gExtractor: **Towards Automated Extraction of Malware Deception Parameters**

Mohammed Noraden Alsaleh*
Eastern Michigan University
Ypsilanti, Michigan
malsaleh@emich.edu

Jinpeng Wei, Ehab Al-Shaer, Mohiuddin Ahmed
University of North Carolina at Charlotte
Charlotte, NC
{jwei8,ealshaer,mahmed27}@uncc.edu

ABSTRACT

The lack of agility in cyber defense gives adversaries a significant advantage for discovering cyber targets and planning their attacks in stealthy and undetectable manner. While there has been significant research on detecting or predicting attacks, adversaries can always scan the network, learn about countermeasures, and develop new evasion techniques. 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, developing an efficient active deception environment usually requires human intelligence and analysis to characterize the attackers' behaviors (e.g., malware actions). This manual process significantly limits the capability of cyber deception to actively respond to new attacks (malware) in a timely manner.

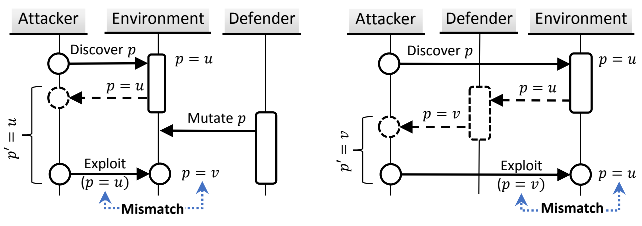
In this paper, we present a new analytic framework and an implemented prototype, called *gExtractor*, to analyze the malware behavior and automatically extract the deception parameters using symbolic execution in order to enable the automated creation of cyber deception schemes. The deception parameters are environmental variables on which attackers depend to discover the target system and reach their goals; Yet, they can be reconfigured and/or misrepresented by the defender in the cyber environment. Our *gExtractor* approach contributes to the scientific and system foundations of reasoning about autonomous cyber deception. Our prototype was developed based on customizing a symbolic execution engine for analyzing Microsoft Windows malware. Our case studies of recent malware instances show that *gExtractor* can be used to identify various critical parameters effective for cyber deception.

1 INTRODUCTION

Malware attacks have evolved to be highly evasive against prevention and detection techniques. It has been reported that at least 360,000 new malicious files were detected every day and one ransomware attack was reported every 40 seconds in 2017 [25]. This

*Most of this research was performed while being a PhD student at UNC Charlotte

SSPREW-8, December 3–4, 2018, San Juan, PR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6096-8/18/12…$15.00
https://doi.org/10.1145/3289239.3289244



(a) Mutation. **(b) Misrepresentation.**
Figure 1: Attacker's Dependency on System Parameters.

reveals severe limitations in the prevention and detection technologies, such as anti-virus, perimeter firewalls, and intrusion detection systems. Active Cyber Deception (ACD) has emerged as an effective defense for cyber resilience [13] that can corrupt and steer adversaries' decisions to (1) *deflect* them to false targets, (2) *distort* their perception about the environment, (3) *deplete* their resources, and (4) *discover* their motives, tactics, and techniques [1, 14].

Advanced cyber threats often start with intensive reconnaissance by interacting with cyber to learn the true values of its parameters, such as keyboard layout, geolocation, hardware ID, IP address, service type, OS/platform type, and registry keys to discover vulnerable targets and achieve their goals. We call such parameters "Critical Parameters". ACD can be particularly effective during this phase by providing false perceptions about the configuration of the cyber environment [1]. There are two key mechanisms to accomplish this: (1) parameter *mutation* to frequently change the ground truth, the real values of the system parameters, such as IP address and route mutation [12, 16], or (2) parameter *misrepresentation* to change only the values discovered by the attacker, while the ground truth is intact. We call such critical parameters that can be feasibly and cost-effectively mutated or misrepresented the "Deception Parameters". Figure 1 shows the two deception mechanisms with respect to the environment parameter p. It shows that the adversary knowledge about p was falsified by either changing p to a new value (mutation) or lying about its true value (misrepresentation). Both mechanisms are needed in cyber deception because mutation can be infeasible or too expensive, and misrepresentation can sometimes be uncovered.

Effective planning of cyber deception may require a sequence of mutations and/or misrepresentations of deception parameters in order to steer the adversary, represented by malware code, to desired deception goals (i.e., deflection, distortion, depletion, or discovery). However, the key challenge that we address in this work is to identify the most appropriate *deception parameters* against given malware. For example, let us consider a Trojan horse that steals trade secrets only if the victim computer's keyboard layout is for Brazil. If we want to feed the adversary misleading trade information by running the Trojan in the United States, we have to first make the victim machine's keyboard layout appear to be for Brazil, and then plant fake documents for the Trojan to fetch. Our goal in this case is to automatically identify that the keyboard layout, among others, is a candidate deception parameter.

In this paper, we present a systematic approach and automated tool to analyze malware binary code and identify *"what"* deception parameters can accomplish the deception goals. This requires deception-oriented analysis of malware behaviors, which goes beyond existing dynamic analysis that is usually tailored towards attack detection. Thus, we extend the existing dynamic analysis and symbolic execution frameworks, which track the execution of malware symbolically, to analyze system and library API calls that particularly entail interactions with the cyber environment. We then identify the deception parameters that can impact the malware decision-making. Since these parameters can be interdependent and they might exhibit varying deception accuracy and cost, our analysis selects consistent sets of parameters that creates resilient and cost-effective deception schemes. We summarize our contributions as follows:

- We present *gExtractor*, a deception-oriented malware analysis tool that intercepts and tracks the malware interactions with the environment, and maps them to specific deception parameters.
- We developed formal constraints to extract deception parameters that constitute consistent, resilient, and cost-effective deception.
- We implemented *gExtractor* and evaluated it using various types of malware codes. Our evaluation demonstrates the ability of *gExtractor* to extract effective deception parameters.

While some previous work, such as Moving Target Defense (MTD) [12, 17, 33, 35, 38, 41, 42] and decoy technologies [3, 23, 34] attempt to invalidate attacker's perception, the deception parameters and schemes were engineered manually, which significantly limits the ability for creating deception actions automatically against novel malware. The ultimate goal of this research is to automate active cyber deception against malware attacks. Thus, unlike IPS/IDS, our objective is to detect and deceive, rather than detect and block, by enabling the malware to execute in a real or virtual deception environment configured based on the extracted deception parameters. To the best of our knowledge, this is the first work that uses automated reasoning to infer deception parameters based on malware analysis.

We implemented *gExtractor* on top of the Selective Symbolic Execution engine (S^2E) [7] with the assist of our custom plugins to execute malware in a real controlled environment, intercept system and library API calls, mark the relevant symbolic information, and collect execution logs. This facilitates the construction of a comprehensive malware behavior model that covers all possible execution paths. The constructed model is further processed to (1) prune out execution paths that are not relevant to the deception goals, and (2) eliminate the don't-care symbolic variables that have no impact on the deception goals.

To demonstrate the value of our approach, we used *gExtractor* to analyze about 30 recent variants of three major malware families: Cryptocurrency-mining malware, ransomware, and Credential-stealing malware. We discuss in the evaluation section one representative for each family by modeling its behavior, extracting candidate deception parameters, and showing how they can be used to design different deception schemes for different goals. Our case studies presented in this paper show that our approach can discover effective deception parameters. For example, the bitcoin miner case study (Section 4.1) reveals multiple parameters including Windows Script Host engine, win32_processor WMI (Windows Management Instrumentation) class that can be used to deflect the malware by misinforming false platform type, and the bitcoin hashing results that can be used to corrupt the results in mining pool and depleting the adversary resources (i.e., score).

The rest of this paper is organized as follows: In Section 2, we present the process of constructing the malware behavior model by executing the malware symbolically. Then, we present our approach to refine the malware behavior models and extract candidate deception parameters in Section 3. Real malware case studies are presented in Section 4. Finally, we discuss the related works and conclude in Section 5 and 6, respectively.

2 MODELING ATTACK BEHAVIOR USING BINARY SYMBOLIC EXECUTION

To extract the complete behavior of a cyber attack, we execute its binaries (i.e., malware) symbolically and build a model that represents its behavior with respect to selected system parameters. Given that the correct set of system parameters is selected, symbolic execution can cover all relevant execution paths. Before going through the technical steps of the symbolic malware analysis, we present the attack behavior model.

2.1 Attack Behavior Model

The *attack behavior model* describes how the attack behaves based on the results of its interaction with the environment. The malware interacts with its environment through system and user library APIs characterized by their input and output arguments. Some of these arguments may be attacker-specific variables and cannot be controlled by the environment, while other parameters can be reconfigured or misrepresented. We assume that a mapping between the selected system or library APIs' arguments and the corresponding parameters in the environment, such as *files*, *registry entries*, *system time*, *processes*, *keyboard layout*, *geolocations*, *hardware ID*, *C&C*, *Internet connection*, *IP address* or *host name*, and *communication protocols*, is given. For example, the *from* argument of the *recvfrom* API can be mapped to a system parameter that represents the IP address of the sender machine.

We define the attack behavior model as a graph of *Points of Interaction (PoI)* nodes and *Fork* nodes. The PoIs refer to the points in the malware control flow at which the malware interacts with the environment by invoking system or library APIs. The fork nodes represent the points in the control flow at which the malware makes a control decision based on the results of its interactions with the environment.

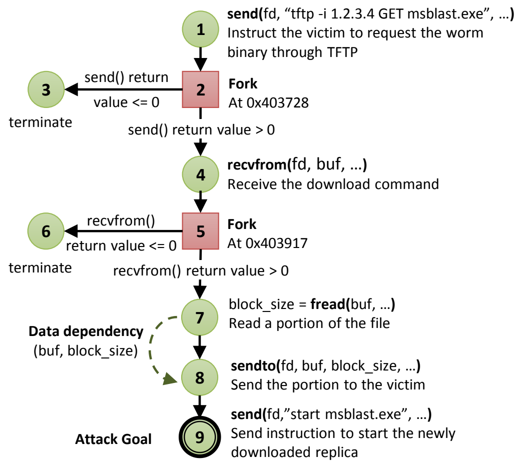

send(fd, "tftp -i 1.2.3.4 GET msblast.exe", ...)
Instruct the victim to request the worm binary through TFTP
1
2
Fork
At 0x403728
send() return value <= 0
3
terminate
send() return value > 0
4
recvfrom(fd, buf, ...)
Receive the download command
5
Fork
At 0x403917
recvfrom() return value <= 0
6
terminate
recvfrom() return value > 0
7
block_size = fread(buf, ...)
Read a portion of the file
Data dependency
(buf, block_size)
8
sendto(fd, buf, block_size, ...)
Send the portion to the victim
9
Attack Goal
send(fd,"start msblast.exe", ...)
Send instruction to start the newly downloaded replica

Figure 2: Example of Attack Behavior Model

To formally model the attack behavior, let Γ be the set of selected System and Library APIs, where each $\gamma \in \Gamma$ takes a fixed number of input augments ($I_\gamma = \{i_1, ..., i_n\}$) and returns a fixed number of output arguments ($O_\gamma = \{o_1, ..., o_m\}$). We model the attack behavior as the directed graph $G = (P, \mu, E, \nu)$, where:

- P is a set of nodes that represent the PoI and Fork nodes. The type function $\mu : P \longrightarrow \{\boldsymbol{PoI}, \boldsymbol{Fork}\} \times (\Gamma \cup \emptyset)$ associates nodes with their types. If the node represents a PoI, μ further maps it to the appropriate system and library API from the set Γ.
- $E \subseteq P \times P$ is the set of edges that represents the dependency between the nodes in P. A directed edge $e = (p_i, p_j)$ is added from node p_i to node p_j if there is a control or data dependency between them. The dependency function $\nu : E \longrightarrow \mathcal{L}_O$ associates each edge to a constraint expressed as a logic formula in the logic $\mathcal{L}_O$ with support for quantifier-free integer, real, and bit-vector linear arithmetic. Expressions in $\mathcal{L}_O$ are defined over the set of output arguments $O = \bigcup_{\gamma \in \Gamma} O_\gamma$.

In Figure 2, we show an example of attack behavior model that represents a portion of the Blaster worm that delivers a copy of the worm to an exploited victim. Round nodes represent PoIs and square nodes represent fork points. The solid edges represent control dependency, while dashed ones represent data dependency. In this model, the worm first sends an instruction to a remote command shell process running on the exploited victim through the *send* library API, then it waits for a download request through the *recvfrom* API call. The attack code checks if these operations are executed successfully and terminates otherwise as depicted through the conditions shown on the outbound edges from the fork nodes 2 and 5. At node 7, the worm starts reading its executable file from the disk into a memory buffer, through *fread*, and sending the content of the buffer to the remote victim, through the *sendto* API. There is a data dependence between the third argument of the *sendto* call, which represents the number of bytes to transmit, and the return value of the *fread* call, which represents the number of bytes read from the worm file.

2.2 Malware Symbolic Execution

We utilize the S^{2}E engine to symbolically execute malware binaries. The path coverage and the progress of the executed program depends on the correct marking of symbolic variables. Since we are interested in the interactions of the malware with its environment through selected system and library APIs, we intercept these calls and mark their output arguments as symbolic. This allows us to capture the malware decisions based on those arguments and track the corresponding execution paths. In the current version of our implementation, we select about 130 APIs that cover activities related to networking, file system and registry manipulation, system information and configuration, system services control, and UI operations. We list these APIs in Appendix A.

Marking Symbolic Variables. To mark the appropriate symbolic variables, we take advantage of the *Annotation* plugin provided by S^{2}E, which combines monitoring and instrumentation capabilities and executes user-supplied scripts, written in LUA language, at run time when a specific annotated instruction or function call is encountered. We define an annotation entry for each API. The annotation entry consists of the module name, the address of the API within the module, and the annotation function. We identified the module names and addresses using static/dynamic code analysis tools, such as IDA and Ollydbg. The annotation function is executed at the exit of the intercepted call. It reads the addresses of the return and output arguments of the call and marks the appropriate memory locations and registers as symbolic. Note that output arguments may have different sizes and structures. Hence, we need custom scripts to mark each individual output argument of the intercepted APIs. The return values of APIs are typically held in the *EAX* register and we use special method provided by S^{2}E to mark its value as symbolic. It should be noted that system calls and user library APIs are invoked by all applications in the environment, not only the malware process. Therefore, our annotation functions check the name of the process that invokes them and ignore calls from irrelevant processes.

Building the Attack Behavior Model. After preparing the appropriate annotation entries, we execute the malware using S^{2}E to collect the execution traces. We configured the annotation functions to record the arguments, the call stack, and other meta-data, such as the time-stamp and the execution path number for each intercepted system and library call. By design, S^{2}E intercepts branch statements whose conditions are based on symbolic variables and forks new states of the program for each possible branch. We collect the traces and branching conditions of all execution paths and build the attack behavior model as follows:

- We create a PoI node for each system or library API call logged by our annotation functions. Similarly, the traces contain special log entries for state forking operations. Those are used to create the *Fork* nodes in our model.
- For each node in the model, we add a control dependency edge from the node preceding it in the execution path. If the preceding node is a *Fork* node, the edge will be associated with a branching condition in terms of the symbolic variables.
- To capture the data dependency, we check the values of all the input arguments upon the entry of each API call. If the value

is a symbolic expression, this implies that it is a transformation of previously created symbolic variables. Hence, we add a data dependency edge from the PoI nodes in which the symbols of the expression were created.

3 DECEPTION PARAMETERS EXTRACTION

Given the attack behavior model generated through symbolic execution, we extract a set of system parameters that help in designing effective deception schemes to meet the deception goals. Recall that the attack behavior model describes the complete behavior of a malware with respect to selected system parameters. However, that does not mean that every parameter in the attack behavior model is a feasible candidate for deception. That is, mutating or misrepresenting its value may not be sufficient to successfully deceive the attacker. We analyze the attack behavior model to select the appropriate set(s) of deception parameters that can help in designing deception schemes without dictating particular ones.

We present the following four criteria ($C1$ - $C4$) that must be considered to decide on which parameters are appropriate for effective deception and which are not:

- $C1$ (Goal Dependency): the selected deception parameters can directly or indirectly affect the outcomes of the attack in terms of whether the attacker can reach her goal. Hence, parameters that are used only in execution paths that do not lead to particular goals might be excluded.
- $C2$ (Resilience): in cases where multiple attack paths lead to particular goals, selected parameters must provide deception in all the paths, not only one.
- $C3$ (Consistency): the selected deception parameters must preserve the integrity of the environment from the attacker's point of view. As system parameters may be interdependent, deception schemes must take this into consideration, such that misrepresenting one parameter without misrepresenting its dependents accordingly does not disclose the deception.
- $C4$ (Cost-Effectiveness): although multiple parameters may exist in the execution paths leading to particular goals, mutating or misrepresenting different parameters may require different costs and provide different benefits from the defender's point of view. Defenders must select the most cost-effective set of parameters for deception.

To extract the parameters that satisfy the four criteria, we follow two steps. First, we refine the attack behavior model to eliminate irrelevant execution paths and symbolic variables, which ensures that the refined model contains only parameters that satisfy $C1$. Second, we construct a constraints optimization problem based on the refined model. We add the appropriate constraints to extract the parameters that satisfy $C2$ and $C3$. Further, we encode the estimated costs of using the candidate deception parameters to select the most cost-effective set that complies with $C4$.

3.1 Refining the Attack Behavior Model

The complete attack behavior model contains many execution paths that may not be relevant to our deception analysis. In this refinement step, we (1) identify the set of execution paths that are relevant to deception and (2) eliminate the don't-care symbolic variables.

Identifying Relevant Paths. Recall that deception is not about blocking attacks, rather, it is about misleading and forcing them to follow particular paths that serve the desired deception goals. Hence, the selection of relevant execution paths from the attack behavior model depends on the deception goal. Following our definitions of the four goals of deception, the paths relevant to distortion keep the malware misinformed about the environment to slow it down or force it to make more environment checks. This is reflected in the paths that exhibit aggressive interactions and queries with and about the environment. On the other hand, the relevant paths for depletion and discovery are those that lead the malware to interact with the remote master or adversaries, while paths in which the malware loses interest and abandons the system are relevant to the deflection goal.

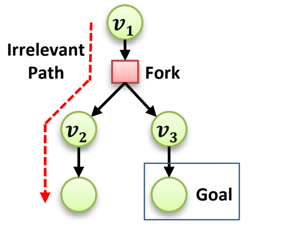
Definition 3.1 (Relevant Paths). A relevant path with respect to a particular deception goal is an execution path that exhibits particular patterns of interactions with the environment that can be leveraged by the defender to achieve the deception goal.

Regardless of which deception goal is desired, it can be represented as a single call or a sequence of calls to system and library APIs. To recognize deception goals, we can leverage existing techniques that identify specific behaviors through patterns of call sequences, such as [8, 29, 31]. Then, the PoI nodes in our attack behavior model will be used to identify the execution paths that exhibit that particular sequence of calls. By pruning out all other paths that do not exhibit the desired sequence, we end up with a portion of the original behavior model that contains only the paths relevant to the deception goal. In Figure 3a, we show a simple example of an attack behavior model that has two paths, one leads to the desired goal and the other leads to attack termination. In this case, the left path is considered irrelevant and it will be pruned out. For a concrete real-world example, in order to deceive the FTP Credential Stealer malware in Section 4.2 with honey FTP passwords, the environment must not run OllyDbg because otherwise the malware would follow an execution path irrelevant to the deception goal.

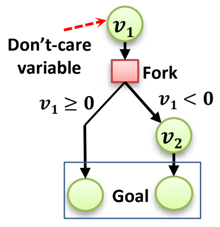
Eliminating Don't-Care Variables. To clarify this step, we need first to define the execution path constraints. A path constraint is a logical expression that captures the conditions on the selected symbolic variables that need to be met in order for the execution to follow that particular path. Recall that we associate a set of symbolic variables to each PoI node p in the attack behavior model ($p \in P, \mu(p) = PoI$), which correspond to its output arguments. Later in the execution, an expression will be generated for each branch at the following forking nodes in terms of the symbolic variables causing the fork. Those expressions are captured in the resulting edges of the fork nodes and mapped through the dependency function $\nu(.)$.

The constraint of an entire path in our attack behavior model is simply the conjunction of the logical expressions associated with all the edges that belong to the path. Formally, let $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, where $(\forall_{i\in[1,n]} : p_i \in P)$, represents a node path in the attack behavior model. Further, let $e_{i,j} \in E$ denote the edge between the nodes $p_i \in \mathcal{P}$ and $p_j \in \mathcal{P}$. The path constraint of the execution path represented by $\mathcal{P}$ can be computed as $\bigwedge_{i\in[1,n-1]} \mu(e_{i,i+1})$, where $\mu(e_{i,i+1})$ is the expression of the edge $e_{i,i+1}$. We define the don't-care symbolic variables as follows.

Definition 3.2 (Don't-Care Variables). A don't-care variable with respect to particular deception goal is a symbolic variable that is

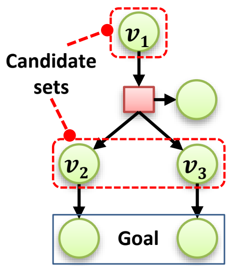


(a) Relevant Path Selection.

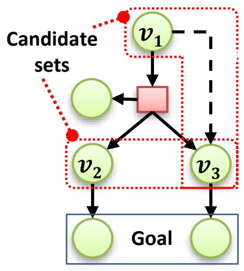


(b) Don't-care Elimination.

Figure 3: Attack Behavior Model Refinement.



(a) Resilient Deception.



(b) Consistent Deception.

Figure 4: Deception Parameters Selection.

part of one or multiple execution path constraints and its value is irrelevant to the desired deception goals.

As Figure 3b illustrates, although there is a decision taken based on the symbolic variable v_1, the desired goal will be reached regardless of the variable's value. This makes v_1 a don't-care variable with respect to the desired goal and it can be excluded from further deception analysis.

After eliminating the irrelevant paths and the don't-care variables, we end up with refined path constraints for the relevant paths. Any parameter extracted based on this refined model complies with $C1$ criteria.

3.2 Selecting Deception Parameters

In this step, we define a constraints optimization problem based on the refined attack behavior model to find an optimal set of deception parameters. Each PoI node in the refined attack behavior model is associated with a set of symbolic variables, which represent the output arguments of the malware interactions with the environment.

Although, the symbolic variables augment the attacker's perception about the environment, extracting the deception parameters out of them is not a trivial process due to the following. First, there is no necessarily one-to-one mapping between the symbolic variables and the system parameters since the output of one interaction may be determined by multiple system parameters. Hence, we need to map the symbolic variables to the appropriate system parameters utilizing experts knowledge of the system and the system and library APIs. The documentation of the APIs can also be used to extract this mapping as it normally specifies the possible outputs of APIs and the cases in which each value is returned based on the system and the environment states. Second, multiple candidate sets of deception parameters may exist in the model. As illustrated in Figure 4a, selecting either $\{v_1\}$ or $\{v_2, v_3\}$ satisfies $C2$ criteria, but they might be associated with different costs. Third, the interdependence between system parameters may mandate selecting additional parameters to satisfy $C3$, which increases the cost of deception. For example, in Figure 4b, although selecting $\{v_1\}$ is sufficient to satisfy $C2$, we also need to select $\{v_3\}$ to satisfy $C3$ because of the dependency of v_3 on v_1. To satisfy $C4$, our selection must consider all possible candidate sets to find the optimal cost-effective one.

To formalize the problem of selecting the optimal set of deception parameters, let V be the set of system parameters and let S be the set of symbolic variables in the refined attack behavior model. We define the following mapping functions:

- $\theta(.) : S \longrightarrow 2^V$ is the parameters assignment function that maps each symbolic variable in the path constraints to the corresponding system parameter(s).
- $\delta(.) : V \longrightarrow Z^*$ is the cost function that determines the cost of deception through each system parameter. Z^* is the set of non-negative integers.

Based on these assumptions, the deception parameters selection is defined as the problem of selecting a set of system parameters, such that (1) at least one parameter is selected for each relevant path (to comply with $C2$), (2) if a parameter is selected, all its dependent are also selected (to comply with C3), and (3) the selected parameters incur the minimum cost on the defender (to comply with $C4$).

To model this problem, we define the set of Boolean variables $\{d_1, d_2, ..., d_m\}$ with a variable for each system parameter in V. This set represents the decision variables of the constraints optimization problem, where d_i is set to 1 if the i-th parameter in V is selected for deception and d_i is set to 0 otherwise. Then, we unfold the refined attack behavior model into the set $\mathcal{T}$ of paths, where each $t \in \mathcal{T}$ is a sequence of symbolic variables ($t \subseteq S$). The first constraint that at least one deception parameter is selected for each path is expressed as follows:

$$\bigwedge_{t \in \mathcal{T}} \left(\bigvee_{s \in t} \bigvee_{i \in \theta(s)} d_i \right) \tag{1}$$

where $\theta(s)$ is the parameters assignment function that returns the indices of the system parameters associated with the symbolic variable s. To calculate the total deception cost, we compute the value C that represents the cumulative cost of all the selected deception parameters.

$$C = \sum_{i \in [1,m]} (d_i \; ? \; \delta(v_i) \; : \; 0) \tag{2}$$

where v_i is the i-th element in the set V of system parameters and the notation $(\psi \; ? \; v_1 : v_2)$ represents the *if-then-else* construct that evaluates to the value v_1 if ψ is *true*, and to the value v_2 if ψ is *false*. We add another set of constraints to capture the dependency between the system parameters. If a parameter is selected for deception, all the dependent parameters must also be selected. To capture this set of constraints, let $\epsilon(.) : V \longrightarrow 2^V$ be a dependency function that maps each system parameter to a set of dependent parameters.

Then, we represent the dependency constraints as follows:

$$\bigwedge_{v_i \in V} \left(d_i \rightarrow \bigwedge_{j \in \epsilon(v_i)} d_j \right) \tag{3}$$

After adding the constraints we can solve the constraints optimization problems using a solver (e.g. [10]). The optimization objective in this problem is to minimize the cumulative deception cost denoted by the variable C in Equation 2. The result will be a set of system parameters that satisfies our four criteria to provide resilient, consistent, and cost-effective deception.

4 EVALUATION

We developed a prototype for gExtractor, consisting of two major parts: (1) four new S^2E plugins and minor modifications to the S^2E core, in about 1,700 lines of C++ code; (2) annotation scripts for 130 system and library APIs, in 6,450 lines of LUA code. The optimal deception parameter selection is under development.

We used *gExtractor* to analyze recent malware variants and extracted candidate deception parameters for each of them. The variants we analyzed represent most common types of malware, including Cryptocurrency-mining malware, ransomware, worms, spyware, and Credential-stealing malware. To demonstrate that our systematic approach can indeed extract effective deception parameters, we selected five of the most prevalent malware, namely, *Bitcoin Miner*, *FTP Credential-Stealer*, and three instances of ransomware (*Cerber*, *Locky*, and *Gandcrab*). We discuss in details the process of building the attack behavior model, extracting deception parameters, and we suggest deception schemes utilizing them.

4.1 Case Study I: Bitcoin Miner

We analyzed a recent bitcoin mining malware (MD5: efd1326e5289a9359195120fd6c55290) that works in several stages. First, it drops and runs a Visual Basic (VB) script. Second, the script queries the Windows Management Instrumentation (WMI) service for the processor's information, such as the availability of GPU and the system architecture (32-bit or 64-bit), to download the right executable file for the target system from an external distribution website, *winxcheats.tk*. Third, the downloaded executable (*csrs.exe*) downloads yet another executable (*AudioHD.exe*) from *getsoed9.beget.tech*. The last program (*AudioHD.exe*) interacts with a bitcoin mining pool server at *xmr.pool.minergate.com* to perform the mining on behalf of an account, which is hard-coded in the executable.

Malware Behavior. Using *gExtractor*, we construct the behavior model of this malware (see the simplified version in Figure 5), which covers the malware execution stages. We use common patterns of API calls to recognize significant malware activity. For example, the use of APIs that create new processes (e.g., *ShellExecuteExA* and *WshShell.Run*), indicates the beginnings of consecutive malware stages. Moreover, interacting with a well-known bitcoin mining pool server through networking and HTTP APIs reveals that one goal of this malware is to use the victim machine to perform bitcoin mining on behalf of the attacker. Therefore, we refine the malware behavior model by recognizing the relevant paths that lead to that goal and design deception schemes around it. After mapping the symbolic variables of the relevant paths' constraints to the system

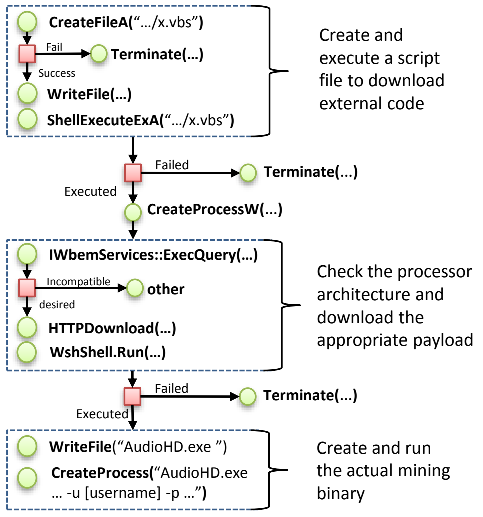

CreateFileA("…/x.vbs")
Fail
Terminate(…)
Success
WriteFile(…)
ShellExecuteExA("…/x.vbs")
Create and execute a script file to download external code
Failed
Terminate(…)
Executed
CreateProcessW(…)
IWbemServices::ExecQuery(…)
Incompatible
other
desired
HTTPDownload(…)
WshShell.Run(…)
Check the processor architecture and download the appropriate payload
Failed
Terminate(…)
Executed
WriteFile("AudioHD.exe ")
CreateProcess("AudioHD.exe … -u [username] -p …")
Create and run the actual mining binary

Figure 5: Simplified Behavior Model of the Bitcoin Miner

parameters, our analysis reveals the following necessary conditions for successful mining:

1. The file *C:\Windows\system32\wscript.exe* exists.
2. Windows Script Host (*WSH*) engine is enabled to run Visual Basic scripts.
3. *WMI* service and *Microsoft Win32 WMI* provider are running.
4. *win32_processor* WMI class reports the correct processor information.
5. The distribution website (*http://winxcheats.tk*) is available and hosts the executable file (under */miners/3/csrs.exe*).
6. The second distribution site (*getsoed9.beget.tech*) is available and hosts the second executable file (*AudioHD.exe*).
7. The bitcoin mining pool server (*xmr.pool.minergate.com*) is still running correctly.
8. The hard-coded account (*iden1930@mail.ru*) is authenticated successfully at the mining pool server.
9. The target system can run the file *AudioHD.exe* successfully.

To clarify how *gExtractor* facilitates the detection of such conditions, let us take condition 2 as an example. We mark the output parameter *"Buffer"* of the *RegQueryValueExW* API call, which is required to successfully complete the second stage of the malware, as symbolic. The API's input parameter, *hKey*, refers to the registry key *"HKLM\SOFTWARE\Microsoft\Windows Script Host\Settings\"* and the other input parameter *"ValueName"* is set to *"Enabled"*. Then we observe that *"Buffer"* is used in a conditional jump, and in one path the message *"Windows Script Host access is disabled on this machine"* is displayed before the process terminates, while in another path we do not see this message. Alternatively, we see multiple queries to the WMI service. The first path will be regarded as irrelevant and pruned out by *gExtractor* and we will only consider the later. Similarly, *gExtractor* can detect the dependence of this malware on the remaining conditions by tracking the decisions taken based on the associated symbolic variables and refining the behavior model.

Deception Parameters. We analyzed the refined bitcoin miner behavior model with respect to different deception goals: deflection, distortion, depletion, and discovery. We identified the major deception parameter that satisfies our criteria defined in Section 3.2 and can be utilized to achieve each goal. In the following, we discuss a number of recommended deception schemes based on these parameters and we provide a summary with estimated deception costs in Table 1.

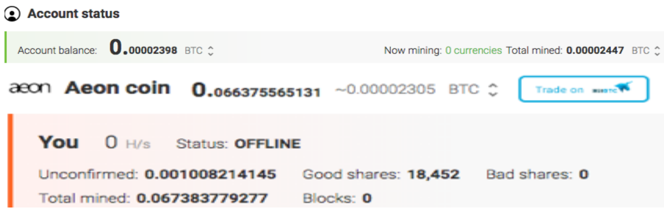
Deflection Schemes. For this purpose, we can enhance the designated script host *C:\Windows\system32\wscript.exe*. If the malicious VB script initiates a connection to a critical server, the enhanced *wscript.exe* can rewrite the VB script statement so that it connects to a honey server instead. This scheme could have high development cost because it requires a change to a Windows system utility, for which we do not have a source code. In terms of operation cost, this scheme can have high cost because it can confuse benign applications that need to run VB scripts, even if this is on a honeypot. However, it has little configuration cost because the current Windows OS does not have a configuration option to replace *wscript.exe* with an alternative version.

Discovery Schemes. We can use the *Windows Script Host (WSH) engine* to construct a discovery strategy against malware that needs to run VB scripts. The WSH enables applications to run VB scripts and JScripts, and it provides a configuration option (via Windows registry) to enable/disable the VB script support. By enabling it, we can observe malware behavior through its VB scripts and have a better understanding of the malware. This strategy incurs only a low configuration cost.

Distortion Schemes. Through the *"win32_processor WMI class"* parameter, we can construct a distortion scheme that returns misinformation about the system's architecture in order to confuse the malware (or the attacker behind the malware) who queries the *win32_processor class* interface. This strategy requires a change to the implementation of the *win32_processor class* interface, so there can be some development cost, and it can have a low operation cost if it is used on a honeypot.

Depletion Schemes. The last parameter in Table 1 is the resulting hash, which the malware sends reliably to the mining pool server. We can create a depletion strategy by corrupting the results so that they become invalid. Deceiving the malware to send excessive invalid results damages the attacker's reputation or cause financial losses (e.g., the mining pool server bans her account, freezes her mining wallet, or applies a penalty to her account). This scheme requires writing code to carry out the scheme, so it has some development costs. It has no operation cost because it modifies only the data of the malware.

We have experimentally confirmed the feasibility of depleting the attacker by corrupting the resulting hashes. In order to profit from mining on a victim machine, the attacker communicates with the mining pool server under her mining username in order to receive the credit. Therefore, at mining time, the attacker username must be present on the victim machine. We leveraged this fundamental "vulnerability" of this malware (i.e., revealing the mining username) for an effective deception. Based on our study of multiple mining pools, they establish various penalty policies for participants who submit invalid hashes. In Table 2, we summarize the negative impact



Account status

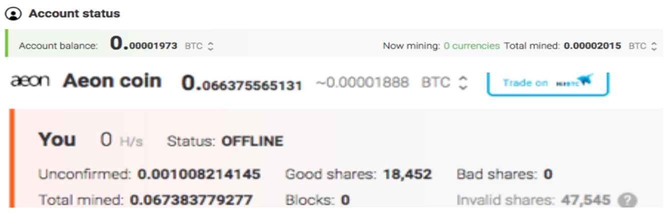
Account balance: 0.00002398 BTC
Now mining: 0 currencies Total mined: 0.00002447 BTC

aeon Aeon coin 0.066375565131 ~0.00002305 BTC

You 0 H/s Status: OFFLINE

Unconfirmed: 0.001008214145 Good shares: 18,452 Bad shares: 0
Total mined: 0.067383779277 Blocks: 0

Figure 6: Account Status before Mining for Aeon Coin



Account status

Account balance: 0.00001973 BTC
Now mining: 0 currencies Total mined: 0.00002015 BTC

aeon Aeon coin 0.066375565131 ~0.00001888 BTC

You 0 H/s Status: OFFLINE

Unconfirmed: 0.001008214145 Good shares: 18,452 Bad shares: 0
Total mined: 0.067383779277 Blocks: 0 Invalid shares: 47,545

Figure 7: Account Status after Submission of 40,000 Invalid Shares for Aeon Coin

of submitting invalid shares to several public mining pools. We can see that misbehaving users are often banned to some extent and their wallets can even be locked.

To prove the effectiveness of this depletion scheme, we built a tool that deliberately send invalid hashes on behalf of a particular user. Different mining pool servers may implement different protocols to authorize jobs and submit resulting hashes. However, most of them use a protocol called *STRATUM* over *HTTP* [36] and they define their own methods that can be used by the users to log into the server, get new jobs, and submit resulting hashes. We obtain the names and the required parameters of these methods along with other communication settings by analyzing the mining malware. Then, we submit a login request to the pool server. In response, the pool server returns a job and an Id that corresponds to the username. At that point, a legitimate miner will use the job data to generate a hash and send it back to the server. However, our tool will generate and send a random result instead, which will most likely be recognized as an invalid hash by the pool server.

To confirm that the mining pool servers penalize users who send invalid hashes, we created a user account at *Minergate* and sent a large number of random hashes on behalf of our new user. After submitting around $40,000$ invalid hashes, the account balance decreases from 0.00002398 to 0.00001973 which complies with the policy of *Minergate*. Figures 6 and 7 show the change in the account state before and after we sent the random hashes. Note that we have no means to verify whether a real attacker will be penalized if her username is used, because we do not have access to her account balance. However, when we perform the same actions using the attacker's username (iden1930@mail.ru) extracted from the malware analysis, the response from the mining pool server indicates that the submitted shares are detected as invalid. Since our account is penalized in compliance with the policy, we believe the attacker's account should be punished as well.

Table 1: Deception Schemes Against the Bitcoin Mining Malware
(CC: configuration cost, OC: operation cost, DC: development cost)

Parameter	Deception Goal	Deception Action	Estimated Cost
wscript.exe	Deflection	Replace it with a version that rewrites the input VB script for better protection	No CC; High OC; High DC
WSH engine	Discovery	Enable its capability to run VB scripts	Low CC; No OC; No DC
win32_processor WMI class	Distortion	Change the way that it handles requests (e.g., returning misinformation about processors)	No CC; Low OC if used on a honeypot; Medium DC
The resulting hash	Depletion	Corrupt the resulting hash	No CC; No OC; High DC

Table 2: Negative Impact of Submitting Invalid Hashes

Mining Pool	Banned	Payouts locked	Balance reduced
moneroocean.stream	For 1-10 min	No	No
xmrpool.net	Yes	No	No
supportxmr.com	Yes	Yes	No
www.viaxmr.com	Temporary	No	No
minergate.com	No	No	Yes
slushpool.com	Yes	No	No
moriaxmr.com	For 10 min	No	No
ratchetmining.com	For 10 min	No	No

4.2 Case Study II: FTP Credential-Stealer

In this case study, we analyze a recent malware (MD5: 7572fb188134-d141eac1751b19b79a70) that scans the victim system for sensitive information, such as FTP login passwords and then sends the stolen information to a remote server.

Malware Behavior. This malware consists of two processes. The first process employs multiple methods to check whether the malware is being analyzed, then terminates immediately if the checking result is positive. If no signs of analysis are detected, the first process drops and launches another piece of malware, which collects sensitive information from the victim system and sends it to a remote server under the adversary's control. A simplified version of the behavior model of this malware, generated by *gExtractor*, is shown in Figure 8.

The first malware process is heavily obfuscated and employs multiple tricks to evade analysis: (1) it tests whether the executable file's name contains any of the strings *"sandbox", "malware", "virus", or "self"*; (2) it scans the list of running processes for known dynamic analysis tools, such as *procmon.exe, procmon64.exe, procexp.exe, ollydbg.exe, and windbg.exe*; (3) it checks the *BeingDebugged* flag in its PEB (Process Environment Block) [11] at multiple places of its code section; (4) it checks whether it is running inside a virtual machine by matching the result of the *CPUID* instruction with *"KVMKVM", "XenVMM", "Microsoft Hv", and "pri hyperv"*; (5) it extracts the second malware binary from its resource section, decrypts it, and then uses process injection to launch it in a second process. If any sign of malware analysis is detected, the malware immediately terminates.

In the second process, the malware collects sensitive information from the Windows registry and the local file system, and sends it to a remote site as follows. First, it searches certain Windows

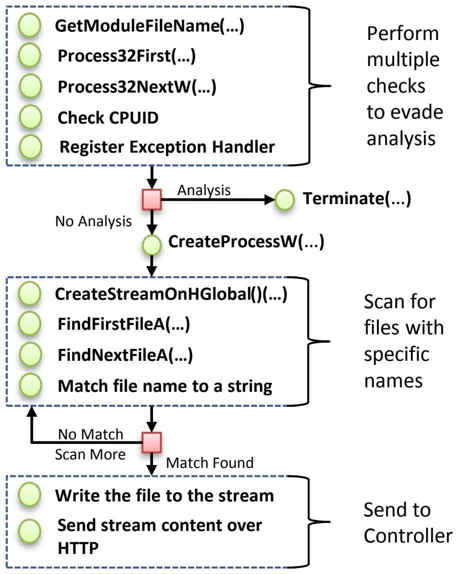

GetModuleFileName(...)
Process32First(...)
Process32NextW(...)
Check CPUID
Register Exception Handler
Perform multiple checks to evade analysis
Analysis
Terminate(...)
No Analysis
CreateProcessW(...)
CreateStreamOnHGlobal()(...)
FindFirstFileA(...)
FindNextFileA(...)
Match file name to a string
Scan for files with specific names
No Match
Scan More
Match Found
Write the file to the stream
Send stream content over HTTP
Send to Controller

Figure 8: Simplified Model of the FTP Credential-Stealer

registry keys, which correspond to a specific list of FTP clients, for saved login credentials. For example, to steal information related to WinSCP, it searches for the key "Software\Martin Prikryl". If the key is found, it recursively enumerates the subkeys with the names *"HostName", "UserName", "Password", "RemoteDirectory"*, and *"PortNumber"*, read their values, and stores them in a stream object for later exfiltration. Strings such as *"Software\Martin Prikryl"* and *"HostName"* are hard-coded in the malware. Second, it looks up files whose path contains particular patterns *(e.g., "WS_FTP", "LastSessionFile", "FTPRush", "Quick.dat", and "History.dat")*, and if any such file exists, it stores the file's path and content in the stream object. To optimize the search, it focuses on known folders, identified by their Constant Special Item ID List (CSIDL) values, such as the users' public documents, desktop, and local settings. It also searches the folders of installed applications discovered by their *"UninstallString"* registry values under the registry key *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall*.

After collecting the targeted information, the malware extracts the data from the stream object (via the API call sequence {*GetHGlobalFromStream*, *GlobalLock*}), then it constructs and sends a HTTP POST message to *"http://www.luxzar.com/drake/november/omg/hot/gate.php"*. The HTTP communication is conducted by the API call

Table 3: Defense Strategies against FTP Credential-Stealing Malware

(CC: configuration cost, OC: operation cost, DC: development cost)

Parameter	Deception Goal	Deception Action	Cost
Malware file name	Discovery	Avoid naming the malware “sandbox.exe”, “malware.exe”, “virus.exe”, or “self.exe”	Low CC; no OC
Dynamic analysis tools	Discovery	Rename the dynamic analysis tools	Low CC; no OC
Result of the *CPUID* instruction	Discovery	Deny that the true result is one of “KVMKVM”, “XenVMM”, “Microsoft Hv”, and “pri hyperv”	Low CC; High OC; High DC
Registry entries of WinSCP	Depletion	Plant encrypted and invalid FTP passwords to waste the energy of the attacker who tries to decrypt and use those invalid passwords	Low CC; Low OC; No DC
Registry entries of WinSCP	Discovery	Plant honey FTP passwords to entice the attacker to login to a honey FTP server	Medium CC; Low OC; No DC
Registry entries of applications that maintain login credentials	Depletion	Plant encrypted and invalid login credentials	Low CC; High OC; High DC
Registry entries of applications that maintain login credentials	Discovery	Plant honey login credentials	Medium CC; Low OC; No DC
Files that contain sensitive information	Depletion	Plant Honey files with seemingly sensitive information to waste the energy of the attacker who tries to act upon the content of the files	Low CC; Low OC; No DC
Files that contain sensitive information	Deflection Distortion Discovery	Depending on the meaning of the file content, plant crafted content that can help the defender achieve Deflection / Distortion / Discovery goals	Varying CC; Low OC; No DC

sequence {*InternetCrackUrlA, ObtainUserAgentString, socket, connect, setsocketopt, send, closesocket*}.

Deception Parameters. We employ the methods discussed in Section 3 to the behavior model we obtain from the above analysis. We recognize a number of deception parameters that enable different deception schemes, as summarized in Table 3.

Discovery Schemes. Since the malware applies many checks to evade analysis, these checks can be used to inspire effective discovery schemes that encourage the malware to run normally. Specifically, we can rename common analysis tools and modify the behavior of the *CPUID* instruction so that it gives an impression that the environment is not a virtual machine, which is commonly the case for malware analysis. The cost of renaming common dynamic analysis tools is low. However, the cost of manipulating the result of the *CPUID* instruction can be high: it is cheap if the environment has a way to intercept *CPUID* instructions in software (e.g., on top of QEMU), but it is infeasible otherwise. Alternatively, the registry entries of the FTP clients, such as WinSCP, can be leveraged to lure the attacker to honeypots so we can learn more about its capabilities and intents. We can create honey FTP accounts, save the honey login credentials in WinSCP, and run the malware so it delivers the honey login credentials to the attacker. The configuration cost of this kind of scheme is medium because it is necessary to set up the honey FTP server and deploy monitoring tools.

Depletion Schemes. The registry entries of the FTP clients can also be leveraged to feed the attacker fake login credential and deplete her resources and effort. For example, we can install WinSCP in the environment and save many sessions with fake values for the information targeted by the attacker (e.g., username and password) decreasing the likelihood of her landing on legitimate victims. An even better scheme is to create an encrypted version of an invalid password and save it in the Windows registry entry for WinSCP, which will give the attacker an additional burden to decrypt the password, thus further depleting the attacker’s resources.

Deception Schemes using the File System. Similar to registry entries, files that contain sensitive information are useful parameters for multiple goals: depletion, deflection, distortion and discovery. For example, we can plant honey files with seemingly sensitive but useless information to waste the energy of the attacker who tries to act upon the content of the honey files. Although the general idea is well known, the specific details as to which files should be planted can be greatly informed by analyzing the malware decisions. The cost of carrying out these kinds of strategies can vary depending on the purposes of the files: it may require simple editing of a file on one hand, or development of tools to create the files on the other; the operation cost may also vary depending on the purpose of the files: if they are used only by attackers, the cost is low, but if they are used by benign users, the cost can be quite steep, since the honey content can confuse benign users.

4.3 Case Study III: Cerber, Locky, and Gandcrab

Ransomware has moved from the 22nd most common variety of malware in the 2014 data breach investigations report to the fifth most common in 2017’s report [32]. We analyzed three instances of this family using *gExtractor*: Cerber, Locky, and Gandcrab and we summarize the analysis in the following. Since these instances encrypt files on the compromised computers, one can expect that their functionality relies heavily on file-related interactions. However, they share some functionality with traditional other types of malware. First, during the initialization, they access the system registry to setup the keys that guarantee persistence and they use mutant-based infection markers. Second, they communicate with a C&C server for key sharing and reporting. However, the communication is limited to information sharing and the server commands do not normally change the main goal of those malware.

Table 4: Selected File-Related Interactions

Interaction	System Parameter(s)
GetSystemDirectory	System Directory Path
SearchPath	File Existence
FindFirstFile	File Existence
NtQueryInformationFile	File Existence/Info
GetFileAttributes	File Existence/Attributes
GetFileSize	File Existence/Size
NtReadFile	File Content
NtWriteFile	File Content

Deception Parameters. In Table 4, we highlight the major interactions with the file system and we map them to system parameters. Note that some interactions depend on multiple parameters. For example, *GetFileSize* will fail if the file no longer exists and it if does, it will return the file size. Hence, it depends on both the existence of the file and its size. The persistence registry keys and the mutant infection markers, the common folders (e.g., temp and system directory), and the parameters in Table 4 (e.g., files names, sizes, and attributes) were all part of the relevant paths that lead to the goal and can be candidates for deception. In addition, after refining the behavior models, we observed a difference between *Cerber* and the *Locky* that confirms others' manual observations. Although *Cerber* called the *sendto* API to communicate with the C&C server, the encryption process starts regardless of the success of the communication. This means that the success of the *sendto* call, manifested as a symbolic variable, was not part of the conditions to reach the goal and it was pruned out. However, successful communication was a condition to start encryption in the *Locky* instance we evaluated.

Depletion Schemes. We developed a depletion scheme against the adversary behind ransomware, specifically Gandcrab (MD5: 8a45b0941ec2af89bfd9ed3-3dae2053f). The malware sends encrypted information about the victim to the C&C server, then receives a response message derived from the message sent, which implies that the C&C server has to process the message from the malware. Therefore, we can overwhelm the C&C server by sending a substantial number of messages to it in a brief period. We have experimentally verified that (1) it is possible to replay the same message many times while the C&C server responds to each individual message; (2) if we intentionally send a corrupted encrypted message, the C&C server responds with an error, which implies that it tries to decrypt the message but fails. Both confirm the feasibility of our deception scheme.

4.4 Challenges and Future Work

Through our case studies, we recognized a few technical challenges with respect to our approach. First, it is non-trivial to build a general deception parameter extraction technique due to inherent limitations of symbolic execution, for example, state space explosion. Second, a naive use of symbolic execution cannot effectively discover interesting malware dependency on the environment because the execution can slip into paths leading to other than the desired goals, such as getting stuck in loops. We have implemented simple heuristics to limit state forking inside code blocks that will be repeatedly executed and inside dynamically linked system libraries. However, we plan to develop new plugins that would guide the symbolic execution engine towards more meaningful paths leveraging existing approaches that were previously proposed to address similar challenges in dynamic taint analysis and mixed concrete and symbolic execution [19, 30]. Finally, currently we specify coarse-grained deception cost (e.g., Low or High); in future work we need to quantitatively measure the cost of deception through different system parameters.

5 RELATED WORK

Randomization and moving target defense are well-investigated techniques toward agile cyber that can proactively disrupt advanced attacks. Randomization techniques, such as instruction set randomization [28], compiler-generated software diversity [15] and address space layout Randomization [35], introduce unpredictability to confuse the adversary and invalidate her assumptions about the system. Moving target defense techniques, such as [12, 18, 33, 41, 42], mutate specific static system parameters proactively over time. For example, NASR [2] randomizes IP addresses based on DHCP over time. Similarly, the authors in [41] propose to periodically migrate VMs to make it harder for adversaries to locate targeted VMs. In another direction, deception techniques, such as honeynets and honeypots [3, 20, 23, 34], divert attackers away from their targets to consume their resources and protract their reconnaissance. Although these techniques and many other similar ones have been successful, within acceptable performance overheads, in deterring and deceiving the targeted attacks, they were designed in an adhoc manner to counteract specific attacks. Our proposed analytic framework makes this process systematic and decrease the need for manual intervention and the reliance on human intelligence to design effective active cyber deception schemes.

Analyzing and exposing behaviors of malware is another research topic that has been extensively discussed in the literature [4, 6, 24, 39, 40]. Forced execution [37] and X-Force [27] were designed for brute-force exhausting path space without providing semantics information for each path's trigger condition. To discover the trigger conditions, Brumley *et al.* [5] applied taint analysis and symbolic execution to derive the condition of malware's hidden behavior. Moser [26] introduced a snapshot based approach that could be applied to expose malware's environment-sensitive behaviors. Hasten [21] was proposed as an automatic tool to identify and skip malware's stalling code. In [22], Kolbitsch *et al.* proposed a multipath execution scheme for Java-script-based malware. Other research [9, 37] proposed techniques to force the execution of different malware functionalities. While our work needs to analyze malware, we have a different goal: to automatically discover system parameters that can be mutated or misrepresented to deceive, rather than detect, malware. We can benefit from all existing malware analysis techniques, and in this paper we choose symbolic execution in particular.

6 CONCLUSION

We present the first analytic framework towards automated creation of deception schemes based on rigorous malware binary code execution and automated reasoning of attack behaviors and decision-making process. We have implemented a tool that models the complete behavior of given malware in terms of its interactions with and dependence on the environment. We further analyze the malware behavior beyond traditional dynamic and symbolic malware analysis to track the malware decisions with respect to system parameters and identify those relevant to deception. Moreover, since multiple competing parameters may be identified, we select the optimal set of parameters that can be used to construct consistent, resilient, and cost-effective deception. We demonstrated through three detailed case studies how our deception-oriented analysis can lead to effective deception schemes against major malware types: cryptocurrency mining malware, credential-stealing malware, and ransomware. In addition, we have experimentally verified the deception schemes against bitcoin mining malware and ransomware.

REFERENCES

[1] Ehab Al-Shaer and Mohammad Ashiqur Rahman. 2015. Attribution, Temptation, and Expectation: A Formal Framework for Defense-by-Deception in Cyberwarfare. In *Cyber Warfare*. Springer, 57–80.

[2] Spyros Antonatos, Periklis Akritidis, Evangelos P Markatos, and Kostas G Anagnostakis. 2007. Defending against hitlist worms using network address space randomization. *Computer Networks* 51, 12 (2007), 3471–3490.

[3] Frederico Araujo, Kevin W. Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. 2014. From Patches to Honey-Patches: Lightweight Attacker Misdirection, Deception, and Disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 942–953. https://doi.org/10.1145/2660267.2660329

[4] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. 2010. Efficient Detection of Split Personalities in Malware. In *Proc of NDSS'10*.

[5] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. 2008. Automatically Identifying Trigger-based Behavior in Malware. In *Botnet Analysis and Defense*, W. Lee, C. Wang, and D. Dagon (Eds.). Vol. 36. Springer, 65–88.

[6] P. Royal C. Song and W. Lee. 2012. Impeding Automated Malware Analysis with Environment-sensitive Malware. In *Proc. of HotSec'12*.

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 2.

[8] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2007. Mining Specifications of Malicious Behavior. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/1287624.1287628

[9] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Krugel, and S. Zanero. 2010. Identifying Dormant Functionality in Malware Programs. In *Proc. of S&P'10*.

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[11] Nicolas Falliere. 2007. Windows Anti-Debug Reference. https://www.symantec.com/connect/articles/windows-anti-debug-reference. (2007). [Online; accessed 04-February-2018].

[12] Syed Fida Hussain Gillani, Ehab Al-Shaer, Samantha Lo ?, Qi Duan, and Mostafa Ammar ?and Ellen Zegura. 2015. Agile Virtualized Infrastructure to Proactively Defend Against Cyber Attacks. In *Infocom*.

[13] Harriet Goldman, Rosalie McQuaid, and Jeffrey Picciotto. 2011. Cyber resilience for mission assurance. In *Technologies for Homeland Security (HST), 2011 IEEE International Conference on*. IEEE, 236–241.

[14] Kristin E Heckman, Frank J Stech, Roshan K Thomas, Ben Schmoker, and Alexander W Tsow. 2015. *Cyber denial, deception and counter deception*. Springer.

[15] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. 2011. Compiler-generated software diversity. In *Moving Target Defense*. Springer, 77–98.

[16] Haadi Jafarian, Qi Duan, and Ehab Al-Shaer. 2016. Effective Address Mutation Approach for Disrupting Reconnaissance Attacks. *To appear in IEEE Transactions on Information Forensics and Security* (2016).

[17] J. H. Jafarian, E. Al-Shaer, and Q. Duan. 2015. An Effective Address Mutation Approach for Disrupting Reconnaissance Attacks. *IEEE Transactions on Information Forensics and Security* 10, 12 (Dec 2015), 2562–2577. https://doi.org/10.1109/TIFS.2015.2467358

[18] Sushil Jajodia, Anup K. Ghosh, V.S Subrahmanian, Vipin Swarup, Cliff Wang, and Xiaoyang Sean Wang (Eds.). 2013. *Moving Target Defense II - Application of Game Theory and Adversarial Modeling*. Advances in Information Security, Vol. 100. Springer.

[19] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. Dta++: dynamic taint analysis with targeted control-flow propagation.. In *NDSS*.

[20] Amanjot Kaur. 2013. Dynamic Honeypot Construction. (2013).

[21] C. Kolbitsch, E. Kirda, and C. Kruegel. 2011. The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code. In *Proc. of CCS'11*.

[22] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. 2012. Rozzle: De-Cloaking Internet Malware. In *Proc. of S&P'12*.

[23] Sukwha Kyung, Wonkyu Han, Naveen Tiwari, Vaibhav Hemant Dixit, Lakshmi Srinivas, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. 2017. HONEYPROXY: Design and Implementation of Next-Generation Honeynet via SDN. In *IEEE Conference on Communications and Network Security (CNS)*.

[24] Martina L, Clemens K., and M.Paolo. 2011. Detecting Environment-Sensitive Malware. In *Proc. of RAID'11*.

[25] Kaspersky Lab. 2017. Kaspersky Security Bulletin. Overall statistics for 2017. https://securelist.com/ksb-overall-statistics-2017/83453/. (2017).

[26] A. Moser, C. Kruegel, and E. Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *Proc. of S&P'07*.

[27] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 2014 USENIX Security Symposium*. San Diego, CA.

[28] Georgios Portokalidis and Angelos D Keromytis. 2011. Global ISR: Toward a comprehensive defense against unauthorized code execution. In *Moving Target Defense*. Springer, 49–76.

[29] Yong Qiao, Yuexiang Yang, Jie He, Chuan Tang, and Zhixue Liu. 2014. *CBM: Free, Automatic Malware Analysis Framework Using API Call Sequences*. Springer Berlin Heidelberg, Berlin, Heidelberg, 225–236. https://doi.org/10.1007/978-3-642-37832-4_21

[30] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 225–236.

[31] Madhu K. Shankarapani, Subbu Ramamoorthy, Ram S. Movva, and Srinivas Mukkamala. 2011. Malware Detection Using Assembly and API Call Sequences. *J. Comput. Virol.* 7, 2 (May 2011), 107–119. https://doi.org/10.1007/s11416-010-0141-5

[32] Verizon Enterprise Solutions. 2017. 2017 Data Breach Investigations Report. http://www.verizonenterprise.com/verizon-insights-lab/dbir/2017/. (2017).

[33] Nathaniel Soule, Borislava Simidchieva, Fusun Yaman, Ronald Watro, Joseph Loyall, Michael Atighetchi, Marco Carvalho, David Last, David Myers, and Capt Bridget Flatley. 2015. Quantifying & Minimizing Attack Surfaces Containing Moving Target Defenses. (2015).

[34] Jianhua Sun, Kun Sun, and Qi Li. 2017. CyberMoat: Camouflaging critical server infrastructures with large scale decoy farms. In *Communications and Network Security (CNS), 2017 IEEE Conference on*. IEEE, 1–9.

[35] PaX Team. 2015. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt. (2015). [Online; accessed 10-Feburary-2017].

[36] Slush Pool Team. 2017. Stratum Mining Protocol Official Documentation. https://slushpool.com/help/manual/stratum-protocol/. (2017).

[37] J. Wilhelm and T. Chiueh. 2007. A forced sampled execution approach to kernel rootkit identification.. In *Proc. of RAID'07*.

[38] Jun Xu, Z. Kalbarczyk, and R. K. Iyer. 2003. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings*. 260–269. https://doi.org/10.1109/RELDIS.2003.1238076

[39] Z. Xu, L. Chen, G. Gu, and C. Kruegel. 2012. PeerPress: Utilizing Enemies' P2P Strength against Them. In *Proc.of CCS'12*.

[40] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. 2014. GoldenEye: Efficiently and Effectively Unveiling Malware's Targeted Environment. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*.

[41] Yulong Zhang, Min Li, Kun Bai, Meng Yu, and Wanyu Zang. 2012. Incentive Compatible Moving Target Defense against VM-Colocation Attacks in Clouds. In *Information Security and Privacy Research*, Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou (Eds.). IFIP Advances in Information and Communication Technology, Vol. 376. Springer Berlin Heidelberg, 388–399. https://doi.org/10.1007/978-3-642-30436-1_32

[42] Quanyan Zhu and Tamer Başar. 2013. Game-theoretic approach to feedback-driven multi-stage moving target defense. In *Decision and Game Theory for Security*. Springer, 246–263.

A APPLICATION PROGRAM INTERFACES

Table 5 shows the complete list of the APIs we intercept and mark their output arguments as symbolic.

Table 5: The complete list of APIs Intercepted by *gExtractor*

Library	API	Library	API	Library	API
kernel32.dll	GetComputerNameA	kernel32.dll	GetComputerNameW	kernel32.dll	GetTimeZoneInformation
kernel32.dll	GetDiskFreeSpaceW	kernel32.dll	GetDiskFreeSpaceExW	kernel32.dll	GetSystemTime
kernel32.dll	GetTickCount	kernel32.dll	GetSystemTimeAsFileTime	kernel32.dll	GetFileAttributesExW
kernel32.dll	SearchPathW	kernel32.dll	GetSystemDirectoryW	kernel32.dll	SetFileTime
kernel32.dll	GetTempPathW	kernel32.dll	GetFileType	kernel32.dll	CreateDirectoryW
kernel32.dll	GetSystemDirectoryA	kernel32.dll	SetFileInformationByHandle	kernel32.dll	GetFileInformationByHandleEx
kernel32.dll	CopyFileW	kernel32.dll	SetFilePointer	kernel32.dll	CopyFileA
kernel32.dll	GetSystemWindowsDirectoryW	kernel32.dll	SetFilePointerEx	kernel32.dll	CopyFileExW
kernel32.dll	SetFileAttributesW	kernel32.dll	CreateDirectoryExW	kernel32.dll	GetFileSize
kernel32.dll	GetSystemWindowsDirectoryA	kernel32.dll	GetFileInformationByHandle	kernel32.dll	DeleteFileW
kernel32.dll	GetFileAttributesW	kernel32.dll	RemoveDirectoryW	kernel32.dll	FindFirstFileExA
kernel32.dll	MoveFileWithProgressW	kernel32.dll	SetEndOfFile	kernel32.dll	RemoveDirectoryA
kernel32.dll	FindFirstFileExW	kernel32.dll	GetFileSizeEx	kernel32.dll	GetSystemInfo
kernel32.dll	GetNativeSystemInfo	kernel32.dll	SetErrorMode	secur32.dll	GetUserNameExW
secur32.dll	GetUserNameExA	ntdll.dll	NtQueryAttributesFile	ntdll.dll	NtQueryFullAttributesFile
ntdll.dll	NtOpenFile	ntdll.dll	NtReadFile	ntdll.dll	NtWriteFile
ntdll.dll	NtQuerySystemInformation	ntdll.dll	NtQueryMultipleValueKey	version.dll	GetFileVersionInfoW
version.dll	GetFileVersionInfoExW	version.dll	GetFileVersionInfoSizeExW	version.dll	GetFileVersionInfoSizeW
user32.dll	GetSystemMetrics	user32.dll	RegisterHotKey	user32.dll	EnumWindows
user32.dll	FindWindowExA	user32.dll	GetForegroundWindow	user32.dll	LoadStringA
user32.dll	DrawTextExW	user32.dll	FindWindowW	user32.dll	LoadStringW
user32.dll	FindWindowExW	user32.dll	FindWindowA	user32.dll	DrawTextExA
user32.dll	GetAsyncKeyState	user32.dll	SetWindowsHookExA	user32.dll	SetWindowsHookExW
user32.dll	GetKeyboardState	user32.dll	GetKeyState	user32.dll	UnhookWindowsHookEx
crypt32.dll	CertControlStore	crypt32.dll	CertOpenSystemStoreA	crypt32.dll	CertOpenStore
crypt32.dll	CertCreateCertificateContext	crypt32.dll	CertOpenSystemStoreW	netapi32.dll	NetUserGetLocalGroups
netapi32.dll	NetUserGetInfo	netapi32.dll	NetShareEnum	advapi32.dll	GetUserNameW
advapi32.dll	GetUserNameA	advapi32.dll	LookupAccountSidW	advapi32.dll	EnumServicesStatusW
advapi32.dll	StartServiceW	advapi32.dll	OpenServiceA	advapi32.dll	CreateServiceA
advapi32.dll	OpenSCManagerW	advapi32.dll	OpenServiceW	advapi32.dll	ControlService
advapi32.dll	StartServiceA	advapi32.dll	DeleteService	advapi32.dll	OpenSCManagerA
advapi32.dll	CreateServiceW	advapi32.dll	EnumServicesStatusA	advapi32.dll	LookupPrivilegeValueW
advapi32.dll	RegCreateKeyExW	advapi32.dll	RegDeleteKeyA	advapi32.dll	RegEnumValueW
advapi32.dll	RegCloseKey	advapi32.dll	RegCreateKeyExA	advapi32.dll	RegSetValueExW
advapi32.dll	RegQueryInfoKeyW	advapi32.dll	RegQueryValueExA	advapi32.dll	RegEnumKeyExW
advapi32.dll	RegOpenKeyExW	advapi32.dll	RegSetValueExA	advapi32.dll	RegDeleteValueW
advapi32.dll	RegEnumValueA	advapi32.dll	RegEnumKeyW	advapi32.dll	RegDeleteKeyW
advapi32.dll	RegOpenKeyExA	advapi32.dll	RegDeleteValueA	advapi32.dll	RegEnumKeyExA
advapi32.dll	RegQueryInfoKeyA	advapi32.dll	RegQueryValueExW	srvcli.dll	NetShareEnum
wininet.dll	InternetGetConnectedState	wininet.dll	InternetReadFile	wininet.dll	InternetOpen
wininet.dll	InternetConnect	wininet.dll	HttpOpenRequest	wininet.dll	HttpSendRequest
wininet.dll	InternetQueryOption	wininet.dll	InternetSetOption	wininet.dll	HttpQueryInfo
wininet.dll	InternetQueryDataAvailable	ws2_32.dll	WSAStartup	ws2_32.dll	sendto
ws2_32.dll	recvfrom	ws2_32.dll	send	ws2_32.dll	bind
ws2_32.dll	select	ws2_32.dll	connect		