits in that path. However, the other path leads the malware to its encryption process. At the end of the exploration, we utilize the constraint solver to concretize the symbolic variable. We obtain the value for dwSystemAffinity, which reports a value greater than 0x2 for the CPU's number of cores. By concretizing these solutions in the memory of the concrete process, we can observe the execution of the malicious code hence the MSGs, which we couldn't observe using SODA.

#### 3.5.1.2 **Detecting killswitch within a piece of malware**

The famous WannaCry contains a kill switch that can cause the malware to terminate itself and thus stop spreading, according to security researchers. WannaCry infects systems through a malicious program that first tries to connect to a given web domain. The kill switch appears to work like this: If the malicious program can't connect to the domain, it'll proceed with the infection. If the connection succeeds, the program will stop the attack.

**Analysis** In this case study, we analyzed a recent malware (SHA1 hash 24d004a104d 4d54034dbcffc2a4b19a11f390 08a575aa614ea04703480b1022c) containing the killswitch and explained how to detect and react accordingly using symbSODA. Therefore, our target is to fool the malware by pretending the web domain is up and running using symbSODA and thus activate the killswitch within the malware to shut it down.

**Execution with symbSODA** Therefore, we performed some manual analysis on

the malware to find that the malware calls InternetOpenA, InternetOpenUrlA and InternetCloseHandle to open a connection to the killswitch URL (hxxp://www[.]iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com). Since the malware doesn't check the return value of InternetOpenA, we do not have to hook InternetOpenA. Otherwise, we might have to hook InternetOpenA and force it to return some dummy, non-NULL value. Therefore, hooking InternetOpenUrlA and executing it symbolically should be enough to trigger the killswitch.

In this case, we mark the return value of InternetOpenUrlA as symbolic. As a result, the program is forked, resulting in one state where InternetOpenUrlA returns a dummy handle and another state where InternetOpenUrlA returns NULL (which means failure). We followed the steps we did earlier and kept logging the execution traces. With respect to the API calls made by the WannaCry executable, we could see that in state 0, the malware called TerminateProcess right after the InternetOpenUrlA call was successful and InternetCloseHandle closed the handle. This indicates that the killswitch was triggered. Now, in the other state 1, we could see the malware called APIs to start service, load resource, create a child process, and perform encryption. In this scenario, symbSODA helped us identify the killswitch and enable it while the regular SODA could not.

#### 3.5.1.3 **Detecting exfiltration behavior**

Information stealers, or simply InfoStealers, are a type of malware that searches and collects sensitive and personal information (such as credit card numbers, cryptocurrency wallets, browser data and email credentials) from the victim machine and exfiltrates the sensitive data to the adversary. This kind of malware targets specific applications or files to collect information. Therefore, if the target files and applications do not exist on the system, the malware cannot exfiltrate them.

**Analysis** In this case study, we analyzed a recent malware (MD5: eccad903b4c27d149e159338f58481a9) known as LokiBot. The malware targets the browsers and FTP

clients to collect sensitive information and exfiltrate them. Therefore, the malware doesn't show the exfiltration capability if the target files do not exist on the system. In the following paragraph, we explain how symbSODA can detect the exfiltration behavior of the malware and deceive it with fake credentials.

**Execution with symbSODA** We analyzed the malware manually and observed that the malware has a list of target applications/files. At first it performs file search operation using APIs such as FindFirstFile and PathFileExists. If these APIs do not return the expected result then the malware doesn't perform the file reading or exfiltration operation. As a result, we won't be able to observe this path of the execution. Therefore, to uncover this behavior, we mark the return values of these two APIs as symbolic. However, just marking the return value symbolic is not sufficient; we also created the files being searched by the malware with honey content using the approach mentioned in [10]. As a result, we observed with symbSODA that the malware performed the read operation on the honey files and eventually exfiltrated their content. Therefore, we could identify the MSGs regarding operations like file read and exfiltration which we couldn't observe using SODA.

### 3.5.2 Evaluation of MSG extraction

#### 3.5.2.1 **Ground-Truth (GT1)**

We created a ground truth that can be utilized to validate the accuracy of the MSG extraction. Malware source codes can be found on GitHub; specifically, we are interested in the malware source code written using WinAPIs and C++. We downloaded 45 malware source codes from GitHub such as [64, 65], along with the comments and descriptions explaining the malware's capabilities. In our context, we consider the malware capabilities as malware behaviors. We manually went through these source codes, identified the 98 distinct API sequences (in our context, MSGs), and mapped them to 34 associate malware behaviors (according to comments and description). In addition, we manually mapped these 34 malware behaviors to MITRE

techniques. This ground truth is referred to as GT1 and will be used in further evaluations.

#### 3.5.2.2 Evaluation Metrics and Expectations

We obtain the binaries by building the source code of the downloaded malware. We ran them through our API Call Tracer and extracted the execution traces. The expectation for GT1 is that the MSG extractor should extract these 98 distinct API sequences (in our context, MSGs) from the traces. However, the MSG extractor may extract more MSGs than expected due to two reasons, 1) Some WinAPIs contain internal calls to other WinAPIs that are not apparent in the source code. 2) If the program does not specify memory management, Windows manages it (APIs such as VirtualAlloc and VirtualFree appear in the trace even though they are not in the source code). As a result, accuracy/precision is not the appropriate metric for evaluating our MSG extraction; instead, recall is the ideal metric for evaluation because it represents relevant instances that were retrieved.

#### 3.5.2.3 Result obtained from the malware datasets regarding GT1

We then fed the retrieved traces from the previous step into the MSG extractor, which retrieved 121 unique MSGs. We manually went through these MSGs and confirmed 96 of them are as expected (belonging to the ground-truth). However, we validated that the remaining two expected MSGs were retrieved as well, but the extracted API sequences differed from the expected ones due to an internal call to other WinAPIs. We also ran the same experiment with our previous work, SODA [52], which could identify 92 out of these expected 98 MSGs. SODA couldn't identify the other four as the environment didn't meet the malware expectation and the malware evaded the analysis or quit early. In such cases, symSODA outperforms SODA as it has symbolic execution capability to provide the correct environmental values and go past the evasive point. MSG extraction result is shown in Table 3.3. Recall value for

Table 3.3: Comparison with other State-of-the-art tools in terms of discovering malware behaviors/capabilities using GT1 and their individual recall values

| **Tools** | **Malware Behavior/Capability Identification (Using GT1)** | | |
|---|---|---|---|
| | Identified | Not Identified | Recall |
| **Kris et al. [66]** | 60 | 38 | 0.61 |
| **FORECAST [67]** | 33 | 65 | 0.34 |
| **DodgeTron [10]** | 9 | 89 | 0.09 |
| **SODA [52]** | 92 | 6 | 0.94 |
| **symSODA** | 96 | 2 | 0.98 |

our MSG extraction approach is:

$$Recall_{GT1} = \frac{TP}{TP + FN} = \frac{96}{96 + 2} = 0.979$$

The recall value is promising and demonstrate the effectiveness of our MSG extraction procedure.

### 3.5.3 Comparison with other state-of-the-art tools in terms of discovering malware behaviors/capabilities

We empirically compared symbSODA with Kris et al. [66], FORECAST [67], DodgeTron [10] and SODA [52] that are capable of discovering malware behavior/capabilities. For comparison, we used the collected traces (obtained by the API tracer) of the 42 malware used to build GT1. We fed these traces to different tools and observed how many behaviors/capabilities were found by each tool. The comparison is presented in Table 3.3. We found that symbSODA outperforms them at identifying malware capabilities. In Table 3.4, we provide more comparative results to demonstrate symbSODA's better coverage in detecting malware capabilities and presenting them in the MITRE ATT&CK framework as compared to the existing tools.

### 3.5.4 Comparison with existing Sandboxes

Identification of malware behavior at the run-time is critical for symbSODA to select accurate deception ploys. Thus, we also compared symbSODA with existing

Table 3.4: Comparison of symbSODA with other related tools in terms of detecting different MITRE ATT&CK techniques within malware execution.

| MITRE Tactics | MITRE Techniques | Tools | | | |
|---|---|---|---|---|---|
| | | symbSODA | Kris et al. [66] | FORECAST | DodgeTron |
| Execution | Execution Through API | ✓ | ✓ | - | - |
| | Rundll32 | ✓ | ✓ | - | - |
| | Command-line Interface | ✓ | ✓ | - | - |
| | Service Execution | ✓ | ✓ | - | - |
| | Powershell | ✓ | ✓ | - | - |
| | WMI | - | ✓ | - | - |
| | Shared Module | ✓ | - | - | - |
| Persistance | Registry Run/Keys Start Folder | ✓ | ✓ | ✓ | - |
| | New Service | ✓ | ✓ | - | - |
| | Modify Existing Service | ✓ | ✓ | - | - |
| | Hooking | - | ✓ | - | - |
| | Scheduled Tasks | - | ✓ | - | - |
| | Image File Execution | - | ✓ | - | - |
| | Create of Modify System Process | ✓ | - | - | - |
| | Boot or Logon Autostart Execution | ✓ | - | - | - |
| Privilege Escalation | Process Injection | ✓ | ✓ | ✓ | - |
| | Access Token Manipulation | ✓ | ✓ | - | - |
| | Exploitation for Privilege Escalation | ✓ | ✓ | - | - |
| Defense Evasion | Obfuscated Files | ✓ | ✓ | - | - |
| | Software Packing | ✓ | ✓ | - | - |
| | Deobfuscated/Decode Files | ✓ | ✓ | - | - |
| | Masquerading | ✓ | ✓ | - | - |
| | DLL Side-Loading | - | ✓ | - | - |
| | Modify Registry | ✓ | - | - | - |
| | File Deletion | ✓ | - | - | - |
| | Virtualization/Sandbox Evasion | ✓ | - | - | - |
| | File Deletion | ✓ | - | - | - |
| Credential Access | Input Capture | ✓ | ✓ | - | - |
| | Credential Dumping | ✓ | ✓ | - | ✓ |
| | Data from local system | ✓ | - | - | ✓ |
| | Unsecured Credentials: Credentials in Files | ✓ | - | - | ✓ |
| | Credentials from Password Stores: Credentials from Web Browsers | ✓ | - | - | ✓ |
| | Unsecured Credentials: Credentials in Registry | ✓ | - | - | ✓ |
| Discovery | Query Registry | ✓ | ✓ | - | - |
| | Security Software Discover | ✓ | ✓ | - | - |
| | Process Discovery | ✓ | ✓ | - | - |
| | System Information Discovery | ✓ | ✓ | ✓ | - |
| | System Network Configuration Discovery | ✓ | ✓ | - | - |
| | File and Directory Discovery | ✓ | - | - | - |
| | Application Window Discovery | ✓ | - | - | - |
| | System Service Discovery | ✓ | - | - | - |
| | System Owner/User Discovery | ✓ | - | - | - |
| | Software Discovery | ✓ | - | - | - |
| Lateral Movement | Remote File Copy | ✓ | ✓ | - | - |
| | Remote Desktop Connection | - | ✓ | - | - |
| | Replication Through Removeable Media | - | ✓ | - | - |
| Collection | Clipboard Data | ✓ | ✓ | - | - |
| | Screen Capture | ✓ | ✓ | ✓ | - |
| | Email Collection | ✓ | ✓ | - | - |
| | Audio Collection | ✓ | - | - | - |
| | Video Collection | ✓ | - | - | - |
| Exfiltration | Generic | ✓ | ✓ | ✓ | ✓ |
| C&C | Generic | ✓ | ✓ | ✓ | - |
| | File Transfer Protocols | ✓ | - | - | - |
| | Ingression Tool Transfer | ✓ | - | - | - |
| Impact | System Shutdown/Reboot | ✓ | - | - | - |
| | Service Stop | ✓ | - | - | - |
| | Data Encrypted for Impact | ✓ | - | - | - |

Table 3.5: Number of techniques (T) and procedures (P) discovered by symbSODA compared to Cuckoo sandbox, Any.run and SODA

| Family | Malware Family | Discovery | Cuckoo | Any.run | SODA | symbSODA |
|---|---|---|---|---|---|---|
| InfoStealer | Fareit | T | 8 | 7 | 8 | 8 |
| | | P | 39 | 126 | 149 | 149 |
| | LokiBot | T | 7 | 2 | 11 | 11 |
| | | P | 21 | 173 | 243 | 258 |
| | Pony | T | 8 | 4 | 17 | 17 |
| | | P | 231 | 191 | 582 | 597 |
| | Racoon | T | 7 | 8 | 16 | 16 |
| | | P | 45 | 23 | 51 | 53 |
| Ransomware | Ryuk | T | 6 | 3 | 6 | 6 |
| | | P | 27 | 32 | 102 | 107 |
| | GandCrab | T | 8 | 10 | 10 | 10 |
| | | P | 192 | 109 | 245 | 248 |
| RAT | Gh0st | T | 2 | 2 | 6 | 6 |
| | | P | 4 | 57 | 69 | 71 |
| | VanilaRat | T | 1 | 0 | 12 | 12 |
| | | P | 1 | 0 | 12 | 12 |
| | Quasar | T | 5 | 2 | 13 | 13 |
| | | P | 14 | 4 | 16 | 16 |

Sandboxes such as Cuckoo [68] and Any.run [69]. Both Cuckoo and Any.run presents the observed malware behaviors in the form of MITRE ATT&CK framework. In this experiment, we specifically focus on Techniques and Procedures. A **Technique** represents how an adversary accomplishes the tactic by performing an action and a **Procedure** indicates the specific details of how an adversary carries out a technique. In this experiment, we ran nine (9) distinct malware across Cuckoo, Any.run, SODA, and symbSODA and listed the observed malware behaviors in the form of Technique (T) and Procedure (P) in Table 3.5. Clearly, symbSODA discovers more techniques and procedures than Cuckoo, Any.run, and SODA.

### 3.5.5 MSG Classifier Evaluation

#### 3.5.5.1 **Ground-Truth (GT2)**

We created a ground truth that can be utilized to evaluate the MSG-to-MITRE mapping using a remote access Trojan (RAT). RAT is a type of malware that incorporates a backdoor for gaining administrative access over the infected system. RATs usually have two modules, one runs on the attackers-end (commonly referred to as `"server"` or `"command and control server"` or `"C&C server"`) and the

other runs on the victim-end (client). The C&C server establishes remote communication on the victim's machine and sends commands to perform malicious tasks. Typically, each command is associated with a specific malicious behavior. We took advantage of this to create our second ground truth. According to the descriptions provided in GitHub, we obtained the source code for 13 different RATs from GitHub capable of performing 33 distinct malicious behaviors. We obtain the binary files by compiling the source codes. We run the client in the API Call Tracer and the C&C server on another VM connected to our API Call Tracer. We use the C&C server to perform each command one at a time and store the traces in a log file. This design implies that each log file corresponds to a particular malicious behavior. We manually extract MSGs from each log file and map them to malicious behaviors and MITRE techniques. From these traces, the MSG extractor retrieved 80 distinct MSGs. We manually filtered and mapped these 80 MSGs to 28 distinct malicious behaviors (as we obtained previously from the 13 RATs), which are mapped to 31 MITRE techniques. This ground truth is referred to as GT2 and will be used in further evaluations.

#### 3.5.5.2 **Evaluation results**

In our experiment, we classify eighty (80) MSGs mapped to thirty-one (31) MITRE techniques, where each MSG is mapped to one or more techniques. Out of the eighty (80) MSGs only twelve (12) are mapped to more than one techniques, nine (9) of them are mapped to two (2) techniques and three (3) are mapped to four (4) techniques. This multiple-techniques mapping occurs because some MSGs can achieve multiple behaviours that can be mapped to multiple techniques.

Table 3.6 shows MSG Classifier parameters we used for the experiment. We used three metrics to evaluate our tool: top-n accuracy and median and average ranking of the correct technique. Top-n accuracy is the accuracy of the predictions where we consider a prediction as true when the correct technique falls in the $n$ highest-ranked techniques.

Table 3.6: MSG Classifier Parameters

| | |
|---|---|
| Minimum word frequency | 4 |
| API TF-IDF enriching threshold | 20% |
| Word2Vec similarity threshold | 70% |
| Maximum number of words per MITRE technique | 40 |

Table 3.7: Top-n Accuracy of MSG Classifier

| n | 1 | 2 | 3 | 4 | 5 | 13 | 16 |
|---|---|---|---|---|---|---|---|
| Top-n accuracy | 63.75% | 81.25% | 82.5% | 86.3% | 90.0% | 96.2% | 98.7% |

Table 3.7 shows the top-n accuracy for $n$ being 1 to 5, 13 and 16. MSG Classifier was able to rank the correct technique as the first out of 31 techniques with an accuracy of 63.75%. This accuracy jumped to 81.25% for top-2 and 90.0% for top-5. After that, accuracy reached 96.2% for top-13 and 98.7% for top-16. MSG Classifier also achieved a median ranking of 1 and an average ranking of 2.68 of the correct technique.

#### 3.5.5.3 **Analysis of MSG Classifier's results**

We first investigated outputs where the top-1 predicted technique was incorrect. In many cases, the mapped technique was similar to the correct technique (GT2). Although most MSGs in our ground truth are mapped to only one technique, these MSGs can also be used to achieve other techniques. To study this, we analyze the results of incorrectly mapped MSGs by investigating if the corresponding MSG can achieve the highest two ranked techniques. For the highest-ranked technique, out of the 29 incorrectly mapped MSG, we found that 20 of them can be used to achieve a different technique than the one it mapped to (In the GT2). This implies that our mapping (by our tool) was correct. For example, an MSG that contains the APIs: "ControlService" and "CreateService" are mapped to the "Stop Service" technique in our ground truth, which is a technique used by adversaries to stop or disable services. However, MSG Classifier mapped it to the "System Services" technique, a technique used by adversaries to abuse system services to execute commands. The MSG can be used to achieve these two techniques, but our ground truth only maps it to the "Stop

Table 3.8: Top-n accuracies after analysis as well as excluding StackOverflow and enriching component.

| Experiment | Original | Excluding Stackoverflow | Excluding Enriching | After Result Analysis |
|---|---|---|---|---|
| Top-1 Accuracy | 63.75% | 48.75% | 58.75% | 88.75% |
| Top-2 Accuracy | 81.25% | 62.5% | 73.75% | 96.25% |

service" technique because the RAT that we used to extract the MSG behavior from used the MSG to stop service and not to use services to execute commands. When updating the ground truth based on our results analysis, the actual top-1 accuracy of the MSG Classifier increased to 88.75%, and the top-2 accuracy increased to 96.25% as shown in Table 3.8.

We then investigated cases where the algorithm fails to predict the correct technique in top-2 techniques and discovered two reasons: 1) The semantic gap between MSG and the correct MITRE technique. MSG Classifier could not bridge this gap as it requires a deeper understanding of the API behavior and MITRE techniques. For example, an MSG containing "IsDebuggerPresent," which checks if the calling process is being debugged, should be mapped to the MITRE technique "System Information Discovery," a technique an attacker uses to gather information from the OS. This mapping represents a high semantic gap as the algorithm needs to understand that checking if a process is being debugged means the adversary is gathering information about the system. 2) MSGs that consist of general-purpose APIs like CreateFile or CreateProcess that can be used by many techniques, making it difficult to identify the correct technique without checking the API's arguments. This excluding of arguments is a limitation of the tool that we leave for future work.

#### 3.5.5.4 MSG Classifier components importance

We studied the importance of adding Stack Overflow data and the enriching component of the MSG Classifier by running two experiments. In the first experiment, we run MSG Classifier using only MSDN for API text description and excluding Stack Overflow question-answer pairs. In the second experiment, we run MSG Classifier

without the enriching step. Table 3.8 shows the top-1 and top-2 accuracies for the original MSG Classifier and the two experiments. When we excluded Stack Overflow, top-1 accuracy dropped to 48.75% with a decrement of 23.53%, and top-2 accuracy dropped to 62.5% with a decrement of 23.07%. When excluding the enrichment component, top-1 accuracy dropped to 58.75% with a decrement of 7.84%, where the top-2 accuracy dropped to 73.75%, with a 9.23% decrements. These accuracies' decrements show the importance of Stack Overflow and the enriching component in MSG Classifier, while Stack Overlow contributed more to the results than the enriching component.

### 3.5.6 Performance Analysis of symbSODA

In this subsection, we evaluate the performance of symbSODA. We first evaluate the time required by symbSODA to orchestrate the deception actions, which can be divided into 1) Deployment time and 2) Overhead (Malware response time). We also evaluated symbSODA with multiple OECs and a single OES to demonstrate the scalability of our approach.

#### 3.5.6.1 **Deployment time**

We allow users to create new profiles or select pre-built ones. Users essentially choose which deception ploys to deploy by selecting/creating profiles. This deployment consists of the following three aspects: 1) generating the configuration file, 2) preparing the necessary HF, and 3) forwarding the End-Point DLL and the configuration file to the OEC.

**Experiment setup and result** In our evaluation, a user creates her profile based on a total of 50 ploys that symbSODA provides. Notably, the default deception strategy is NativeExecute, in which symbSODA allows the malware to run to discover the malware's actions in run-time. For NativeExecute, HF preparation is not required as symbSODA does not modify the response; instead, it only monitors the

Table 3.9: Malware deception overhead ($T_1$ = time without deception, $T_2$ = time with symbSODA deception, O = Overhead).

| Malware Type | Focused Malicious Behavior | Strategy | Deception Goal | Deception Action | Expectation (Observance to consider deception ploy worked) | T1 (sec) | T2 (sec) | O (%) |
|---|---|---|---|---|---|---|---|---|
| RAT | Remote command Execution | FakeExecute | Depletion | Execute the remote command in HF and show it to the malware | Command executed in HF and is shown to the attacker (C&C server is in our control) | 13 | 15 | 15% |
| InfoStealer | Steal credentials from the browsers | FakeExecute | Depletion | Show honey credentials from the HF | Honey credentials is seen to be exfiltrated (using packet capture) | 38 | 43 | 13% |
| Ransomware | Encrypt files for Impact | FakeSuccess | Diversion | Pretend the encryption took place without performing it | This malware creates a ransom note after successful encryption (observe the note being created) | 126 | 144 | 14% |
| Spyware | Capture screen | FakeExecute | Discovery | Capture screen from the HF and send it to the attacker | Captured screen of the HF is uploaded to our FTP server (redirected using ApateDNS) | 61 | 65 | 7% |

API calls for discovery. Therefore, the deployment time will be minimal. We repeat the experiment five times, changing the deception ploys by increasing NativeExecute. Figure 3.14 depicts the various deployment timelines for various ploys. Deployment time decreases as expected as NativeExecute increases and HF configuration time is also reduced. The maximum deployment time recorded is 72 sec. As the deployment occurred before any malware entered the system, the required deployment time is reasonable.

#### 3.5.6.2 Determine overhead/response delay time by comparing with the native execution

When a malicious process is confirmed by the detection agent, the OEC injects the End-Point DLL into the malware process. At first, we calculated the dynamic deception delay by running a malware sample in a machine without symbSODA and then with symbSODA. Finally, we calculated the time difference as system overhead due to the orchestration. The deployment time is not considered as we already evaluated it in the previous experiment. This experiment is performed on four types of malware (RAT, InfoStealer, Ransomware and Spyware). For each malware, we selected a malicious behavior and a relevant ploy to deceive it and recorded the malware execution time to complete the chosen behavior execution. We ran the experiment twice, once without symbSODA and then with symbSODA, recorded the execution time and calculated the overhead time. The experimental result is presented in Table 3.9. Our data shows that the maximum overhead time was 18 seconds (15% increment

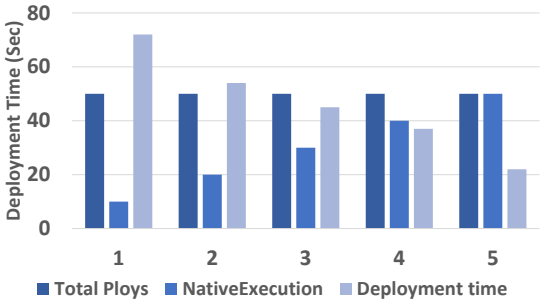



Figure 3.14: symbSODA deployment time with different ploys

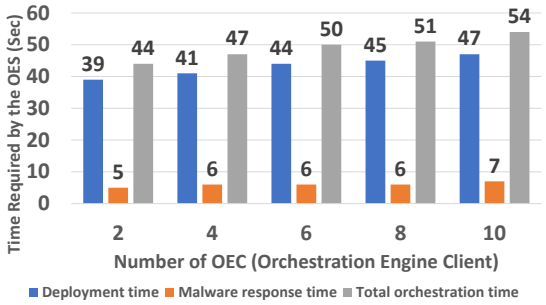



Figure 3.15: Avg time of a single OES to orchestrate and serve multiple OECs

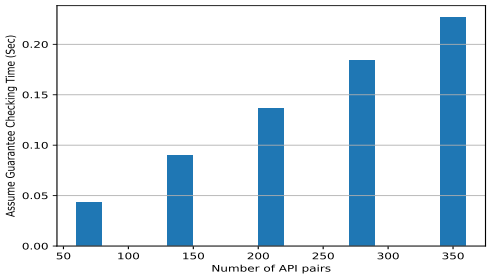



Figure 3.16: Assume Guarantee Verification Time

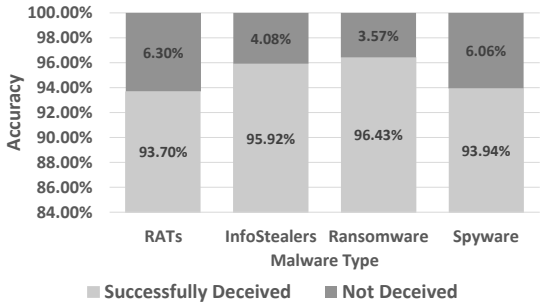



Figure 3.17: Accuracy of symbSODA across different malware types

compared to the normal malware execution) which is minimal/insignificant compared to the running/campaign period of malware.

#### 3.5.6.3 Measuring Scalability

To evaluate the scalability of our approach, we run multiple OECs that send service requests to the OES at the same time. The purpose of this experiment is to investigate how the OES performs when several OECs request services simultaneously.

**Experimental setup:** To maintain consistency, we used the same malware across all the OECs and created the same profile from each of them. Initially, we started the experiment using two OECs (clients) sending requests to the OES for profile selection/creation and recorded the deployment time. Then we increased the number of OECs by two and continued to monitor deployment time until the OECs count reached ten. Finally, we repeated the same experiment while simultaneously running the malware on different OECs to record malware response time and calculated the

overhead. Note that the execution time of the malware without symbSODA is 15 seconds.

**Actual result** Our obtained result is presented in Figure 3.15. As we can see, the overall orchestration time has increased as the number of OECs increases (the rightmost bars). From this experimental result, we can infer that even if the orchestration time increased, OES could still serve its service successfully with a minor/negligible overhead (maximum of 7s) compared to the malware's entire execution time (127s).

#### 3.5.6.4 Assume Guarantee Verification Time

Figure 3.16 shows the time to run the assume-guarantee verification. We implemented the verification algorithm that can handle any Boolean formula with an arbitrary number of parenthesis and variables. We can observe that for hundreds of API pairs, the running time is less than one second and almost linear with the number of pairings.

### 3.5.7 End-to-End Accuracy of symbSODA

In this section, we discuss the overall accuracy of symbSODA in terms of deceiving malware successfully. In this experiment, we first used the 42 open-source malware samples for GT1 (mentioned in Section 3.5.2) and 13 open-source RATs for GT2 (mentioned in Section 3.5.5) to create the deception ploys and prepare the End-Point DLL. Then we used four types of malware (RATs, InfoStealers, Ransomware and Spyware) for testing.

**Datasets and evaluation metrics** In this experiment, we used six (6) RATs, 122 InfoStealers, 96 Ransomware, and 31 spyware. We ensure that these malware samples are not used to create the deception ploys. Malware can be deceived at different MITRE tactics and techniques levels or just at a single point. For example, a malware collects some critical information about the system and exfiltrates it to the C2 server. We can deceive the malware at each phase separately or both phases. In

other words, we can use different ploys to deceive malware. To assess the accuracy of symbSODA, we verify how many used ploys were successful in deceiving the malware. The evaluation metrics: if the user selects N-number of deception ploys and if symbSODA uses M of them to deceive malware then, we calculate our accuracy to be (M/N)*100%.

**Observation criteria to consider deception ploy worked** For RATs, it's easy to observe the effectiveness of our detection ploys since we have the C&C by ourselves created from the source code. For each ploy, we observe the effect via the C&C server. For InfoStealers, we use Wireshark to examine the exfiltrated credentials to determine whether our deception ploys are working. For Ransomware, malicious activities are clearly visible (ransom note creation, file encryption). Typically Ransomware creates a ransom note after successfully encrypting the files on the host machine. The successful indication of our deception would be to fool the malware in creating the ransom note, even if the encryption did not take place. In the case of Spyware, it collects information about the victims and uploads it to the attacker. Using Wireshark, we identify the IP address where Spyware supposes to upload the file and used ApateDNS (proxy) to redirect the packet to our hosted FTP server.

#### 3.5.7.1 Experimental setup, expectations and results

To evaluate symbSODA with RATs, we used six (6) RATs with 37 distinct malicious behaviors. Based on different deception strategies and goals, we identified 116 valid deception ploys that can be deployed to deceive these RATs. We selected and deployed these 116 ploys and observed that symbSODA could deceive the RATs in 107 of them. In the case of InfoStealers, we observed eight (8) distinct malicious behaviors for which we identified 49 valid deception ploys. We selected and deployed these 49 ploys and observed that symbSODA could deceive the InfoStealers in 47 of them. For Ransomware, we observed eleven (11) distinct malicious behaviors for which we identified 28 valid deception ploys. We selected and deployed these 28 ploys and

observed that symbSODA could deceive the Ransomware in 27 of them. Finally, for Spyware, we observed thirteen (13) distinct malicious behaviors for which we identified 33 valid deception ploys. We selected and deployed these 33 ploys and observed that symbSODA could deceive the Spyware in 31 of them. Figure 3.17 presents the accuracy of symbSODA across different malware types. Overall, on average symbSODA achieved an accuracy of 95% (224 out of 237 ploys were successful in deceiving malware).

## 3.6 Related Work

Honeypots [6, 46], honeynets [45] and honeypatches [11] are frequently used to complement traditional detection and prevention mechanisms. Such techniques entice attackers with attractive false information (i.e., baits) to deflect them from real targets. Advanced honeypot and honeynet strategies, such as shadow honeypot [7], conceal information from the attacker by producing an instrumented shadow replica of the original system with fake information. Anomaly detection sensors redirect malicious network traffic to the shadow copy, while legitimate traffic is routed to the actual system. As a result, the precision of such a mechanism is dependent on the accuracy of the anomaly detection sensor in detecting the malicious payload, which is not always accurate [47]. In [22], the authors suggested instrumenting production systems with fake services and mock vulnerabilities to entice attackers. However, the fundamental hurdle for these approaches is the lack of randomness. In [1], the authors provided alternative solutions for creating indistinguishable honeypots from the real system. Unfortunately, these approaches are only theoretical and not directly applicable to real-world situations. Furthermore, skilled attackers may utilize techniques such as [46] to identify and evade these approaches. In contrast, our method uses embedded deception through API hooking and is deployed in the real environment, allowing us to overcome the lack of randomness. Furthermore, the approaches mentioned above do not consider the malware's running context. As these approaches

presume the malware will perform a specific set of malicious actions, all possible deception ploys are deployed ahead of time. Our approach considers the malware's current execution and context, hence only activates the deception ploys required to deceive the task malware is currently performing.

In [9], authors proposed a framework to extract deception parameters - the environmental variables on which the attackers rely to achieve their malicious goal. These deception parameters can be altered or misrepresented to achieve cyber deception. However, the framework neither supports real-time deception nor automated orchestration. In both [5,10], the authors presented an autonomous deception system capable of creating deception schemes by identifying potential HoneyThing candidates and orchestrating a deceptive environment with these HoneyThings. However, the approach provides static deception orchestration using HoneyThings and can only deceive a few malicious behaviors specific to credential stealing. Furthermore, as the approach focuses on deceiving a few malicious behaviors, the API to behavior mapping was limited and created by human experts. In contrast, our method outperforms both mentioned approaches by providing configurable deception as service (in real-time) and automated dynamic orchestration. We also automated the API(MSG)-to-MITRE mapping in order to understand the ongoing context within malware execution to be able to select the appropriate deception ploys at run-time. We note that Scarecrow [70] covers one of the 4D goals (deflect, distort, deplete, and discover), while symbSODA is capable of all 4D goals.

An orthogonal line of works utilized decoy files or honey accounts to detect ransomware [23, 24], general malware [25, 26], or DDoS attacks [27]. In [28] and [29], researchers employed honeypots and honeytokens to detect and prevent web-based attacks. Such strategies are out of the scope of our work as they mainly focus on detection where we are interested in deception. Moreover, these techniques are designed to detect only a particular type of malware (e.g., ransomware and banking trojans).

Our approach is generic and applicable to any malware as long as it has the malicious behaviors for which we designed deception ploys. Our MSG extraction approach is similar to malware behavior graph proposed in [71] but we have a different purpose: MSG is used for deception while malware behavior graphs are used for malware detection. In [66] and [67], the authors attempted to detect malware capabilities and presented them in the form of MITRE ATT&CK framework. However, their mappings were created manually based on their domain knowledge, where our tool can perform automatic malware behavior to MITRE ATT&CK framework mapping.

The importance of keeping deception actions consistent lies in the fact that inconsistent deception can alert attackers to the presence of deception and thus reduce its effectiveness. Inconsistent or conflicting deception actions might also lead to a system crash and compromise the security of the system. A few notable contributions in this area are [53, 54, 72], in which the authors proposed methods for generating logically consistent resource-deception plans that involve using logical inference to identify inconsistencies and iteratively refining the plan to ensure consistency. The series of works also offered an example of applying the method to a hypothetical scenario involving military networks under attack and explained a useful framework for developing effective resource-deception plans. The proposed planners for deception take in a sequence of commands for the operating system and identify potential deceptions that are logically consistent according to the constraints. Since the deception actions of symbSODA are based on MSGs, it performs deception consistency checks at the API level and based on their parameter values rather than at the OS commands level. Therefore, the granularity of the inconsistency checks in symbSODA differs from the methods proposed in [53, 54, 72]. Moreover, API calls cover a broader space of the system's resources than commands.

## 3.7 Discussion and Conclusion

This chapter presents symbSODA, a dynamic cyber deception orchestration system capable of analyzing real-world malware, discovering attack techniques, constructing Deception Playbooks, and orchestrating the environment to deceive malware. symbSODA enhances the state-of-the-art by providing dynamic real-time deception and customization options for users to choose their own deception ploys. Our proposed method of MSG extraction, followed by MSG-to-MITRE mapping, showed a promising result in bridging the gap between malware traces and the MITRE ATT&CK framework. Our extracted MSGs and MSG-to-MITRE mapping can play a vital role in improving the existing tools.

We conducted rigorous evaluations to validate and confirm symbSODA's efficiency and scalability against 225 recent malware and observed accuracy of 95% in deceiving them. Additionally, our approach extracted MSGs with a 97% recall, and our MSG-to-MITRE mapping attained a top-1 accuracy of 88.75%.

We acknowledge a few technical challenges regarding our approach. First, the semantic gap between API description and MITRE ATT&CK description makes automated API-to-MITRE mapping challenging. Additionally, since symbSODA relies on existing malware detection approaches that are imperfect, it can occasionally impact the normal usage of benign processes if they are misclassified as malicious. However, existing detection systems have a reasonably low false-positive rate, which can alleviate the problem. Moreover, malware evasion is a significant and practical issue for symbSODA. Given that we used API hooking to implement deception, symbSODA will be unable to deceive it if any malware can detect and evade API hooking. However, the symbSODA framework can leverage existing techniques that are resistant to malware evasion. For example, the API Call Tracer can be built on top of Barebox [73], which traces system calls via kernel-level hooking. We leave such improvements for future work. It is important to point out that symbSODA

can deceive polymorphic malware for which known samples are available. Although polymorphism changes malware's instructions, it may not change its functionality and the library APIs that its functionality depends on. Therefore, the underlying malicious subgraphs (MSGs) of the malware may remain similar or even the same. Thus, symbSODA would be expected to work against polymorphic malware. However, it's worth noting that the run-time deception ploy component of symbSODA may not successfully deceive malware that contains MSGs that have never been extracted before. However, considering that symbSODA can expand its knowledge base of malicious subgraphs (MSGs) by continuously analyzing more malware samples to learn new MSGs, symbSODA can include sufficient MSGs that enable effective deception ploys against novel malware.

# CHAPTER 4: ranDecepter: Empowering Defense Against Ransomware Attacks through Active Cyber Deception and Binary Reset

## 4.1 Introduction

Recently, there has been a significant increase in cyber-attacks globally, with ransomware being one of the prevalent threats. The number of ransomware variants has been rapidly rising each year, as reported by the Symantec Security report, showing a 46% increase in 2017 [74]. Money-driven ransomware doesn't discriminate and targets various sectors, including police departments and the healthcare industry. Rather than focusing on a specific set of computers, ransomware typically aims to infect a large number of victims. A notable example is the widespread attack of the Wannacry ransomware, which exploited the EthernalBlue vulnerability and affected thousands of victims [75].

Ransomware, a form of malware, shares common characteristics with other malware but also possesses distinct properties of its own. It employs similar strategies as other malware to avoid detection, spread, and target users. By injecting processes into target programs, extracting valuable user data, and establishing secure communication channels with Command & Control (C&C) servers, ransomware operates with sophistication. However, unlike traditional malware, its primary objective is to encrypt all private files on the victim's system and demand a ransom for their recovery, rather than operating stealthily in the background. These modern ransomware families, commonly referred to as crypto ransomware, employ encryption to restrict users' access to their computers or lock their screens until the ransom is paid. The relative simplicity of coding and customization, coupled with its lucrative nature, has led to an exponential proliferation of ransomware variants.

The field of ransomware detection is experiencing significant activity, and numerous studies have been conducted in this domain. However, the alarming surge in ransomware attacks indicates that current detection methods can still be evaded. Many of the techniques developed for ransomware detection involve static analysis combined with machine learning or signature-driven approaches [76, 77]. Although these methods boast high precision in detecting ransomware, modern advanced malware employs sophisticated obfuscation techniques that easily evade such approaches. To address these limitations, researchers have introduced dynamic analysis approaches for detecting ransomware [78–80]. However, a drawback of these approaches is that they rely on observing system behavior to make decisions, which means that some sensitive files may already be encrypted by the time the system takes action. But since ransomware attacks deal with sensitive data and the encryption process is irreversible unless a ransom is paid, meaning those initial encrypted file may not be recovered. In order to mitigate this challenge, recent studies propose the use of isolated sandboxes to subject new binaries to dynamic analysis before their execution on production systems [81–83]. However, it is important to acknowledge that the feasibility of this approach may vary in real-world scenarios.

Hence, there is a need to develop a proactive and dynamic solution for detecting ransomware in real systems without compromising sensitive files. The use of deception as a means to detect ransomware has been explored previously, focusing on effectively halting its operations. Deception-based approaches have garnered significant attention due to their ability to proactively identify new and emerging attacks [24, 84–88]. These methods involve the distribution of simulated or “honey" files throughout the file system. Any process attempting to access these files is flagged as anomalous. This approach proves advantageous for ransomware detection as it minimizes data loss, given that encrypting a honey file does not result in any actual loss of data due to their fictitious nature. Moreover, these approaches leverage the ransomware’s file

search methodology. Ransomware typically scans and encrypts files by traversing the file system and performing read/write operations. By exploiting this behavior, decoy files are strategically placed for the ransomware to encounter and act upon, providing confirmation to users of a ransomware attack. However, there are certain limitations to this approach. Firstly, it can be costly as honey files need to be implanted at various locations within the system, and they must be designed to always appear first within a sorted directory. Secondly, ransomware can easily bypass this solution by specifically targeting certain files, proving this deception strategy ineffective.

To overcome this limitation, we propose an alternative approach, ranDecepter that focuses on deceiving ransomware at the API level rather than the file-system level. This approach capitalizes on the fact that despite the ransomware's operation on multiple targeted files, the sequence of operations (e.g., searching, listing, reading, encrypting, deleting/overwriting) remains consistent. Therefore, by intercepting and monitoring these APIs, we can make informed decisions regarding ransomware activities. In this scenario, there is no need to track individual files or distribute decoy files across various locations. Consequently, the evasion techniques employed by ransomware in the previous approach become obsolete. Even if the ransomware attempts to evade a decoy-based detector by targeting specific files, it still must execute the same API calls to carry out malicious actions on those files. Hence, ranDecepter can identify the malicious behavior and detect the ransomware-related API sequences.

Addressing the information and resource asymmetry between attackers and defenders is a key challenge in cybersecurity. Attackers operate stealthily, conducting prolonged reconnaissance, while defenders must respond swiftly to emerging threats. Defenders are tasked with safeguarding vast infrastructures, while attackers can inflict significant damage with a single breach. In this study, we demonstrate how malware can be leveraged to retaliate against attackers and deplete their resources by inundating them with misleading or irrelevant information. Malware serves as a channel

between defenders and attackers, with the latter utilizing it to sabotage defenders' resources and sensitive data. However, defenders can also employ techniques to feed misinformation back to attackers through malware, thus depleting their resources. ranDecepter employs an automated method to identify the crucial addresses within a ransomware binary. This effectively establishes a loop within the malware, causing it to repeatedly initiate from the beginning and transmit encryption information/notifications back to the attacker. Consequently, this depletes the attacker's resources as they are required to store each victim's information. Our experimental findings validate the practicality, precision, and scalability of ranDecepter. Our contributions can be summarized as follows:

- Our approach employs API-level deception to detect ransomware in its initial stages without compromising any sensitive files.
- Our approach is the first within our knowledge to incorporate an automated process to identify key addresses within ransomware binaries, establishing a looping mechanism that strategically depletes the attacker's resources by continuously confirming encryption and transmitting keys pertaining to the victim.
- Our approach undergoes evaluation with real-world malware and benign applications, showcasing 100% accuracy in ransomware detection without any false positives, accompanied by minimal response time increment. Additionally, in terms of binary orchestration, our approach effectively depletes attackers' resources, achieving 100% accuracy and substantial resource depletion.

The rest of this work is organized as follows: Firstly, in Section 4.2, we provide background information about the topic and define key terms. Then, in Section 4.3, we present the threat model and the assumptions we have made. Section 4.4 explains the framework in detail, while Section 4.5 showcases the evaluations conducted to determine accuracy and performance analysis. Section 4.6 discusses related work

in the field. Finally, in Section 4.7, we present the limitations, future work, and conclusions drawn from our study.

## 4.2 Background

### 4.2.1 Ransomware and it's behaviors

It is a form of malware designed to encrypt files or lock down systems, rendering them inaccessible to users until a ransom is paid to the attacker. Understanding the major behaviors associated with ransomware is crucial for developing effective defense strategies. This section delves into the key characteristics and behaviors exhibited by ransomware.

- **Search and List Targeted Files for Encryption**: Ransomware exhibits the behavior of searching for and listing specific files on the victim's system for targeted encryption. The malware typically employs various techniques, such as file extension filters, to identify files that hold significant value or are likely to contain sensitive information. By selectively targeting specific files, ransomware aims to maximize the impact of the attack and increase the likelihood of victims paying the ransom.

- **Encryption Key and Unique ID Generation:** Ransomware generates a strong encryption key and scans the victim's system to encrypt files using this key. Simultaneously, it establishes a unique victim ID by incorporating system variables, such as system information, timestamps, random number generation, cryptographic hash functions, and network information. This ID associates encrypted files with the victim, ensuring that only the corresponding decryption key can unlock them. Attackers use the victim ID to identify the victim and associate the correct decryption key with their payment, facilitating file recovery upon successful ransom payment.

- **File System Encryption:** One of the primary behaviors of ransomware is the

encryption of files stored on the victim's system. Ransomware targets a wide range of file types, including documents, images, videos, and databases. By employing robust encryption algorithms, the malware ensures that the victim's files become inaccessible without the decryption key possessed by the attacker. This behavior serves as the basis for extortion, as the victim is compelled to pay the ransom to regain access to their valuable data.

- **Communication with Command and Control (C&C) Servers:** Ransomware often establishes communication channels with Command and Control (C&C) servers operated by the attackers. This connection enables the malware to send status updates, receive decryption keys upon payment, and potentially exfiltrate stolen data from the victim's system. The communication with C&C servers allows the attacker to maintain control over the ransomware and facilitates the coordination of ransom payment and data retrieval processes.

- **Ransom Notes and Instructions:** Ransomware commonly generates and presents ransom notes to the victim after encrypting their files. These notes, often in the form of text files or desktop backgrounds, provide instructions on how to make the ransom payment and regain access to the encrypted data. They typically include details such as the ransom amount, the cryptocurrency address for payment, and a deadline to incentivize prompt action. The language used in these notes can vary, ranging from professional to threatening, aiming to instill a sense of urgency and fear in the victim.

### 4.2.2 Malicious Sub-graphs (MSGs)

To accomplish its malicious objectives, malware utilizes a series of WinAPI calls. By representing each WinAPI as a node and the data flow between them as edges, we can construct a graph representing the sequence of WinAPIs. These graphs, known as malicious subgraphs (MSGs), play a crucial role in identifying and understanding

the execution flow of malware from traces. Analyzing these execution flows enables the design of precise deception plans to counteract malware activities effectively.

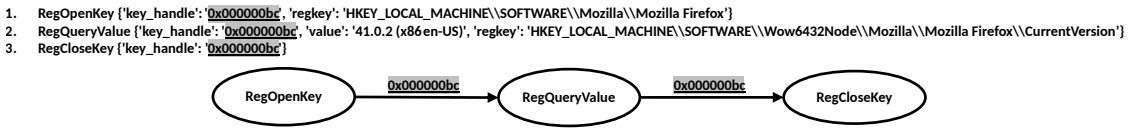



Figure 4.1: Malware execution trace to MSG conversion

**Malicious subgraphs extraction:** Recalling the definition of Malicious subgraphs (MSGs) from the background section, Malware achieves malicious objectives by executing a series of WinAPI calls, forming malicious subgraphs (MSGs) where each WinAPI represents a node and the data flow between them represents edges in a graph structure. As depicted in Figure 4.1, where lines 1-3 depict a concise segment of a ransomware execution trace that can be transformed into an MSG by considering the interdependencies (data-flow) among the nodes, exemplified by the key_handle attribute with a value of "0x000000bc".

### 4.2.3 Deception Strategies

In our implementation of API hooking-based deception, we employ four distinct approaches to respond to malware: FakeFailure, FakeSuccess, FakeExecute, and NativeExecute. These approaches, collectively termed "Deception Strategies", determine our chosen course of action when encountering malware. In this research, our primary focus has been on utilizing the FakeSuccess deception strategy.

- **FakeFailure** simulates a failed operation response to the malware without invoking the actual WinAPI call, deceiving the malware into thinking the operation was unsuccessful.

- **FakeSuccess** simulates a successful operation response to the malware without invoking the actual WinAPI call, providing static content for the operation.

- **FakeExecute** performs the malware's action on a remote machine and sends

back the response, tricking the attacker into receiving information from the remote machine.

- **NativeExecute** allows the malware to run to observe its current and future actions.

## 4.3 Threat Model and Assumptions

In the context of this research, we examine the behavior of ransomware, which may be combined with other malware variants, such as information stealers, to extract additional information from the victim system while simultaneously inflicting encryption-based impact. However, our primary focus is on the behavior of ransomware itself, which primarily involves operations related to file system encryption and the exfiltration of encrypted key(s) to the attacker for future use in decrypting infected files for ransom. In cases where combined malware is involved, we can incorporate deception techniques mentioned in previous works [10, 52, 89] to orchestrate a comprehensive deception strategy that addresses various malicious behaviors.

The Win32 API (Application Programming Interface) is a comprehensive set of functions and services provided by the Windows operating system. It offers developers access to diverse system resources, including file systems, networking capabilities, user interface elements, and hardware devices. It is important to note that this work assumes the ransomware relies on win32 API calls to carry out malicious actions, specifically related to searching and listing targeted files, reading, encrypting, and ultimately deleting/overwriting files.

Various research efforts have focused on detecting ransomware prior to its execution. These approaches involve static analysis of the binary without running the application or dynamic analysis within a controlled environment to uncover the true intent of such programs. However, our approach concentrates on providing a solution for detection once the malware has infiltrated the system and is already executing.

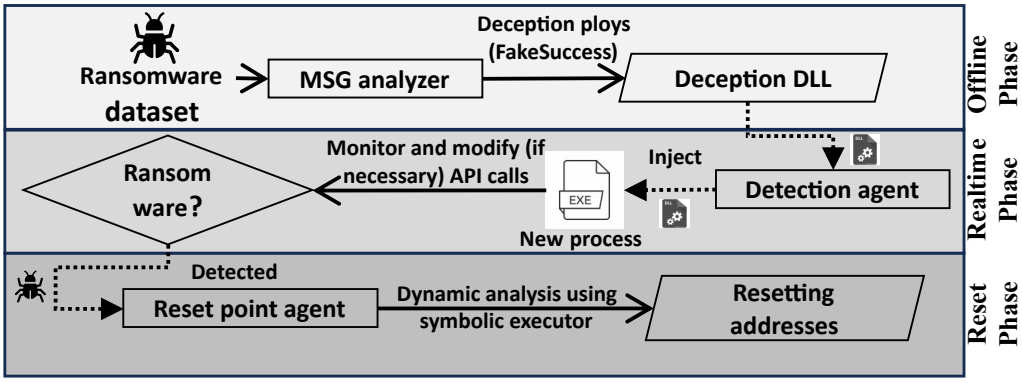



Figure 4.2: Overall system and data flow

In summary, the threat model for our approach involves: a) the infiltration of a ransomware into the system through channels such as email or social engineering, and b) the execution of the malware with the objective of encrypting specific files, utilizing Win32 APIs to carry out these malicious actions.

## 4.4 System overview

This section provides a comprehensive overview of our system and its workflow, as depicted in Figure 4.2. The system comprises three main phases: the offline phase, the real-time phase, and the reset phase.

In the ***offline phase***, we initiate the process by collecting ransomware samples and their corresponding antivirus (AV) labels from diverse sources such as VirusShare, MalShare, Hybrid-Analysis, and VirusTotal. Using Cuckoo Sandbox, we execute these malware samples to gather their runtime execution traces. These traces undergo *MSG analysis*, as detailed in [soda], resulting in the creation of a knowledge base that encompasses ransomware-related MSGs. To deceive and detect ransomware, we developed deception ploys in the form of “FakeSuccess" hooks at the API level. For example, if the malicious action of a particular malware involves encryption, we develop detour functions for encryption-related APIs that return success without actually performing the encryption. The hooks and detour functions for the relevant APIs, as listed in the ransomware MSG knowledge base, are implemented in C++ using the EasyHook library. The outcome is a *Deception DLL* that contains the

necessary APIs for monitoring and embedding deception ploys.

In the **real-time phase**, we create a *Detection Agent* using VBScript, which actively monitors and identifies newly initiated processes on the victim's machine. When a new process is detected, the Deception DLL is injected into it for further monitoring. The Deception DLL has the capability to monitor invoked APIs and their parameters, as well as manipulate input parameters and return values. In simpler terms, it can 1) determine if the current process invokes APIs in a sequence that matches the MSGs within the offline knowledge base, and 2) modify API calls to neutralize their actions, such as encryption. If ransomware is detected during this phase, the executable file associated with the ransomware is transferred to a deceptive environment for further analysis aimed at identifying critical addresses within the ransomware binary.

In the ***reset phase***, we employ dynamic analysis using a symbolic executor [89] to pinpoint critical addresses within the malware, enabling the creation of a looping mechanism. By establishing a loop within the malware, our approach triggers repeated execution of the malware, leading to the continuous transmission of encryption information, notifications, and secret keys back to the attacker. This strategy depletes the attacker's resources, as they must store information for each victim.

### 4.4.1 Offline Phase: Deception DLL Creation

This phase comprises two main steps: (a) Building a knowledge base of relevant malicious subgraphs associated with ransomware, and (b) Designing and implementing a deception DLL using the knowledge base, primarily employing the FakeSuccess deception strategy. Under this strategy, ranDecepter provides deceptive responses to the malware, simulating successful operations without invoking the actual WinAPIs. For instance, when the ransomware attempts to delete the original file after encryption by calling the DeleteFile API, our system intercepts the call and returns a "true" value without executing the actual DeleteFile operation.

To extract malicious subgraphs (MSGs) from malware, we adopt the approach

outlined in [90]. We customize the Cuckoo Sandbox to create a dynamic malware analysis environment, where we execute the malware and capture their corresponding execution traces. Subsequently, we apply an MSG extraction algorithm based on [90] to these traces, generating the MSG knowledge base. At the end of this stage, we extract all the MSGs as well as APIs corresponding to ransomware in a form of a knowledge base, which serves as the foundation for planning deception ploys.

**Deception ploys planning**. We have devised and implemented deception ploys using the “FakeSuccess" deception strategy, in which fake successful outcomes are returned for the operations executed by the malware. The primary objective is to create an illusion of successful execution to the malware and the attacker, making them believe that the attack has been carried out successfully. However, under the hood, our system refrains from actually performing these operations, thereby preserving the system's resources and preventing further infection.

Let us now delve into the meticulous planning of our deception ploys, as outlined in Table 4.1. It should be noted that the table presents only a subset of our deception ploys and their corresponding malicious subgraphs (MSGs) for the purpose of in-depth discussion and enhanced comprehension. Our system incorporates additional ploys and MSGs that are not listed here.

Taking the “File encryption" behavior as an example, Table 4.1 illustrates three MSGs associated with different malware implementations. In the first scenario, the malware derives a session key from a hardcoded password embedded within its code. Alternatively, the password could be obtained dynamically from the command-and-control (C&C) server at runtime. In the second and third scenarios, the malware directly generates the session key using the CryptGenKey API and subsequently employs it for file encryption.

Across all three cases, the malware follows a consistent pattern: it creates a new file, writes the encrypted content into this newly generated file, and later deletes the

Table 4.1: Deception ploy planning for ransomware (O=original file, D=Destination file, Pwd= password, S=session key, CO=content of the original file, P= public key). N.B. FileHeader writing is optional, hence noted in italic font.

| Malware Behavior | Malicious subgraph | Deception Actions |
|---|---|---|
| File encryption | **Case 1:** If session key is derived from password CreateFile(O)-CreateFile(D)-CryptAcquireContext-CryptCreateHash(Pwd)-CryptDeriveKey(S)-*WriteFile(D)[FileHeader]*-ReadFile(O)-CryptEncrypt(CO)-WriteFile(D)-CloseHandle(O)-CloseHandle(D)-CryptDestroyHash-CryptDestroyKey-CryptReleaseContext<br><br>**Case 2:** If session key is not derived from password CreateFile(O)-CreateFile(D)-CryptAcquireContext-CryptGenKey(S)-CryptExportKey-*WriteFile(D)[FileHeader]*-ReadFile(O)-CryptEncrypt(CO)-WriteFile(D)-CloseHandle(O)-CloseHandle(D)-CryptDestroyKey-CryptReleaseContext<br><br>**Case 3:** If public key is used in addition to session key for key exchange CreateFile(O)-CreateFile(D)-CryptAcquireContext-CryptGenKey(S)-CryptGetUserKey(P)-CryptExportKey-CryptDestroyKey(P)-*WriteFile(D)[FileHeader]*-ReadFile(O)-CryptEncrypt(O)-WriteFile(D)-CloseHandle(O)-CloseHandle(D)-CryptDestroyKey-CryptReleaseContext | In all three scenarios, the process involves the creation of a new destination file where the encrypted data is written. The specific actions undertaken are as follows:<br>1. Upon the invocation of CreateFile(D), a new destination file is generated.<br>2. When CryptEncrypt(CO) is called, our detour function intercepts the execution and returns a “True" value without actually executing CryptEncrypt, ensuring that the original file's content remains unencrypted.<br>3. WriteFile(D) is used to write the encrypted content into the destination file. However, we return a success status (True) without actually performing the write operation.<br>4. The completion of the operation on the destination file is indicated by CloseHandle(D). Consequently, we delete the destination file, retaining only the original files within the system. |
| File deletion | DeleteFile(O) | Ransomware calls DeleteFile(O) to delete the target file. Upon invocation of this API we return true without deleting the file. |

original file. To deceive this behavior using the FakeSuccess strategy, our objective is to make the malware believe that it has successfully executed all these operations, including the creation of the new file, encryption of the original file's content, writing the content into the new file, and ultimately deleting the original file. However, our ultimate aim is to retain the integrity of the original file, ensuring it remains undeleted and conserving system resources by avoiding the creation of unnecessary files. To achieve this, we employ API hooking on the CryptEncrypt API. Upon invocation, we intercept the call and return a “True" value without actually executing CryptEncrypt. Consequently, the content of the original file remains unencrypted. In theory, the plaintext data from the original file is expected to be written to the new file. However, to prevent the creation of these superfluous files and conserve system disk space, we also ensure that the WriteFile operation on the newly created destination file returns a success status (“True") without actually performing the write operation.

Now, one may question why we did not apply the same strategy to the CreateFile operation on the destination file. The reason lies in the intricacy of the CreateFile operation, as it returns a file handle that is subsequently utilized in many subsequent

operations. To achieve FakeSuccess for all these operations would be cumbersome and, in some cases, unpredictable from the outset. Therefore, we adopt a simple workaround: allowing the malware to create these files and, when the malware perceives that the operations on these files are complete, it invokes the CloseHandle API with the file handle generated during the CreateFile API call. During this invocation, our detour function calls the DeleteFile API to eliminate the destination file, ensuring that no extraneous files are created. This approach effectively simulates the effect of performing FakeSuccess on the CreateFile API.

At the completion of this stage, we compile an extensive table akin to Table 4.1, which serves as a comprehensive guide outlining the APIs that require hooking, the appropriate response strategies, and their potential sequencing. In essence, this stage provides us with the blueprint for implementation and the requisite features to effectively deceive and detect ransomware.

**Deception ploys implementation in the form of Deception DLL**. In this phase, we proceed to programmatically implement the actions described in the previous step. The deception actions are implemented using API hooking. Our approach involves creating a DLL file, referred to as Deception DLL, which is injected into malware using the DLL injection method. API hooking is a widely recognized technique for intercepting API calls made by targeted executables and enabling the monitoring and modification of API responses. To accomplish this, we employ the EasyHook library [57], a free and open-source hooking library designed for both 32-bit and 64-bit Windows processes. We implement a detouring function for each original WinAPI call we want to intercept. As a result, the detouring functions take control of the malware and provide us with the option to enforce our defined deception actions to modify the execution.

The outcome of this offline phase is a DLL file encompassing all the necessary deception actions required to deceive and detect ransomware (shown in Figure 4.3

in yellow). This Deception DLL is then prepared for injection into the monitored processes during the realtime phase.

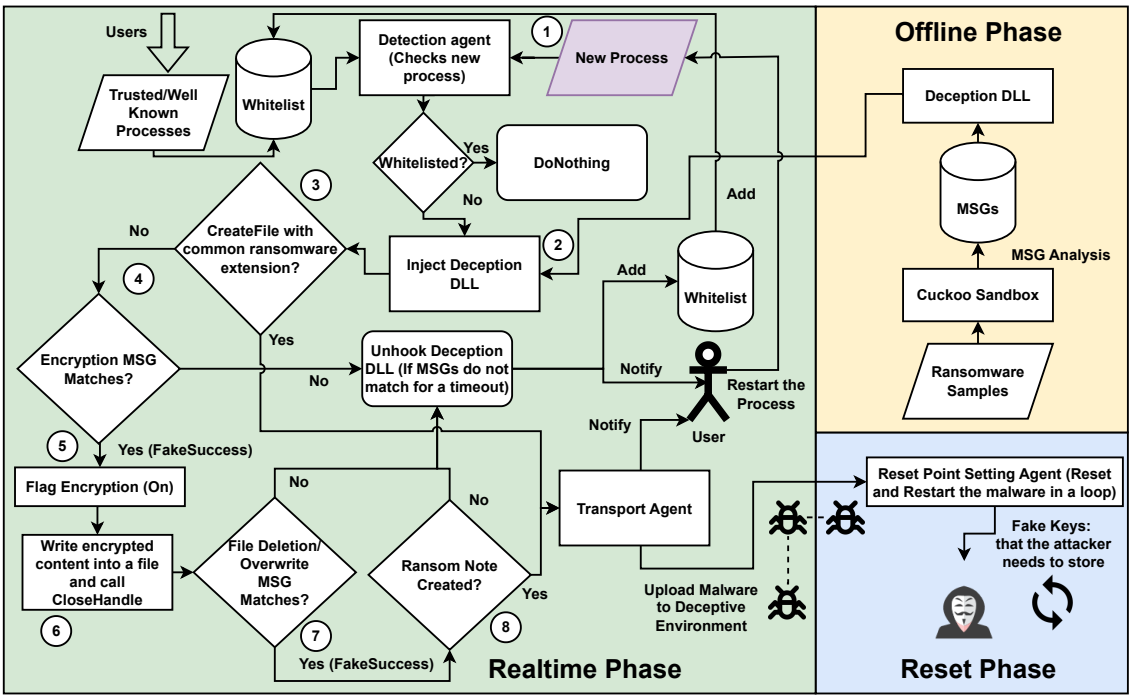



Figure 4.3: Detailed illustration showcasing the various phases and components of our system, depicting their data flow and decision-making processes.

### 4.4.2 Realtime Phase: Ransomware Detection using Embedded Deception (API) Hooks

In this phase, we inject the Deception DLL created in the previous phase into various processes to observe their runtime behaviors and determine whether they exhibit ransomware-related behaviors. We will discuss this section with the help of Figure 4.3 (specifically, the part marked in green), which provides the details of different phases and components of our system as well as their data flow and decision-making processes.

The realtime phase starts with a detection agent which looks out for the creation of new process. Upon new process creation, it suspends the process at it's entry point and checks the process against a whitelist, created by the user of the system as well

as experts that lists well-known benign processes. If the process is already whitelisted then the detection agent does nothing. Otherwise, the process is not yet whitelisted, and the detection agent injects the Deception DLL into this new process (marked as 1 and 2 in Figure 4.3). In the following paragraphs, we describe which APIs are hooked and the rationale behind it and the steps taken on the invocation:

**CreateFile, MoveFile, MoveFileWithProgress and Sleep:** One of the critical behaviors exhibited by ransomware involves file creation, encryption, and deletion. Therefore, the CreateFile API call plays a vital role in providing insights into ransomware activities. Specifically, we focus on the "dwCreationDisposition" parameter, which includes values such as "CREATE_ALWAYS" and "CREATE_NEW," indicating the creation of a new file. Furthermore, we examine the "lpFileName" parameter to determine the file name. To identify ransomware attacks, we consult an extensive list of known ransomware file extensions provided by security analysts [91]. If we detect any of these extensions during the CreateFile call, it serves as a clear indication of a ransomware attack as shown in Figure 4.3 (marked as 3). In such cases, our system immediately terminates the process and informs the Transport Agent for proceeding to the next phase, known as the reset phase (marked blue in Figure 4.3).

However, some ransomware variants append these known extensions after completing the encryption process, using the MoveFile/MoveFileWithProgress API call for file renaming. Therefore, we also perform the ransomware extension check during the MoveFile/MoveFileWithProgress API call (marked as 7 in Figure 4.3).

Now a question can be raised regarding the potential presence of benign applications frequently utilizing the CreateFile API. To address this concern, we employ a tracking mechanism for a specific duration. Through extensive experimentation, we have determined that the delay introduced by our hooking functions is negligible, as detailed in Section 4.5.3. Moreover, our system offers the flexibility of incorporating a configurable timer, which is managed through a global variable. Nevertheless, based

on our comprehensive evaluations, we have determined that the introduction of hooks in our system incurs minimal delay. Consequently, the use of a timer for operating the hooks is optional and can be configured according to the user's preferences and specific system requirements. To facilitate this tracking process, our deception DLL maintains a comprehensive record of file names and their associated handles in a designated text file, referred to as "files_in_action". When the CloseHandle function is invoked with the corresponding handle, the corresponding entry is promptly removed from the text file.

Considering that malware often employs the Sleep API to introduce delays for evasion purposes, we also hook the Sleep API and include its duration in the timer variable. This ensures that malware cannot evade our approach by utilizing sleep operations.

**WriteFile, CloseHandle, CryptAcquireContext and CryptEncrypt:** Given that CloseHandle and WriteFile are frequently invoked by benign processes, the detour functions for these APIs will also be called numerous times. In these cases, the detour functions simply execute the original CloseHandle and WriteFile APIs without any modifications, until evidence of encryption-related operations is discovered.

Referring to the MSGs listed in Table 4.1, we identify CryptAcquireContext as the entry API call for encryption operations. When the process calls CryptAcquireContext, we set the Encryption flag (marked as 5 in Figure 4.3). As long as the Encryption flag is not set, the detour functions for CloseHandle and WriteFile intercept the calls and execute the original APIs.

Regarding the CryptEncrypt API, a nonzero (TRUE) return value indicates a successful encryption operation. Notably, the **pbData* parameter holds significance, as it refers to a buffer containing the plaintext to be encrypted. In our detour function for CryptEncrypt, we return True without performing the encryption operation, leaving the *pbData* buffer unchanged, containing the plaintext rather than the ciphertext. We

store the address of this buffer and check if any subsequent WriteFile calls utilize it as the *lpBuffer* parameter, which represents the buffer containing the data to be written to the file. Consequently, when the Encryption flag is set and the *lpBuffer* matches the *pbData* within the WriteFile call, the deception DLL returns True, simulating a successful write operation without actually writing anything to the destination file.

Additionally, if the Encryption flag is active and CloseHandle is invoked with the stored handle in the "files_in_action" list, it signifies that the ransomware has completed operations on the corresponding newly created file. As a result, we remove this file to ensure no extraneous files remain in the system (marked as 6 in Figure 4.3).

**DeleteFile:** Once the ransomware finishes encrypting the files, it proceeds to delete the original user files or overwrite them with the encrypted content. Our system addresses the overwriting case through the previously described MoveFile/-MoveFileWithProgress API hooking. Regarding the DeleteFile API call, our system intercepts the call and returns a True value without executing the actual DeleteFile operation. This approach preserves the original files while giving the malware the impression that the deletion was successful. This specific case is denoted as 7 in Figure 4.3.

If the process does not attempt to delete or overwrite files, even after the encryption process within the given timeout period (if given), we mark the process as clean and add it to the whitelist. The rationale behind this decision is that users sometimes use legitimate software to encrypt files before transferring or sharing them. In such cases, there are two possibilities: either the original file is deleted or overwritten, or the legitimate software creates a new file for the user's operations while leaving the original files untouched. If the original files remain unaffected, we consider it safe to mark the process as clean.

However, in scenarios involving deletion or overwriting, as mentioned earlier, we employ the deception technique to keep the original files intact. One might ques-

tion the impact on legitimate applications that perform encryption, file deletion, or overwriting, as our deception strategy inhibits their intended operations. The main reason behind this approach is that ransomware exhibits behaviors that closely resemble those of benign applications that perform similar operations. As a result, many systems mistakenly categorize ransomware as benign. In such situations, we rely on the distinct characteristics of ransomware, such as ransom notes or changes to the desktop background displaying ransom payment details.

We continue applying the FakeSuccess deception strategy to the application under investigation and actively search for ransom notes or changes in the desktop background at well-known locations within the system for a predefined timer. If no such indications are found, the system considered it to be a benign application, hence added it to the whitelist and unhooked the Deception DLL; at the same time, the system notifies the user that it has blocked the application's operation due to its similarity to ransomware behavior (as it performed encryption and deletion). We then request the user to restart the process (application) with their desired operation again. Consequently, our system will not interfere with this process in the future, as it is marked as a whitelisted application. However, we acknowledge that our interference with the benign application in this scenario may cause a delay of up to five minutes and require the application to be rerun. We believe that this temporary inconvenience is a small price to pay for the overall safety and protection of the user's system, considering the potential disastrous consequences that can arise from a ransomware attack.

**Ransom note creation:** In order to identify ransomware notes, we leverage the CreateFile API, specifically focusing on key locations such as Desktop, Documents, Downloads, Music, Pictures, and Videos. When a new file is created, we perform a keyword search within these locations, looking for terms such as "ransom," "encrypt(ed)," "pay(ment)," "bitcoin," "lose," "decrypt(ed)," and "delete(d)." If any of these keywords are found, we notify the user to review the file(s) and confirm whether

their system is under a ransomware attack. Additionally, to detect ransom notes through background changes, we utilize image-to-text conversion using the Google Cloud Vision API [92]. We have also implemented effective strategies from [76] to enhance the detection of ransom notes. Furthermore, we have incorporated effective strategies from [76] to enhance the accuracy of ransom note detection. In our future work, we plan to implement an approach similar to [93], which integrates Latent Semantic Analysis (LSA) - an NLP-based analysis to assess the similarities between file contents and known ransom notes, further improving the effectiveness of ransom note detection.

To summarize, during the realtime phase, we employ deception techniques to detect malicious processes, specifically ransomware. Once a malicious process is identified, we proceed to transfer the corresponding executable file associated with the process to a carefully controlled deceptive environment. This transfer is facilitated by a transport agent, ensuring the secure relocation of the file for further analysis.

### 4.4.3 Reset Phase: Exhausting attackers' resources by repeatedly initiating malware through binary orchestration

In this phase, we conduct dynamic analysis on the malware to identify critical addresses within its code. This analysis enables the creation of a looping mechanism within the malware, resulting in the repetitive execution of its operations. Through this loop, the malware continuously transmits encryption keys to the attacker, which will be utilized for decrypting the files upon ransom payment. As a result, the attacker must store these keys for future decryption operations.

To accomplish the task of identifying these crucial addresses, we leverage gExtractor, a dynamic malware analysis tool built upon a selective symbolic execution engine. It provides comprehensive reporting of user-level APIs invoked by the malware, along with associated parameters for 390 Windows APIs. It also reports the entry point of the binary using objdump [94], which serves as the starting point for program exe-

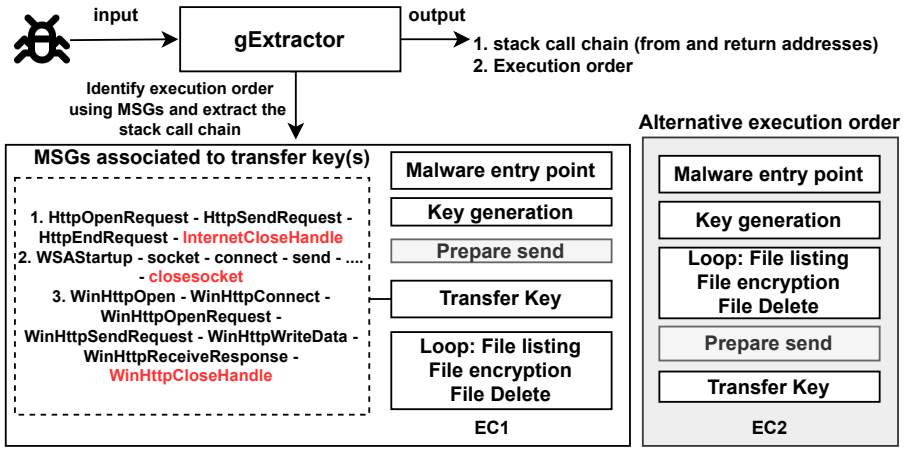



Figure 4.4: This figure shows the identified execution chains (EC1 and EC2). On the left side, we presented the MSGs ransomware use to perform "Transfer Key(s)". Red colored APIs indicate the end of their respective MSG.

cution. Furthermore, gExtractor captures the active call stack chain of an API call, encompassing the caller's (virtual) address and the first address block of the called API. As the chain grows with subsequent function calls, gExtractor removes entries as the execution returns to the caller.

Within the context of binary orchestration, a crucial step involves understanding the sequence of operations performed by the malware. This understanding is essential for identifying the points at which the malware generates encryption keys. By analyzing the execution traces provided by gExtractor and utilizing the knowledge acquired during the offline phase regarding the malware's operation-specific malicious subgraphs (MSGs), we can automatically identify the execution chain, which represents the order of these operations. Through experiments conducted on seven ransomware samples from different families (CryptoLocker, WannaCry, Ryuk, GandCrab), we have identified two major distinct execution chains, visually presented in Figure 4.4 as EC1 and EC2. In addition, there are a few other variations, where the loop operation depicted in Figure 4.4 is divided into two parts: the first for target file listing and the second for encryption and original file deletion.

Next, we strategically insert JMP instructions among these operations to create a

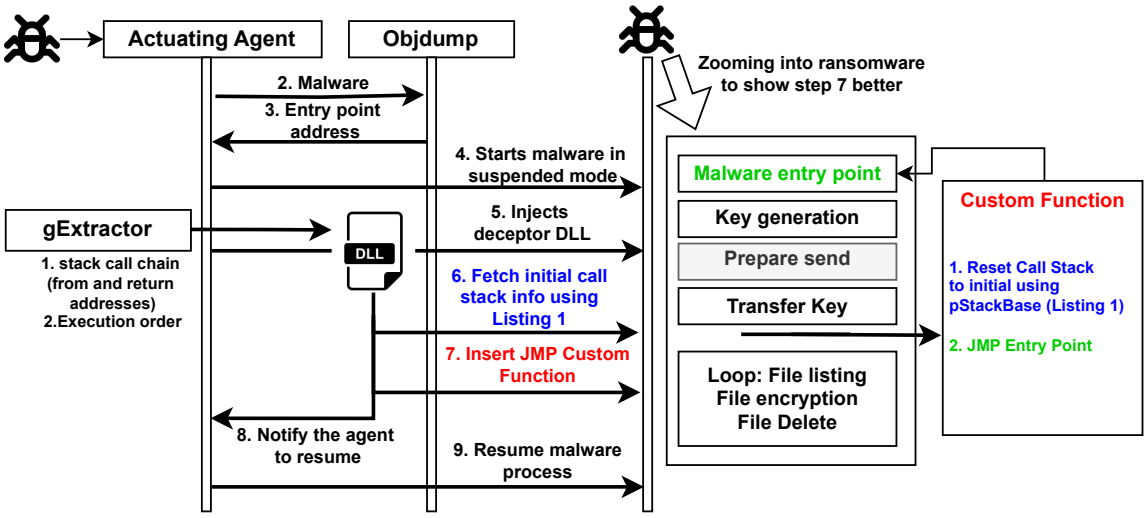



Figure 4.5: Workflow of binary orchestration.

loop. This process involves several critical determinations. Firstly, we determine the address of the malware's entry point, which is provided by gExtractor. Additionally, we need to determine the initial state of the stack (i.e., stack base and limit) at the entry point. We extract such information by running the code shown in Listing 4.1.

The next crucial step involves determining the appropriate insertion point for the JMP instruction. This requires identifying where the key transfer operation concludes within the malware's execution. Utilizing gExtractor, which captures the call stack chain of each API call, we can precisely locate this insertion point. Specifically, we focus on the call stack chain of the last APIs marked in Figure 4.4 in red. This call stack chain reveals the invocation location of the API or set of APIs and the return address within the malware upon the completion of the API call. We insert the JMP instruction at the next address following the return to the malware address pointing to our customized function. The customized function's tasks include restoring the call stack to its initial state as described in the previous step, and JMPing to the malware's entry point. To streamline this process, we employ EasyHook for automation. By implementing these steps, the malware enters a continuous loop, repeatedly restarting and transferring keys to the attacker.

Figure 4.5 illustrates the sequential workflow of binary orchestration after the dy-

namic analysis stage conducted by gExtractor. Through dynamic analysis, we extract the stack call chains (including from and return addresses) of the relevant APIs and determine the malware's execution order. In the next step, we create a "Deceptor DLL," a DLL responsible for randomizing collected data, inserting JMP instructions, and resetting the call stack. Ransomware typically collects system information, timestamps, random numbers, cryptographic hash functions, and network information to generate a unique victim ID. The Deceptor DLL hooks these APIs and ensures that the collected values are randomized, resulting in a new victim ID each time. Once gExtractor completes the dynamic analysis and provides the stack call chains and execution order, the Actuating agent determines the appropriate locations for inserting JMP instructions. The Actuating agent then starts the malware in a suspended mode and injects the Deceptor DLL into the malware process (steps 4 and 5 in Figure 4.5). The Deceptor DLL identifies the initial call stack values by running the code provided in Listing 4.1.

Listing 4.1: Code to find stack values

```
void *pStackBase;
void *pStackLimit;

_asm {
mov eax, fs:[04h]
mov pStackBase, eax
mov eax, fs:[08h]
mov pStackLimit, eax
}
```

Then, the Deceptor DLL inserts the JMP instruction to the customized function, which resets the call stack to its initial state and jumps to the entry point, notifying the Actuating agent to resume the malware process. As a result, we create a loop inside the malware (depicted on the right side of Figure 4.5), ensuring its repetitive execution. This loop facilitates the collection of random values during the collection

stage and enables the continuous transmission of new unique IDs and keys to the attacker for storage.

## 4.5 Evaluations

In this section, we present the comprehensive evaluation of our proposed approach utilizing a diverse range of ransomware variants. Our evaluation experiments aim to validate the accuracy, effectiveness, and overhead of our method, shedding light on its performance and capabilities.

### 4.5.1 Dataset

To evaluate the robustness of our approach in real-time ransomware detection, we curated a dataset consisting of fifteen (15) samples from nine (9) distinct ransomware families. This dataset encompasses a wide spectrum of ransomware variants, enabling us to thoroughly assess the performance and effectiveness of our method. A summary of the dataset is provided in Table 4.2.

Furthermore, to accurately assess the accuracy and effectiveness of our binary orchestration approach, we conducted resource depletion assessments utilizing four publicly available malware samples, such as those found on GitHub [95–98]. These samples allow us to precisely evaluate the efficiency and effectiveness of our approach in terms of resource consumption and depletion at the attacker's side.

### 4.5.2 Evaluation of Accuracy and Effectiveness against Ransomware

In order to assess the accuracy and effectiveness of our approach in real-time ransomware detection, we conducted rigorous evaluations using a dataset comprising 15 real-world ransomware samples, as outlined in Table 4.2. Our system demonstrated exceptional performance, achieving a 100% detection rate for all the evaluated malware instances. The results of the evaluation are presented in Table 4.3.

Upon analyzing the results, we observed that some malware samples, such as m1, m2, and m3 from the WannaCry family, were detected at an early stage during the

Table 4.2: Ransomware dataset

| **Sample ID** | **Malware Family** | **MD5 hash** |
|---|---|---|
| m1 | WannaCry | 84c82835a5d21bbcf75a61706d8ab549 |
| m2 | WannaCry | 80d2cfccef17caa46226147c1b0648e6 |
| m3 | WannaCry | db349b97c37d22f5ea1d1841e3c89eb4 |
| m4 | Dharma | 3dcabb52d7b4c9d0ea8e0182732b39fd |
| m5 | Cerber | 8b6bc16fd137c09a08b02bbe1bb7d670 |
| m6 | Lockbit | 927426bafb84fe8daff84cff77258e0d |
| m7 | Lockbit | 33228a20a7e985f02e2ddd73cccde729 |
| m8 | Cyborg | f98d999b7bb31c89f9ec7094723a78ab |
| m9 | Cyborg | b8208e696f51195e59d1a8f7e7d7e4cd |
| m10 | Cyborg | 74bfab32741f15b9fcfb32aacffab584 |
| m11 | Saturn | bbd4c2d2c72648c8f871b36261be23fd |
| m12 | GandCrab | e8e19525aa73d1714f15552d166aaa84 |
| m13 | GandCrab | e6b43b1028b60000092533446 32e69c4 |
| m14 | TeslaCrypt | 6e080aa085293bb9fbdcc9015337d309 |
| m15 | Xorist | 4a71a07c9c742751044e2197ee8234a8 |

Table 4.3: Ransomware detecting stages and the required time comparison with and without our system.

| **Sample ID** | **Detection stage** | **Time(t) required to reach point "A" by the malware without our system** | **Time(t) required to reach point "A" by the malware with our system** |
|---|---|---|---|
| m1, m2, m3, m4 m12, 13 | CreateFile called with known ransomware extension | "A" here is the first CreateFile call with known ransomware extension. | |
| | | For m1, m2, m3: Avg(t) = 0.0013 ms For, m4 Avg(t) = 1.23 ms For, m12, m13 Avg (t) = 7.8 ms | For m1, m2, m3: Avg(t) = 0.00142 ms For, m4 Avg(t) = 1.37 ms For, m12, m13 Avg (t) = 8.18 ms |
| m5, m6, m7, m8 m9, m10, m11,m14, m15 | Encryption ✓ File deletion /overwrite ✓ Ransom note creation ✓ | "A" here is the first DeleteFile/MoveFile/MoveFileWithProgress called to delete/overwrite file. | |
| | | For m5 Avg(t) = 3.41 ms For, m6, m7 Avg(t) = 7.16 ms For, m8, m9, m10 Avg (t) = 4.83 ms For, m11 Avg (t) = 1.63 ms For, m14 Avg (t) = 9.44 ms For, m15 Avg (t) = 6.78 ms | For m5 Avg(t) = 3.55 ms For, m6, m7 Avg(t) = 7.33 ms For, m8, m9, m10 Avg (t) = 5.01 ms For, m11 Avg (t) = 1.76 ms For, m14 Avg (t) = 9.71 ms For, m15 Avg (t) = 6.93 ms |

creation of files with the .wnry extension. This is due to our system's ability to identify known ransomware extensions and detect suspicious activity during the CreateFile operation. Similarly, malware samples m4 with the .roger extension, and m12 and m13 with the .krab extension, were promptly detected by our system, following the same principle.

Unlike the aforementioned cases, the other nine samples do not employ the known ransomware extensions early in their lifecycle. Instead, they encrypt the files first and subsequently utilize functions such as MoveFile or MoveFileWithProgress to rename the encrypted files with their preferred extensions. Among these nine malware samples, six both created ransom note as file(s) and changed the background to effectively convey the message of encryption and provide payment details. Conversely, the remaining three samples opted for a simpler approach, generating a readme-like file that solely contained the necessary payment instructions to notify the user of the compromised state of their system.

We have conducted an analysis to measure the time difference between the execution with and without the implementation of our approach, in order to assess the additional overhead introduced by our hookings. The results are presented in Table 4.3. Based on the findings, it is evident that our hookings introduce a maximum response time increase of 11%. The lowest observed increase is approximately 1%, while the average increase ranges between 2% and 3%. These increments are minimal and negligible considering the significance of timely ransomware detection. Moreover, the observed increases are in the order of milliseconds, further emphasizing their negligible impact on the overall system performance.

### 4.5.3 Evaluation against the Benign Applications

To comprehensively evaluate our system, it is crucial to assess its performance in terms of false positives and response delay when tested against common benign applications. In this evaluation, we selected several widely used applications, including

Mozilla Firefox (Browser), Notepad, Calculator, MS Word, MS PowerPoint, Zoom, 7Zip, and WinSCP. The results of this assessment are presented in Table 4.4.

During the evaluation, we executed these benign applications and performed basic operations to determine if our system flagged them as ransomware. Additionally, we measured the additional response time incurred by the API hooking and the accompanying MSG/Extension checks. As expected, our system accurately identified these benign applications as non-ransomware instances, as they did not exhibit any behavior indicative of ransomware activity. Although a few instances of encryption and file deletion/renaming were observed, they were not classified as ransomware since no files with known ransomware extensions or ransom notes were created. Consequently, these benign applications were registered in the whitelist after a designated timeout period.

To assess the response delay attributed to the API hookings and associated checks, we conducted the same tasks using the aforementioned benign applications, both with and without our system, while utilizing the Selenium automation tool to ensure precise measurements. Table 4.4 reveals that none of the applications were flagged as ransomware during the CreateFile stage, as anticipated. Although Mozilla Firefox, 7Zip, and WinSCP exhibited ransomware-like behavior such as encryption and file deletion/overwriting, their absence of ransom note creation prevented them from being classified as ransomware.

Furthermore, we performed an analysis to quantify the additional overhead introduced by our hookings, as reflected in the response time. The results indicate a maximum response time increment of 28.26% for 7Zip, followed by Mozilla Firefox (23.002%) and WinSCP (21.12%). In comparison, the remaining applications demonstrated response time increments ranging from 0.91% to 1.71%, which are negligible and have minimal impact on system performance.

Table 4.4: Comparison of false positives using benign applications to test our system. The table displays the presence of ransomware-like behavior and relevant MSGs indicated by a checkmark (✓). The abbreviations CF, ENC, FD, and RN represent CreateFile with known ransomware extensions, Encryption, File deletion/overwrite, and Ransom note creation, respectively. The last two columns compare the response delay for the same task conducted without and with our system, with the increment (INC) of response time provided in the last column within brackets.

| **Benign application** | **Detection stage** | | | | **Time(t) required to reach point "A" by the application without our system** | **Time(t) required to reach point "A" by the application with our system (INC)** |
|---|---|---|---|---|---|---|
| | CF | ENC | FD | RN | | |
| Notepad | X | X | X | X | 2.10 s | 2.13 s (1.42%) |
| Calculator | X | X | X | X | 1.17 s | 1.19s (1.71%) |
| MS Word | X | X | X | X | 3.28 s | 3.31 s (0.91%) |
| MS PowerPoint | X | X | X | X | 5.66 s | 5.72 s (1.06%) |
| Mozilla Firefox | X | ✓ | ✓ | X | 12.39 s | 15.24 s (23.002%) |
| Zoom | X | X | X | X | 6.23 s | 6.31s (0.96%) |
| 7zip | X | ✓ | ✓ | X | 32.7 s | 41.94 s (28.26%) |
| WinSCP | X | ✓ | ✓ | X | 25.9 s | 31.37 s (21.12%) |

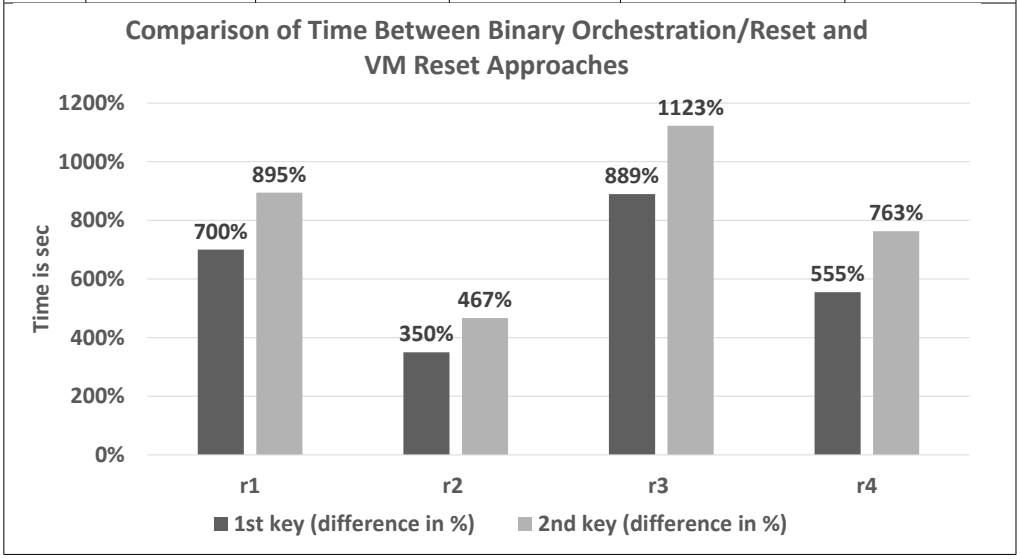
### 4.5.4 Accuracy and Performance Analysis of Binary Orchestration in the Reset Phase

To thoroughly evaluate the efficacy and precision of our binary orchestration approach, it is imperative to have access to the server-side code of ransomware, which enables us to observe the ongoing transmission of encryption keys from the malware to the server. While acquiring the server-side code presents certain challenges, we have managed to obtain four publicly available ransomware samples, namely NekRos Ransomware [95], Cryptonite [96], Jasmin Ransomware [97], and a proof-of-concept Windows crypto-ransomware [98]. These samples are accompanied by their corresponding server code, and we employ them to assess the accuracy and performance of our system during the reset phase. In subsequent sections, we will refer to them as r1, r2, r3, and r4, respectively.

**Accuracy Assessment**: To conduct the accuracy assessment, we acquired the source code of these four malware samples and compiled them to generate the corresponding executables. Subsequently, we executed the client component (intended for the victim's system) within gExtractor and extracted the relevant addresses. The

Table 4.5: Comparison of Time Between Binary Orchestration/Reset and VM Reset Approaches

| Sample | Time required to extract addresses from the gExtractor | Time recorded to receive the first key entry from the client (using binary orchestration/reset) | Time recorded to receive the first key entry from the client (using VM restart) | Time recorded to receive the second key entry from the client (using binary orchestration/reset) | Time recorded to receive the first key entry from the client (using VM restart) |
|---|---|---|---|---|---|
| r1 | 11 s | 24 s | 192 s | 38 s | 378 s |
| r2 | 15 s | 48 s | 216 s | 78 s | 442 s |
| r3 | 9 s | 19 s | 188 s | 31s | 379 s |
| r4 | 13 s | 31 s | 203 s | 46 s | 397 s |




execution chains and operations of all four samples were identified as expected, as depicted in Figure 4.4. The Actuating agent then prepared the Deceptor DLL based on the extracted addresses from gExtractor, injecting it into the malware process. Following this step, we observed the arrival of new notifications indicating victim infections, accompanied by associated decryption keys and unique victim IDs, being transmitted repeatedly. This confirmation affirms the successful and accurate implementation of binary orchestration.

**Performance Assessment**: The time required by the gExtractor to execute malware and collect execution information is not taken into consideration, as it is a standalone module used from a previous study [89] to facilitate binary orchestration. Once we receive the execution trace from the gExtractor, we proceed with extracting addresses from the generated trace and configuring the Deceptor DLL, which is then injected into the malware process through the Actuating agent. Therefore, our focus is on the time required for address extraction from the gExtractor log and the time

taken to receive the first and second key storage requests at the server's (attacker's) end.

Alternatively, we can achieve the same effect by repeatedly running the malware from a clean state. However, this requires rerunning the malware in a Virtual Machine (VM) capable of varying the data collected during the collection phase. Each time the server receives a key storage request, it stores the sent information in its database and sends a request to a controller that restarts the VM hosting the malware from a clean state, repeating the steps again. While this VM reset approach can achieve the desired result, it is time-consuming and resource-intensive. Therefore, we compared our binary orchestration approach with this setup to highlight the significant time savings achieved. The findings are presented in Table 4.5, clearly demonstrating the efficiency of our approach in quickly restarting the malware from a fresh state. From Table 4.5, it is evident that the VM reset approach requires 350% to 1123% more time to achieve the same impact at the attacker's side.

Once the binary orchestration successfully establishes a loop within the malware, the malware continuously sends key storage requests, resulting in the population of the database on the attacker's side. We conducted an experiment running this setup for a full day and recorded the number of entries created in the attacker's database at intervals of 1 hour, 12 hours, and 24 hours. Additionally, we measured the increment in database size caused by these extra entries. The recorded measurements are presented in Table 4.6.

At the end of the full day, our system generated a maximum of 8,513 entries for the ransomware sample r3, and a minimum of 2,532 entries for the ransomware sample r2. In terms of memory consumption, our system utilized a maximum of 1,479.87 KB in the attacker's database for the ransomware sample r4, and a minimum of 592.49 KB for the ransomware sample r2 using a single agent. These results confirm the memory consumption at the attacker's side achieved by our approach. It is worth noting that

Table 4.6: Resource Depletion at the Attacker's End: Extra Database Entries Generated by Binary Orchestration and the Additional Space Required to Store Them

| sample | Extra DB Entries (misinformation) | | | Additional space required due to extra DB entries | | |
|---|---|---|---|---|---|---|
| | 1 Hour | 12 Hours | 24 Hours | 1 Hour | 12 Hours | 24 Hours |
| **r1** | 268 | 3198 | 6484 | 27.751 KB | 334.34 KB | 671.39 KB |
| **r2** | 107 | 1284 | 2532 | 23.647 KB | 292.75 KB | 592.49 KB |
| **r3** | 356 | 4189 | 8513 | 46.28 KB | 565.52 KB | 1132.23 KB |
| **r4** | 197 | 2319 | 4728 | 60.87 KB | 730.48 KB | 1479.87 KB |

with multiple agents in action, we have the potential to consume even more memory at the attacker's side.

Our evaluation demonstrates that our proposed ransomware detection approach is accurate, effective, and efficient. It achieved a 100% detection rate for real-world ransomware samples and minimized false positives with benign applications. The additional response time introduced by the system was negligible. Binary orchestration proved to be faster and more efficient than the VM reset approach, resulting in significant time savings. The resource depletion analysis highlighted the impact on database size and memory consumption. Overall, our approach offers a robust solution for real-time ransomware detection and mitigation.

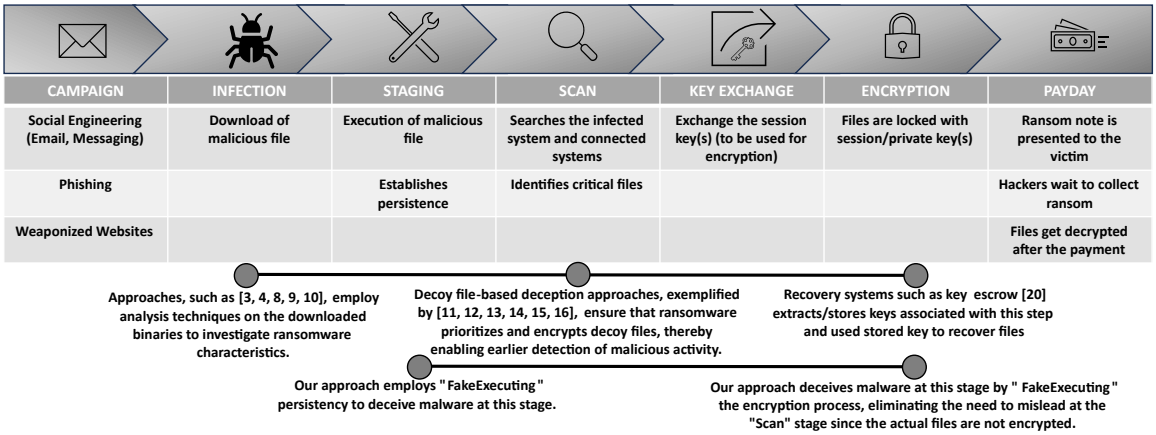



Figure 4.6: Our system integrates deception mechanisms during the Staging and Encryption stages to identify and neutralize/defuse ransomware encryption within its kill chain

## 4.6 Related work

This section provides an overview of pertinent studies that address various aspects of ransomware defense. Initially, we explore different approaches to ransomware detection and recovery, aligning them with the context of the ransomware kill chain, as depicted in Figure 4.6. As depicted in Figure 4.6, certain approaches, such as those proposed by [76], [77], [81], [82], [83], focus on mitigating ransomware at the "Infection" stage prior to execution. Ideally, addressing ransomware at this stage allows for isolated analysis within a secure environment to ascertain malicious intent. Since these approaches primarily target the pre-"Staging" phase of the kill chain, they fall outside the scope of comparison in this research work.

However, in scenarios where ransomware manages to execute either unintentionally or by utilizing evasion techniques to reach the staging phase, approaches such as those proposed by [84], [24], [85], [86], [87], [88], and our proposed approach can still provide protection to users. These post-"Infection" stage detection methods are discussed and compared with our approach in the following section.

### 4.6.1 Non-Deception Based Ransomware Detection

Various techniques have been proposed for ransomware detection, targeting different stages of the attack lifecycle. Static analysis methods utilize machine learning or signature-based approaches to identify ransomware samples based on their code or behavioral patterns [76, 77]. While these techniques exhibit high precision, they can be easily evaded by sophisticated obfuscation techniques employed by modern ransomware variants. Machine learning-based approaches that rely on static analysis data inherit these limitations.

Dynamic analysis approaches aim to detect ransomware based on its behavior during runtime. These techniques monitor system activities and identify malicious behavior patterns associated with ransomware operations [78–80]. However, a drawback

of these approaches is that they rely on observing system behavior to make decisions, which means that some sensitive files may already be encrypted by the time the system takes action. Machine learning-based approaches that utilize dynamic analysis data face similar limitations.

Isolated sandbox environments have also been used for ransomware detection. New binaries are executed within isolated environments, allowing for dynamic analysis and the identification of malicious behavior before execution on production systems [81–83]. While effective, the practicality of this approach in real-world scenarios may vary, as it assumes that the malware executable will be uploaded to a sandbox environment, which is not always the case.

In a key escrow-based solution called PayBreak [99], the authors propose saving the information related to symmetric keys generated to decrypt the locked files after the infection process. This proactive approach relies on a secure key escrow where only the user has exclusive access. While this approach can recover the encrypted data, it does not stop the attack and requires additional storage to store these keys. Furthermore, it includes keys belonging to benign processes, resulting in unnecessary overhead and storage requirements.

A few approaches, such as [78,100], utilize data backup mechanisms to continuously update and maintain a duplicate copy of the original system, enabling data recovery after a ransomware attack. However, these approaches do not detect or stop the attack itself and incur additional storage costs to store backup data.

### 4.6.2 Deception-Based Ransomware Detection

Deception-based approaches leverage decoy files or deceptive techniques to identify and deceive ransomware. These methods strategically place simulated or "honey" files throughout the file system, monitoring any attempts to access these files as anomalous behavior [24, 84–88]. While effective in minimizing data loss, ransomware can bypass this approach by specifically targeting certain files, rendering the deception strategy

ineffective.

Our approach diverges from file-based deception by focusing on deceiving ransomware at the API level. Through the interception and monitoring of ransomware API calls, we can detect and analyze ransomware activities without compromising sensitive files. Moreover, our solution offers cost-effectiveness and scalability advantages. Unlike file/directory-based solutions that necessitate the management and maintenance of decoy files throughout the entire file system, our approach centers around a single point (API call), resulting in improved scalability and cost efficiency. Additionally, our approach does not rely on the placement of decoy files or the targeting of specific file locations, making it resilient against ransomware that selectively searches and encrypts files. Furthermore, our system prevents non-whitelisted applications from engaging in actual encryption, deletion, or overwriting of files, effectively safeguarding the integrity of the original files and eliminating the need for additional storage expenses.

Furthermore, we introduce an automated method to identify critical addresses within ransomware binaries, facilitating the establishment of a looping mechanism. This mechanism compels the ransomware to repeatedly initiate from the beginning, transmitting encryption information and keys back to the attacker. Consequently, the attacker's resources are depleted, effectively mitigating the impact of ransomware attacks. This unique aspect of our work differentiates it from previous studies in the field.

## 4.7 Discussion & Conclusion

While our approach exhibits promising outcomes in the detection of ransomware and resource depletion, it is important to acknowledge certain limitations. Firstly, our method relies on identifying specific malicious subgraphs (MSGs) during the real-time phase and using them to extract crucial addresses in the reset phase. This may result in reduced effectiveness against ransomware variants that employ different MSGs.

Additionally, the efficacy of our approach may be influenced by evasion techniques employed by ransomware.

To further enhance our approach, future research can focus on developing a more comprehensive understanding of evolving ransomware behaviors by incorporating a broader range of relevant MSGs and their associated evasion strategies. Furthermore, investigating the scalability and performance of our method in larger-scale environments would ensure its practicality and efficiency.

In this study, we introduced a proactive and dynamic approach for detecting ransomware, utilizing API-level deception and automated binary orchestration to deplete attackers' resources. Our method demonstrated notable accuracy in identifying ransomware without any false positives, effectively mitigating the risks of file encryption and data loss. By strategically depleting attackers' resources through continuous confirmation of encryption and transmission of relevant information, our approach serves as a significant deterrent against ransomware attacks. The experimental evaluation conducted with real-world malware and benign applications validated the effectiveness, accuracy, and resource depletion capability of our proposed method. Despite certain limitations, our approach constitutes a valuable contribution to the field of ransomware detection and opens avenues for further research and improvement.

# CHAPTER 5: Conclusion and Future Work

## 5.1 Conclusion

In conclusion, the field of cybersecurity faces significant challenges in detecting and preventing advanced cyber threats. Existing methods for detecting malware behaviors have limitations, as attackers continuously evolve their techniques to evade detection. Active Cyber Deception (ACD) has emerged as a promising approach to overcome these limitations by actively misleading attackers and disrupting their decision-making processes. However, current deception techniques lack agility, resilience, and automation, making them easily detectable and circumventable by skilled attackers.

To address these shortcomings, this dissertation presents three innovative approaches. The first approach introduces DodgeTron, an autonomous cyber deception framework that combines hybrid dynamic analysis and machine learning to automate the creation of deception schemes against malware. DodgeTron leverages deep analysis and clustering techniques to extract deception parameters and categorize malware into known families. This knowledge is used to create a Deception Playbook consisting of HoneyThings for orchestrating deceptive environments. When new malware is detected, the system classifies it into a known family and utilizes the Deception Playbook to deploy the necessary HoneyThings to deceive the malware.

The second approach introduces symbSODA, an autonomous cyber deception system that combines symbolic execution capabilities with API hooking to extract comprehensive malicious sub-graphs (MSGs). These MSGs are then mapped to the MITRE ATT&CK framework to understand the high-level behavior of the malware. symbSODA creates a deception playbook consisting of various ploys tailored to each

MSG, employing different deception goals and strategies. During runtime, symbSODA detects these MSGs within unknown malware and executes the corresponding deception ploys through embedded deceptive API hooks. The approach also incorporates a Deception Planning Verifier to ensure consistent and conflict-free deception actions. In certain cases, if the FakeExecute deception strategy is selected, symbSODA can also leverage DodgeTron to orchestrate the deceptive environment with HoneyThings alongside deceptive API hooks.

The third approach is centered on early detection of ransomware during its lifecycle through the use of API-level deception, as opposed to file-level deception. This approach leverages the extracted malicious sub-graphs (MSGs) obtained from symbSODA and employs API-level deception with FakeSuccess as the chosen deception strategy. By monitoring the runtime behavior of the program without making any modifications to the original file system, this approach eliminates the need for managing and distributing decoy files, resulting in a more scalable and cost-effective solution. Additionally, it establishes a strategic loop within the malware that effectively turns the malware into a channel to feed misinformation back to the attacker. This process depletes the attacker's resources by continuously confirming encryption and transmitting keys relevant to the victim.

Key takeaways from these approaches include the importance of agile and automated deception techniques that can adapt to evolving cyber threats. By combining dynamic malware analysis and symbolic execution, these approaches effectively extract important deception features and provide proactive strategies for defenders to mitigate cyber threats. They bridge the information and resource asymmetry between attackers and defenders, offering effective ways to detect and mislead attackers while minimizing the impact on sensitive files and resources.

Overall, these innovative approaches demonstrate the potential of active cyber deception as a powerful defense technique in the ever-evolving landscape of cybersecurity

and ways of orchestration. They offer new avenues for detecting and mitigating cyber threats while minimizing the impact on sensitive files and resources.

## 5.2 Limitations and Future Work

While this dissertation effectively addresses several critical challenges related to automated deception feature extraction and orchestration, it also presents some limitations that pave the way for future research directions.

One limitation of our current approach is the manual effort required to generate honeyfiles by installing actual software to maintain their close resemblance to real systems. As a future research direction, we are exploring the feasibility of leveraging ChatGPT to automatically generate these honeyfiles. By utilizing ChatGPT, we anticipate a reduction in manual work and storage requirements, as the honeyfiles can be generated on the fly.

Another limitation arises from the use of symbolic execution, which inherits certain constraints such as state-space explosion. To mitigate the impact of state space explosion, we currently impose limitations on path exploration. However, this approach may not effectively discover interesting malware execution code. Future work could focus on addressing the challenges associated with symbolic execution's path explosion by designing an optimized symbolic execution-based analysis agent that can assist in more efficient malware analysis.

While our approach incorporates both dynamic and static deception ploys, they are predefined and may not cover all possible scenarios encountered during runtime, especially when encountering new malicious sub-graphs (MSGs). To overcome this limitation, an interesting future direction would be the generation of autonomous deception strategies and actions based on the analysis of attacker behaviors and system vulnerabilities. The MSG2MITRE framework plays a significant role in achieving this objective. By mapping our MSG2MITRE framework with recently introduced threat intelligence frameworks that provide generic deception actions, it becomes possible to

generate autonomous deception ploys at runtime under specific circumstances. This advancement can greatly enhance the flexibility and adaptability of the deception system.

# REFERENCES

[1] M. L. Bringer, C. A. Chelmecki, and H. Fujinoki, "A survey: Recent advances and future trends in honeypot research," *International Journal of Computer Network and Information Security*, vol. 4, no. 10, p. 63, 2012.

[2] J. Rrushi, "Honeypot evader: Activity-guided propagation versus counter-evasion via decoy os activity," in *Proceedings of the 14th IEEE International Conference on Malicious and Unwanted Software*, 2019.

[3] A. Vetterl and R. Clayton, "Bitter harvest: Systematically fingerprinting low- and medium-interaction honeypots at internet scale," in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[4] L. Alt, R. Beverly, and A. Dainotti, "Uncovering network tarpits with degreaser," in *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 156–165, 2014.

[5] M. M. Islam and E. Al-Shaer, "Active deception framework: an extensible development environment for adaptive cyber deception," in *2020 IEEE Secure Development (SecDev)*, pp. 41–48, IEEE, 2020.

[6] N. Provos *et al.*, "A virtual honeypot framework.," in *USENIX Security Symposium*, vol. 173, pp. 1–14, 2004.

[7] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis, "Detecting targeted attacks using shadow honeypots," in *USENIX Security Symposium*, 2005.

[8] J. Yuill, M. Zappe, D. Denning, and F. Feer, "Honeyfiles: deceptive files for intrusion detection," in *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004.*, pp. 116–122, IEEE, 2004.

[9] M. N. Alsaleh, J. Wei, E. Al-Shaer, and M. Ahmed, "gextractor: Towards automated extraction of malware deception parameters," in *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop*, pp. 1–12, 2018.

[10] M. S. I. Sajid, J. Wei, M. R. Alam, E. Aghaei, and E. Al-Shaer, "Dodgetron: Towards autonomous cyber deception using dynamic hybrid analysis of malware," in *2020 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9, IEEE, 2020.

[11] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," CCS '14.

[12] A. Niakanlahiji, J. H. Jafarian, B.-T. Chu, and E. Al-Shaer, "Honeybug: Personalized cyber deception for web applications," 2020.

[13] K. J. Ferguson-Walter, M. M. Major, C. K. Johnson, and D. H. Muhleman, "Examining the efficacy of decoy-based and psychological cyber deception," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[14] S. Jajodia, A. K. Ghosh, V. S. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*. Springer, 2012.

[15] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer Publishing Company, Incorporated, 1st ed., 2011.

[16] M. M. Islam, Q. Duan, and E. Al-Shaer, "Specification-driven moving target defense synthesis," in *Proceedings of the 6th ACM Workshop on Moving Target Defense*, pp. 13–24, 2019.

[17] M. Janbeglou, M. Zamani, and S. Ibrahim, "Redirecting network traffic toward a fake dns server on a lan," in *2010 3rd International Conference on Computer Science and Information Technology*, vol. 2, pp. 429–433, IEEE, 2010.

[18] E. Al-Shaer, *Toward Network Configuration Randomization for Moving Target Defense*, pp. 153–159. Springer New York, 2011.

[19] Q. Duan, E. Al-Shaer, M. Islam, and H. Jafarian, "Conceal: A strategy composition for resilient cyber deception-framework, metrics and deployment," in *2018 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9, IEEE, 2018.

[20] M. M. Islam, A. Dutta, M. S. I. Sajid, E. Al-Shaer, J. Wei, and S. Farhang, "Chimera: Autonomous planning and orchestration for malware deception," in *2021 IEEE Conference on Communications and Network Security (CNS)*, IEEE, 2021.

[21] E. Al-Shaer, J. Wei, W. Kevin, and C. Wang, *Autonomous Cyber Deception*. Springer, 2019.

[22] F. De Gaspari, S. Jajodia, L. V. Mancini, and A. Panico, "Ahead: A new architecture for active defense," in *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense*, SafeConfig '16, (New York, NY, USA), p. 11–16, Association for Computing Machinery, 2016.

[23] Z. A. Genç, G. Lenzini, and D. Sgandurra, "On deception-based protection against cryptographic ransomware," in *DIMVA*, 2019.

[24] C. Moore, "Detecting ransomware with honeypot techniques," in *CCC*, 2016.

[25] M. Akiyama, T. Yagi, K. Aoki, T. Hariu, and Y. Kadobayashi, "Active credential leakage for observing web-based attack cycle," in *RAID*, 2013.

[26] B. M. Bowen, P. Prabhu, V. P. Kemerlis, S. Sidiroglou, A. D. Keromytis, and S. J. Stolfo, "Botswindler: Tamper resistant injection of believable decoys in vm-based hosts for crimeware detection," in *RAID*, 2010.

[27] L. Krämer, J. Krupp, D. Makita, T. Nishizoe, T. Koide, K. Yoshioka, and C. Rossow, "Amppot: Monitoring and defending against amplification ddos attacks," in *RAID*, 2015.

[28] M. Akiyama, T. Yagi, T. Hariu, and Y. Kadobayashi, "Honeycirculator: distributing credential honeytoken for introspection of web-based attack cycle," *International Journal of Information Security*, 2018.

[29] M. Akiyama, T. Yagi, T. Yada, T. Mori, and Y. Kadobayashi, "Analyzing the ecosystem of malicious url redirection through longitudinal observation from honeypots," *Computers & Security*, 2017.

[30] X. Fu, B. Graham, R. Bettati, and W. Zhao, "On countermeasures to traffic analysis attacks," in *IEEE Systems, Man and Cybernetics SocietyInformation Assurance Workshop, 2003.*

[31] X. Fu, W. Yu, D. Cheng, X. Tan, K. Streff, and S. Graham, "On recognizing virtual honeypots and countermeasures," in *2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing, 2006.*

[32] M. L. Bringer, C. A. Chelmecki, and H. Fujinoki, "A survey: Recent advances and future trends in honeypot research," *International Journal of Computer Network and Information Security*, vol. 4, no. 10, p. 63, 2012.

[33] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A practical approach for adaptive data structure layout randomization," in *ESORICS 2015.*

[34] M. N. Alsaleh, J. Wei, E. Al-Shaer, and M. Ahmed, "gextractor: Towards automated extraction of malware deception parameters," SSPREW'18.

[35] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *ACM Sigplan Notices'11.*

[36] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014.

[37] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013.

[38] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[39] M. Zolotukhin and T. Hämäläinen, "Detection of zero-day malware based on the analysis of opcode sequences," in *CCNC*, 2014.

[40] DodgeTron Demo. `https://drive.google.com/drive/folders/1wQcfVf-rWDe1vrOl4KDGDbdnGOvmq5Av?usp=sharing`.

[41] Hunting Raccoon: The new masked bandit on the block. `https://www.cybereason.com/blog/hunting-raccoon-stealer-the-new-masked-bandit-on-the-block`.

[42] Rob Pantazopoulos, "Loki-Bot: Information Stealer, Keylogger, More!."

[43] A. Continella, M. Carminati, M. Polino, A. Lanzi, S. Zanero, and F. Maggi, "Prometheus: Analyzing webinject-based information stealers," *Journal of Computer Security*, 2017.

[44] A. Buescher, F. Leder, and T. Siebert, "Banksafe information stealer detection inside the web browser," in *RAID*, 2011.

[45] S. Kyung, W. Han, N. Tiwari, V. H. Dixit, L. Srinivas, Z. Zhao, A. Doupé, and G.-J. Ahn, "Honeyproxy: Design and implementation of next-generation honeynet via sdn," in *2017 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9, IEEE, 2017.

[46] N. Provos and T. Holz, *Virtual honeypots: from botnet tracking to intrusion detection*. 2007.

[47] R. Di Pietro and L. V. Mancini, *Intrusion Detection Systems*. Springer Publishing Company, Incorporated, 1 ed., 2008.

[48] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient detection of split personalities in malware," in *NDSS'10*.

[49] M. L, C. K., and M.Paolo, "Detecting Environment-Sensitive Malware," in *Proc. of RAID'11*.

[50] J. Wilhelm and T. Chiueh, "A forced sampled execution approach to kernel rootkit identification.," in *Proc. of RAID 2007*.

[51] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *Proc. of S&P 2007*.

[52] M. S. I. Sajid, J. Wei, B. Abdeen, E. Al-Shaer, M. M. Islam, W. Diong, and L. Khan, "Soda: A system for cyber deception orchestration and automation," in *Annual Computer Security Applications Conference*, pp. 675–689, 2021.

[53] N. C. Rowe, "Counterplanning deceptions to foil cyber-attack plans," in *IEEE Systems, Man and Cybernetics SocietyInformation Assurance Workshop, 2003.*, pp. 203–210, IEEE, 2003.

[54] N. C. Rowe, "Finding logically consistent resource-deception plans for defense in cyberspace," in *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, vol. 1, pp. 563–568, IEEE, 2007.

[55] VirusTotal, "Internet security, file and url analyzer." `https://www.virustotal.com`.

[56] MalShare. `https://malshare.com/index.php`. Online; accessed 10 May 2019.

[57] "Easyhook - the reinvention of windows api hooking," Online.

[58] S. E. Inc., "Stack Overflow - Where Developers Learn, Share, & Build Careers." `https://stackoverflow.com/`, 2023.

[59] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[60] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[61] M. Bobaru, C. Pasareanu, and D. Giannakopoulou, "Automated assume-guarantee reasoning by abstraction refinement," pp. 135–148, 07 2008.

[62] "Dissecting the windows defender driver - wdfilter (part 1)," Online.

[63] "Interactive analysis: Any.run," Online.

[64] popescuadi, "Ransomware - simple c++ ransomware, prove the concept.." `https://github.com/popescuadi/Ransomware`, 2017.

[65] "Keylogger-screen-capture," Online.

[66] K. Oosthoek and C. Doerr, "Sok: Att&ck techniques and trends in windows malware," in *International Conference on Security and Privacy in Communication Systems*, pp. 406–425, Springer, 2019.

[67] O. Alrawi, M. Ike, M. Pruett, R. P. Kasturi, S. Barua, T. Hirani, B. Hill, and B. Saltaformaggio, "Forecasting malware capabilities from cyber attack memory images," in *30th USENIX Security Symposium*, 2021.

[68] "Cuckoo sandbox," Online.

[69] "Any.run," Online.

[70] J. Zhang, Z. Gu, J. Jang, D. Kirat, M. Stoecklin, X. Shu, and H. Huang, "Scarecrow: Deactivating evasive malware via its own evasive logic," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 76–87, IEEE, 2020.

[71] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, (USA), p. 351–366, USENIX Association, 2009.

[72] N. C. Rowe, J. Rrushi, *et al.*, *Introduction to cyberdeception*. Springer, 2016.

[73] D. Kirat, G. Vigna, and C. Kruegel, "Barebox: efficient malware analysis on bare-metal," in *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 403–412, 2011.

[74] "Internet security threat report, ransomware, 2017.."

[75] "Smb exploited: Wannacry use of eternalblue."

[76] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "{UNVEIL}: A {Large-Scale}, automated approach to detecting ransomware," in *25th USENIX security symposium (USENIX Security 16)*, pp. 757–772, 2016.

[77] J. Baldwin and A. Dehghantanha, "Leveraging support vector machine for opcode density based detection of crypto-ransomware," *Cyber threat intelligence*, pp. 107–136, 2018.

[78] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "Shieldfs: a self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd annual conference on computer security applications*, pp. 336–347, 2016.

[79] D. Sgandurra, L. Muñoz-González, R. Mohsen, and E. C. Lupu, "Automated dynamic analysis of ransomware: Benefits, limitations and use for detection," *arXiv preprint arXiv:1609.03020*, 2016.

[80] N. Scaife, H. Carter, P. Traynor, and K. R. Butler, "Cryptolock (and drop it): stopping ransomware attacks on user data," in *2016 IEEE 36th international conference on distributed computing systems (ICDCS)*, pp. 303–312, IEEE, 2016.

[81] O. M. Alhawi, J. Baldwin, and A. Dehghantanha, "Leveraging machine learning techniques for windows ransomware network traffic detection," *Cyber threat intelligence*, pp. 93–106, 2018.

[82] S. Kok, A. Abdullah, and N. Jhanjhi, "Early detection of crypto-ransomware using pre-encryption detection algorithm," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 5, pp. 1984–1999, 2022.

[83] J. Hwang, J. Kim, S. Lee, and K. Kim, "Two-stage ransomware detection using dynamic analysis and machine learning techniques," *Wireless Personal Communications*, vol. 112, pp. 2597–2609, 2020.

[84] J. A. Gómez-Hernández, L. Álvarez-González, and P. García-Teodoro, "R-locker: Thwarting ransomware action through a honeyfile-based approach," *Computers & Security*, vol. 73, pp. 389–398, 2018.

[85] Y. Feng, C. Liu, and B. Liu, "Poster: A new approach to detecting ransomware with deception," in *38th IEEE symposium on security and privacy*, 2017.

[86] S. Mehnaz, A. Mudgerikar, and E. Bertino, "Rwguard: A real-time detection system against cryptographic ransomware," in *International symposium on research in attacks, intrusions, and defenses*, pp. 114–136, Springer, 2018.

[87] Z. Wang, X. Wu, C. Liu, Q. Liu, and J. Zhang, "Ransomtracer: exploiting cyber deception for ransomware tracing," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pp. 227–234, IEEE, 2018.

[88] S. Sheen, K. Asmitha, and S. Venkatesan, "R-sentry: Deception based ransomware detection using file access patterns," *Computers and Electrical Engineering*, vol. 103, p. 108346, 2022.

[89] M. N. Alsaleh, J. Wei, E. Al-Shaer, and M. Ahmed, "gextractor: Towards automated extraction of malware deception parameters," in *the 8th Software Security, Protection, and Reverse Engineering Workshop (SSPREW 2018)*, ACM, 2018.

[90] M. S. I. Sajid, J. Wei, E. Al-Shaer, Q. Duan, B. Abdeen, and L. Khan, "symbsoda: Configurable and verifiable orchestration automation for active malware deception," *ACM Transactions on Privacy and Security*, 2023.

[91] "Ransomware encrypted file extension list."

[92] "Google cloud vision api documentation."

[93] Y. Lemmou, J.-L. Lanet, and E. M. Souidi, "In-depth analysis of ransom note files," *Computers*, vol. 10, no. 11, p. 145, 2021.

[94] "objdump is a command-line program for displaying various information about object files and binary files."

[95] "Nekros is an open-source ransomeware."

[96] "Open source ransomware toolkit cryptonite."

[97] "Jasmin ransomware is an advanced red team tool (wannacry clone) used for simulating real ransomware attacks.."

[98] "A poc windows crypto-ransomware (academic)."

[99] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, "Paybreak: Defense against cryptographic ransomware," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pp. 599–611, 2017.

[100] S. Baek, Y. Jung, A. Mohaisen, S. Lee, and D. Nyang, "Ssd-insider: Internal defense of solid-state drive against ransomware with perfect data recovery," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 875–884, IEEE, 2018.