Chapter 11 Malware Deception with Automatic Analysis and Generation of HoneyResource

Zhaoyan Xu, Jialong Zhang, Zhiqiang Lin, and Guofei Gu

Abstract Malware often contains many system-resource-sensitive condition checks to avoid any duplicate infection, make sure to obtain required resources, or try to infect only targeted computers, etc. If we are able to extract the system resource constraints from malware binary code, and manipulate the environment state as *HoneyResource*, we would then be able to deceive malware for defense purpose, e.g., immunize a computer from infections, or trick malware into believing something. Towards this end, this chapter introduces our preliminary systematic study and a prototype system, AutoVac, for automatically extracting the system resource constraints from malware code and generating HoneyResource (e.g., malware vaccines) based on the system resource conditions.

Keywords Malware analysis · Malware immunization · Malware deception

11.1 Introduction

Malware is a severe threat to our computer systems. To combat malware, the state-of-the-art defense at end hosts mainly focuses on detection techniques, which often fall into two categories: signature-based detection and behavior-based detection. A signature-based approach typically attempts to extract some unique string patterns from malware binaries. Unfortunately, the signature generation and update speed usually cannot keep up with the quickly increasing malware samples each day

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

Z. Xu · J. Zhang · G. Gu (✉)
Texas A&M University, College Station, TX, USA
e-mail: guofei@cse.tamu.edu

Z. Lin
The Ohio State University, Columbus, OH, USA

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_11

in the wild due to the wide use of polymorphisms/packers in malware. While a behavior-based approach could be relatively more stable in terms of detecting the same set of malware and their variants, it is typically very expensive and may cause a noticeable performance overhead on end hosts.

Therefore, the need of new lightweight and complementary techniques for effective malware defense is still pressing. Interestingly, we find malware infection works similarly to pandemic diseases. If we were able to deceive the malware that it has infected the protected host, we would have been able to prevent it from infecting a wider range of machines (considering the case of botnets). Fortunately, we find malware that often contains system-resource-sensitive condition checks or constraints to avoid any duplicate infection, make sure to obtain required resources, or try to infect only targeted computers, etc. For instance, many fast-spreading malware programs (e.g., Conficker [26]) will clearly mark an infected machine as *infected* such that they can avoid wasting time and effort in re-infecting the machine. As such, such resource manipulating scheme can be considered as a more effective and safer way for malware deception. In this context, such resource is one kind of HoneyResource which tricks malware into believing the existence/non-existence of itself.

In general, any system resource/environment variables that are directly or indirectly used in path conditions (such as registry, mutex), or those that lead to the failure of certain system calls, can all be considered as HoneyResource, because these external environment states can impact the behavior of the malware. While it might lead to an over-approximation by considering all these state variables, we can run tests to eliminate the mistakenly classified environment variables.

Based on the above observation, we propose AUTOVAC, a new technique to automatically generate HoneyResource for effective and efficient malware deception. While theoretically manipulating any variable that leads to a conditional check of malware execution could potentially be used as a HoneyResource, we would like to focus on the variables whose states can be controlled by the external environment such as registry, certain file names, etc. As such, the environment resources accessed by malware are of our interest. Specifically, we design a program analysis technique to determine whether the manipulation of these resources can successfully prevent malware's infection/execution. We treat such resources as our malware HoneyResource and derive concrete information needed for generating HoneyResource. After we generate the HoneyResource, we then inject them into end hosts. For example, HoneyResource is able to serve as a kind of *vaccine* for malware. To the best of our knowledge, AUTOVAC is the first systematic work of using program analysis to automatically generate HoneyResource for real-world malware deception.

In summary, we make the following contributions:

- We conduct the first systematic study of malware HoneyResource. We discuss all possible mutable resources of our interest and present a taxonomy of malware HoneyResource.

- We design and implement AUTOVAC, which can automatically track the malware path constraints as well as their propagation, associate them with the external environment resources, and automatically generate HoneyResource.
- We evaluate our system with a large set of real-world malware samples. Experimental results show that it is truly possible to generate working HoneyResource for many real-world malware families, such as Conficker, Sality, and Zeus, and use HoneyResource as a complementary approach in practice.

11.2 Problem Statement and Approach Overview

11.2.1 Malware HoneyResource Background

11.2.1.1 Definition of Malware HoneyResource

From our viewpoint, a malware HoneyResource is a computational preparation that deceives a particular malware program, e.g., trick it into believing something, or prevent its infection. Essentially, malware, like any generic program, usually conducts a series of operations on system resources and outputs the computation result.

Thus, we define a malware HoneyResource as a specific system resource (or a collection of them) that is created or used by malware in order for its normal infection and execution. Such malware HoneyResource typically has two kinds of behavior:

- It simulates the existence of certain computer organism (system environment/resource) such that malware will perform certain activities, e.g., exit upon the awareness of such existence (because it does not want to re-infect the victim again, or the victim does not have a targeted environment, etc.).
- It prevents malware from creating/accessing certain critical computer organism such that malware cannot obtain its essential resources to fulfill the functions.

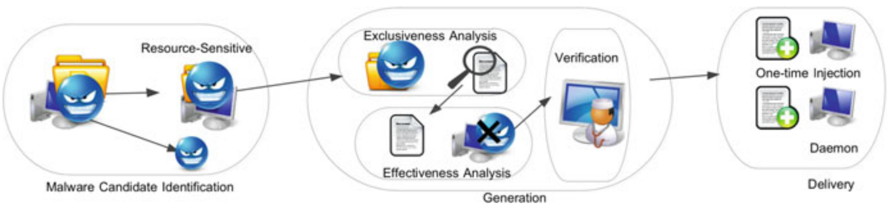

Fig. 11.1 System architecture

11.2.1.2 A Taxonomy of Malware HoneyResource

Besides the aforementioned categories of malware HoneyResource, we can further define different types from different perspectives.

First, from the perspective of identification, the HoneyResource identifier is defined as a combination of *resource type* and *name* of malware-targeted resources. To avoid unwanted side effect to benign software running on end host, the HoneyResource identifier should be as *unique* and *deterministic* as possible. Thus, in our taxonomy, an identifier can be categorized as: static (e.g., constant value), partial static (e.g., it conforms to a specific regular expression), or algorithm-deterministic (e.g., it is calculated with customized algorithms).

To deceive different malware families, the effectiveness of a malware HoneyResource can vary. Based on the effectiveness, we can classify malware HoneyResource into two types: full deception that can completely cease the malware execution (e.g., negating the first few condition checks to prevent any malicious behavior execution), and partial deception that significantly affects the execution of some major functions in malware (e.g., malware is not able to keep persistent in the system if rebooted, or malware is not able to perform key network communication such as C&C, and self-updating).

In terms of HoneyResource delivery and deployment, there could be two categories: direct injection and creation of HoneyResource daemon. Direct injection is very lightweight, e.g., a specific `mutex` name or file name, and the HoneyResource can be simply injected into the target computer once and it will be effective afterwards. HoneyResource daemon requires running a service program (i.e., a daemon) on the targeted machine, and such daemon can prevent the creation (or other access types) of certain specific files, registries, libraries, system services, windows, and processes to further prevent malware from obtaining critical resources or information to fulfill its functionalities (such as for partial deception). More details are presented in Sect. 11.5.

It is worth noting that an ideal malware HoneyResource is those with full deception and one-time direct injection. However, other types of HoneyResource are also useful, as discussed later and shown in our evaluation (Sect. 11.6).

11.2.1.3 Use Case of HoneyResource

As a complementary technique to existing malware defense, HoneyResource may not be used to protect machines from all malware attacks. However, they can be used for current, high-profile, large-scale malware propagation and infections, which may last for a period of time, e.g., several days, weeks, or months. If we can capture the binary at the initial infection stage, we can quickly generate HoneyResource and protect our uninfected machines from the attacks, until a better detection or prevention solutions (e.g., a system/software patch to fix the vulnerability) are available and fully deployed.

11.2.1.4 Target and Assumptions

Not all malware can have HoneyResource. Our target is those malware that has specific system-resource-sensitive behavior, illustrated in the following scenarios:

- Some malware can work only in the scenario in which none of the same malware instances is present in the host. Thus, they have to uniquely *mark* their infected systems through creating and checking certain deterministic identifiers such as mutex, file, as shown in the Conficker example. Our HoneyResource can hence appear to be the malware vaccine to fool the malware and stop its infection.
- Some malware has issues in handling the failure of certain system resource access. Our HoneyResource can try to enforce such failures to make the malware run into their undesired status (e.g., process termination, or important functions being disabled).
- Some targeted malware is designed to work in a specific system environment. Our HoneyResource can attempt to make each protected system different from malware targeted environment, so as to protect machines from the infection.

It is true that some malware may not use system resource checks to make their infection decision. That is, AUTOVAC does have limitations and we discuss in great detail on the possible evasions in Sect. 11.7. We note that while evasions are possible, most of these scenarios are not within the scope and assumptions of our approach. The intention of AUTOVAC is not to replace existing defense approaches, but to complement them from a new perspective. As we show later, once we can successfully extract interested system resource constraints and generate HoneyResource, we can effectively and efficiently deceive malware.

11.2.2 Approach Overview

An overview of AUTOVAC is illustrated in Fig. 11.1. At a high level, it consists of three phases: *Candidate Selection*, *HoneyResource Generation*, and *HoneyResource Delivery/Deployment*.

In **Phase-I** (Sect. 11.3), we will first filter out malware samples that are unlikely to contain HoneyResource. In this step, we profile the normal execution of the malware to obtain an overview of the malware's accessed system resources, including the types of resources and the names of the corresponding resource-identifiers, the operations (e.g., create, and read/write) on the resources, and the corresponding results (e.g., succeed, or fail).

During our profiling, we will also apply a variant of dynamic taint analysis [7] to determine whether the malware's execution will be affected by certain resources it has accessed. The implication is that malware has to be *sensitive* to its resource access result. Otherwise, malware's behavior is deterministic regardless of its resource environment and no HoneyResource willexist for it. Hence, if we find no program branches that depend on any system resource, we filter this malware because it does not contain HoneyResource that we can extract. At the end of this phase, we obtain a list of candidate resources that can affect the control flow of the malware execution.

In **Phase-II** (Sect. 11.4), our task is to generate HoneyResource by testing their exclusiveness and impact on malware execution. It contains three sub-steps.

- **Step-I: Exclusiveness Analysis** In general, system resources are also being used by benign programs. In this step, we would like to *filter the resource identifiers that are not exclusive to malware itself (e.g., some benign programs also use them), in order to avoid false positives*.
- **Step-II: Impact Analysis** The goal of this step is to *measure the potential impact of a certain system resource, i.e., whether it can affect the execution of some interested malware functions*. We start a second-round execution monitoring by manipulating the result of the specific malware's resource operation, which will generate a manipulated trace. We apply program alignment techniques [8] to compare the execution differences between the manipulated trace and the normal trace and determine if the system resource can (significantly) impact the malware functions, e.g., cause malware to stop the execution. At the end of this step, we generate a list of resources that can effectively stop the malware's infection (full deception), or significantly affect the malware's certain functions (partial deception).
- **Step-III: Determinism Analysis** We also have to measure the determinism of the specific system resource identifier, e.g., *filename* or `mutex` name. An effective malware HoneyResource should be *deterministic, such that it can be accurately reproduced/predicted to affect the targeted malware*. A deterministic value could be a fixed/static value, or a value that is generated from a deterministic algorithm (from deterministic resources) or even partial static if certain part is deterministic. To decide if a specific resource identifier is deterministic, we perform *backward taint analysis and program slicing* to fully understand the identifier generation logic and the parameters it depends on. Based on that, we further analyze the root-cause of the identifier generation and generate a program slice responsible for the identifier generation logic.

In **Phase-III**, we deploy the malware HoneyResource at an end host. There are also two situations: direct injection and HoneyResource daemon. We will present their details in Sect. 11.5.

11.3 Phase-I: Candidate Selection

Given a malware sample, AutoVac will first determine whether it is possible to generate a HoneyResource, and at the same time collect the behavior information to facilitate the next step analysis. Since our HoneyResourceis essentially composed of system resources that have a direct or indirect (through propagation) impact on the malware execution, we adopt a variant of dynamic taint analysis [7] to achieve this.

11.3.1 Taint Sources

Taint sources define the origins of the tainted data. Our current focus is on those system-resource-related data that can possibly impact the malware behavior. However, there is a wide range of system resources and certainly some of them cannot be used such as system-assigned random objects. As such, we have to systematically study these resources and identify our taint source. In particular, we use the following criteria to decide whether a system resource should be tainted.

- **Unique Presence** Our focused system resources should be commonly used by malware, and these resources should be *uniquely identified*. Thus, those *transient* system resources, e.g., events, signals, and critical sections, are out of our interest.
- **Less Impact to Benign Software** Our targeted resources should have *little or minor impact* to benign programs. This requirement would exclude many system-wide objects and information, such as timers, performance counters, input/output devices, and removable devices, because they are commonly accessed by benign programs
- **Easier Deployment** Our targeted resources should be lightly deployed onto end hosts. To this end, injecting some specific files or `mutex` into the end host would be viable options. Therefore, files, `mutex`, or registry will be our main targeted resources.

11.3.1.1 API Labeling

After applying the above criteria, eventually `mutex`, static files, and registry items are of our particular interest. Meanwhile, the propagation use of these resources such as process, library, GUI window, and services are also of our interest because these resources depend on some deterministic resource identifiers. However, at the instruction level, these *resource-identifiers* often get accessed through system APIs. Thus, we have to examine each Windows API to define our taint sources.

More specifically, all the system resource access APIs (e.g., `NtQueryObject`) are of our interest. AUTOVAC will taint the return values as well as the affected arguments of these functions. In our design, we examined over 800 windows APIs and we classified them into the following two categories:

- **Tainting the return value** Most APIs only affect the return values (always stored in `EAX`), such as `OpenMutex`, and `NtSaveKey`. For them, we just taint the return value.

- **Tainting the argument** Some APIs store the affected values in the arguments. For instance, `NtOpenKey` and `NtOpenFile` store the return handler in their first parameters.

Besides tainting the return values or arguments, we also need to record the concrete values of the arguments to these APIs because eventually our HoneyResource work by affecting the system environments which are their arguments. Meanwhile, not all the arguments are of our interest, and only those *resource-identifiers*. This is also a tedious procedure to identify these *resource-identifiers*. Table 11.1 shows an example on how we label the two Windows APIs.

Table 11.1 Labeling examples for OpenMutex/ReadFile

	`OpenMutex`	`ReadFile`
Resource type	Mutex	File
Resource-identifier	3rd parameter: *lpName*	1st parameter: *hFile for Handle Map*
Success	`EAX`: Valid handle value	`EAX`: TRUE
Failure	`EAX`: NULL, `GetLastError: 0x02`	`EAX`: FALSE `GetLastError: 0x1E`

11.3.2 *Taint Propagation*

AutoVac has to propagate taint labels for data operations. That is, for any instruction whose source operand has been associated with the tainted labels, we taint the destination operand with the same label. Then, whenever we find a comparison (i.e., predicate) instruction whose operands have been tainted (e.g., `test,cmp`), we will flag this malware most likely having a HoneyResource and pass it to our next phase analysis.

11.3.2.1 Output from Phase-I

As our Phase-I runs the malware in normal settings, it provides a great opportunity to collect the normal malware behavior. To this end, we log all the executed APIs as well as their parameters, along with the precise calling context information including the call stack and the caller-PC (program counter). In addition, our log file also contains the list of the system-resource-sensitive APIs that have been executed, and their propagated taint record that is used in the predicate.

11.4 Phase-II: HoneyResource Generation

Once a malware sample has been flagged to “possibly have a HoneyResource” in **Phase-I**, it will be fed to our **Phase-II** to perform a deeper analysis, including exclusiveness analysis (Sect. 11.4.1), impact analysis (Sect. 11.4.2), and determinism analysis (Sect. 11.4.3). In this section, we present these analyses in greater detail.

11.4.1 Exclusiveness Analysis

The goal of our exclusiveness analysis is to exclude the resources that have been used in benign software. For instance, some resources such as library names `uxtheme.dll`, and `mscrt.dll` could be used in benign programs. We must exclude them otherwise our HoneyResource will have false positives.

In **Phase-I**, AUTOVAC has logged all the *resource-identifiers*, and next we would like to query whether or not each *identifier* is unique to the malware. Our basic idea is inspired by a Googling approach used in the previous studies [27]. Essentially, we use Google query APIs to search *resource-identifiers*. Based on the return results and their context, we infer whether these resources are already associated with benign software. We refer the readers to [27] for more details. In short, from our search query, if the *resource-identifiers* does not conflict with benign software or there is no any matching search result, then we proceed with further analysis.

11.4.2 Impact Analysis

Given a list of the system resources that can (in)directly affect the malware execution and the corresponding APIs provided in **Phase-I**, AUTOVAC will run the malware again in a controlled environment such that we can mutate the return value or involved arguments and test whether malware will exhibit different behavior or not. Our current design is to mutate each involved API one at a time and compare the behavior with our normal execution captured in **Phase-I**.

11.4.2.1 Trace Differential Analysis

Then, the next question is how we compare the malware behavior in two traces: one is a normal execution, and the other is a resource mutated execution.

Finding the differences in two traces has been discussed in the previous literature (e.g., [8, 25]). It is essentially a *program alignment* problem [8]. The basic idea is to align two execution points that are equivalent to each other and then compute the differences only between the *unaligned* instructions. In our scenario, we try to obtain the high-level information such as whether the malware will terminate rather than the minor instruction-level execution differences. Thus, in our design, we use the API call sequences (as we have already logged all the executed APIs and their calling context information) and present an API sequence alignment algorithm as shown in Algorithm 2.

In particular, we adopted an alignment algorithm from Zeller [8], which uses the *execution context* for each instruction for the comparison. If the instruction and its execution context are equivalent (line 4), they are aligned together. However, we do not need to compare instruction by instruction, but rather at the granularity of APIs. Thus, we define a calling execution context as a *triple*:

Algorithm 2: Differential Analysis on the API-Call Traces

```
   $\prod_m$: Manipulated Call Trace, $\prod_n$: Natural Call Trace
   $\Delta_m$: Unaligned Call Trace in $\prod_m$, $\Delta_n$: Unaligned Call Trace in $\prod_n$,
   $f_{\prod}$: $\langle$ name, caller eip, parameter list$\rangle$, $f_{\Delta}$: $\langle$ name, parameter list$\rangle$
1  $\Delta_m \leftarrow \emptyset, \Delta_n \leftarrow \emptyset$
2  for call $f_{\prod_m}$ in $\prod_m$ do
3      for call $f_{\prod_n}$ in $\prod_m$ do
4          if isAligned($f_{\prod_m}$, $f_{\prod_n}$) then
5              GOTO FIND_ALIGNED
6      $\Delta_m = \Delta_m \bigcup f_{\Delta_m}$
7  $\Delta_n = \prod_n$
8  FIND_ALIGNED:
9      $\Delta_n = \prod_m[0, \text{index}(f_{\prod_n})]$
10     $\{f_{\Delta_i}\}$ = Diff($\Delta_m$, $\Delta_n$)
11     return $\{f_{\Delta_i}\}$
```

<API-name, Caller-PC, Parameter list>

For the *parameter list*, we only compare the *static* parameters that are *identical* across different executions. Note that all these information has been logged either in **Phase-I** for the normal execution, or logged in **Phase-II** for the mutated execution. Also, the reason we have to log the *Caller-PC* is for the preciseness.

As illustrated in Algorithm 2, our analysis begins from the start of the trace, then proceeds with a linear searching for each system/library call in the mutated trace and examines whether it could be aligned with some call in the normal run trace (line 2 – 8). If we find an anchor point, we generate two difference sets Δ_m and Δ_n.

Next, we examine the two Δ sets to evaluate the further differences and classify the HoneyResource type. Specifically, we define three kinds of deception effects.

11.4.2.2 Full Deception

If we find APIs such as `ExitThread`, `TerminateProcess`, and `Terminate Thread` in Δ, then certainly the mutated system resources can be served as a full deception HoneyResource, because the malware has killed itself.

11.4.2.3 Partial Deception

Some HoneyResource may significantly weaken certain important functions of malware. We consider them as partial deception. More specifically, we currently focus on the following four types of partial deception:

- **Type-I: Disable Kernel Injection** An important malicious function of malware is to raise its privilege. The common way they use is to inject a kernel driver into an end host. There are several system calls (mainly undocumented), such as `OpenSCManager` have been used for this. Furthermore, some malware commonly copies itself as a new file with its name ending with `.sys`, which implies that some kernel driver is created by the malware.
- **Type-II: Disable Massive Network Behavior** If we find that the normal execution is full of network-related functions, while the manipulated execution is clean from such calls, we consider such deception as **Type-II** Partial Deception.
- **Type-III: Disable Malware Persistence** Malware typically modifies specific registry entries such as `Run` subkeys in multiple register paths. Other autostart approaches include: (a) file operations on `startup` folder or `system.ini` files, (b) creation of new service entries, and (c) access of `winlogon` binary. Through differential analysis, we can tell if these operations are lost in the mutated execution while present in the normal execution.
- **Type-IV: Disable Benign Process Injection** To be more evasive, malware often inject themselves into some benign processes. Processes such as `explorer.exe` and `svchost.exe` are common targets. If we find such a clear pattern in the differential analysis, we consider these HoneyResource as **Type-IV** partial deception.

11.4.2.4 No Deception

If none of the above APIs are in the Δ, then we classify this HoneyResource with no effect to stop or affect malware behavior.

11.4.3 Determinism Analysis

We next need to verify the determinism of the extracted resource-identifiers.

11.4.3.1 Backward Taint Tracking and Program Slicing

Given a resource-identifier, we need to identify whether it is deterministic or entirely random. We choose to trace the root-cause for the generation of the resource-identifier.

To back track the procedure of how malware generates an identifier, we perform a backward taint tracking. The basic idea is to include all the instructions that have contributed to the creation of the resource-identifier, which is the argument of the API of our interest. To this end, starting from data-use of the argument, we back track each executed instruction to check whether or not their operands have been involved to define the data. If so, we taint the source operand as the same symbol and continue the backward propagation. We perform the analysis offline on logged traces.

The termination of our backward tracking is the point to identify the root-cause that generates the identifier's name. We continue backward propagation until tainted source is either from read-only regions (e.g., static strings), or constant values, or the return value of the system APIs. Based on these different sources, we decide whether the generation of the identifier is deterministic or not.

An identifier has a *non-deterministic* type if and only if *all* elements of its composition are resulted from some random functions (e.g., `GetPerformanceCounter` and `GetTempFileName`). As illustrated in the left part of Fig 11.2, if the termination data point is from a read-only segment such as `.rdata`, or constant values, we can easily mark it as *static*. Similarly, if an identifier is constructed using some non-deterministic value combined with some constant value, we can mark it as *partial static*, and such an identifier will be deployed using a slightly different strategy compared to the scenario of purely static identifier.

An identifier could be *algorithm-deterministic*, namely its identifier is generated through certain computation. Some appear-to-be random name can be generated from some invariable seed, such as computer name or hardware serial number. Algorithm-deterministic names will be backward propagated to some semantic-known APIs. We use these APIs to decide the root-cause type when generating the name. One example is shown in the middle part of Fig. 11.2. We use the `GetComputerName` to infer that the input should be a computer name.

For such algorithm-deterministic identifier, we also need to find the generation logic because we need to replay and compute it for each end host. We apply the existing backward program slicing[18] techniques to extract an independent, executable program slice for that. At the end of this step, we delete all the entirely random (non-deterministic) identifiers.

11.4.4 Malware Clinic Test

To further reduce the possible false positives, we design a *Malware Clinic Test* at the end of this phase. Malware Clinic Test aims to inject our HoneyResource into real environments and test whether it will affect the normal use of a computer system. This test environment is automatically configured by running multiple benign software and services. Even though the scheme of clinic test is simple, it is essential to ensure the quality of our generated HoneyResource. If it affects the normal usage, it will be discarded.

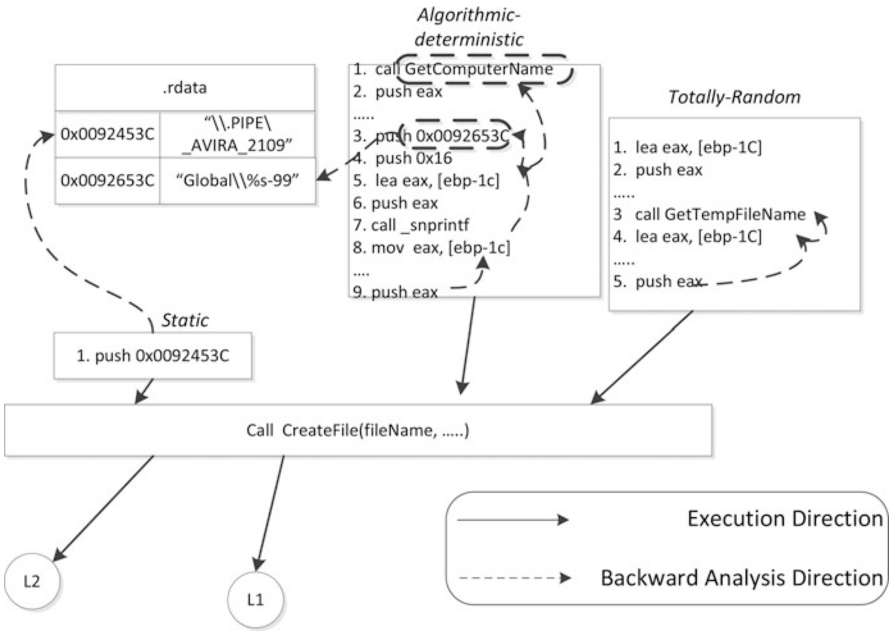

Fig. 11.2 Sample Malware code and the traced behavior

11.5 Phase-III: HoneyResource Delivery and Deployment

After we generate the HoneyResource, we next describe how to deliver and deploy the HoneyResource to an end-user computer.

11.5.1 Direct Injection

Direct injection works for static identifiers. If a HoneyResource stops malware execution by frustrating the presence checking of static type of resources, we inject it by creating or deleting the resources. For instance, if the malware needs to open certain static file (or registry) before proceeding the malicious functionality, then we remove the static file (or registry), or vice versa. Moreover, we accordingly adjust the injected file's access privilege to disallow certain operation such as read and write. In these cases, when a low-privilege malware program attempts to access a resource, which is a common case at the initial infection stage, static HoneyResource efficiently stop further malicious behavior.

11.5.2 HoneyResource Daemon

Daemon works for algorithm-deterministic identifier and partially static identifier. For an algorithm-deterministic identifier, we have extracted a program slice of the resource-identifier generation logic with knowledge about its input, such as a computer name or an IP address. To generate the HoneyResource, we collect these information ahead and run the captured program slice. Such procedure works very similar to Inspector Gadget [18]. Our daemon process runs periodically to check whether the input has been changed and the HoneyResource needs to be re-generated.

Daemon is also designed for identifying resource name represented using regular expressions (i.e., distinguishable partial static HoneyResource). Specifically, at the end host, we dynamically intercept the APIs and resolve their resource-identifiers. If the daemon monitors that a resource identifier matches with our partial static HoneyResource, it will return the predefined result to stop the malware execution.

11.6 Evaluation

We have implemented AutoVac. While our online dynamic analysis can be implemented using virtual machine monitors such as TEMU [4], we use DynamoRIO [2] to implement due to its simplicity and flexibility in binary instrumentation. Our differential analysis module is implemented using offline parsing of the execution logs. Also, to perform tainted analysis, we translate the X86 instructions into an intermediate language *BIL* [10], and then we develop our own parser code to identify the resource-sensitive branches and perform differential analysis. Our exclusiveness analysis involves a search engine query component, for which we implement using the API provided by Google. In this section, we present our evaluation results.

11.6.1 Experiment Dataset

Our test dataset consists of 1,716 malware samples, which are collected from multiple online malware repositories (e.g., [1, 3]) with mostly from Anubis [1].

We also leverage an online malware classification tool, *VirusTotal* [5], to obtain the classification information for these malware. We summarize classification results in Table 11.2. We can see that these malware samples fall into 6 categories such as Backdoor (722 samples), Downloader (574 samples), and Trojan (184 samples).

Table 11.2 Malware's classification from VirusTotal

Category	# Malware	Percentage
Trojan	184	10.72%
Backdoor	722	42.07%
Downloader	574	33.44%
Adware	73	4.25%
Worm	104	6.06%
Virus	59	3.43%
Total	1,716	100%

11.6.2 Evaluation Result on Candidate Selection

In the first step (**Phase-I**), we monitor malware's access to system resources. We conduct this experiment by running these 1,716 malware samples in our analysis environment and each sample runs for 1 min (we tend to believe that the resource checks usually happen in the early stage of the malware execution and we thus choose this 1-min threshold). We hook 89 system/library calls as tainted sources that are related to resource operations. The resources in our evaluation include *file*, *mutex*, *registry*, *window*, *process*, *library*, and *service*. We measure the basic operations for these resources such as read/write for file and registry, and open/create for other resources. Meanwhile, for each execution instance of the hooked function, we examine their callers' `PC` and make sure that it does not belong to the system library's address space. Thus, we do not count the functions that are called inside the system/library calls.

For 1,716 malware samples, we successfully tracked 460,323 occurrences of these API calls. Through our taint analysis in this phase, we identified that 371,015(80.3%) occurrences of the calls will possibly deviate the execution of the malware samples. This result confirms that real-world malware is indeed resource sensitive.

Among these 371,015 occurrences, we further made a statistic study based on the resource type and its corresponding operations. The result is shown in Fig. 11.3. From the figure, we can see that around 37.39% of the resource accesses account for file operation. Mutex (7.07%) and registry (20.08%) are also commonly accessed by malware. We consider these three types of resources that can be efficiently delivered using the injection scheme. Meanwhile, malware's logic is also commonly sensitive to other types of resources such as windows (13.14%), process (8.02%), library (6.6%), and service (3.4%).

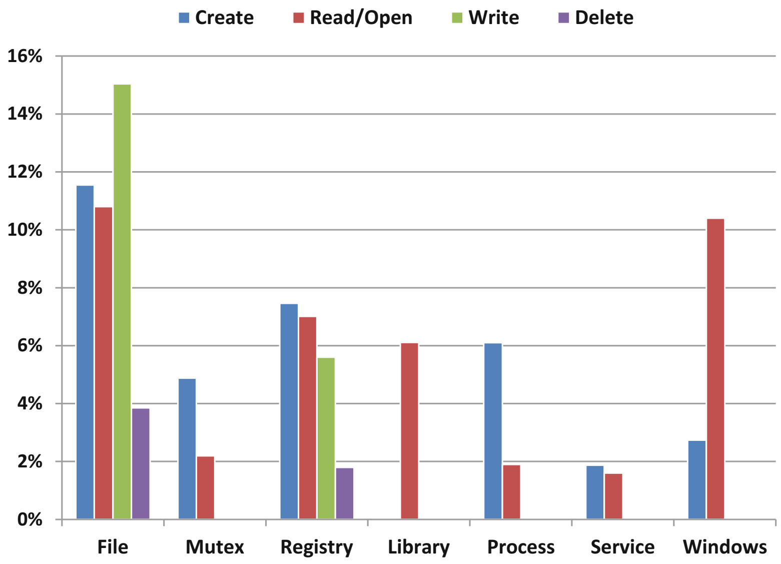

Fig. 11.3 Statistics on Malware's resource-sensitive behaviors

11.6.3 Evaluation on HoneyResource Generation

Table 11.3 HoneyResource samples (Operation type symbols—check Existence (E), Create (C), Read (R), and Write (W), Impact symbol—Termination (T), Process Hijacking (H), Persistence (P), Kernel Injection (K), and Network Massive Attack (N))

Seq	Type	OperType	Impact	Identifier	Malicious sample Md5
1	Mutex	E	T	!VoqA.I4	df1df624c5da833d3882d22a2e2456c9
2	File	C,R,W	P,H	%system32% \twinrsdi.exe	1b6fb589f36654af0ef44aa92f94324a
3	File	C,E,R,	P,H,N	%system32% \dwdsregt.exe	24784256bbbb936dc1e0999c307883c8
4	File	C,E,R,W	K,P	%system32% \driver\qatpcks.sys	27d18e20e253391112d50b2b49440aea
5	Mutex	E	T	GTSKISNAUOI	ee5878eab962b032c78c1d6eec7ec917
6	Mutex	E	P,H	fx221	af48ecfcc1812d6f814a26792107b80e
7	Mutex	C,E	T	)ryt-24qtqq26sn]9c	b534b75da5fc3b9b178c60bf10b1feca
8	Mutex	C,E,R	P,H	_AVIRA_2109	04a93b1f08a1675c67c9975a7024c3d6
9	File	C,E,R,W	P,H	%system32% \ shlmon.exe	af48ecfcc1812d6f814a26792107b80e
10	File	C,E,R,W	T,P	%system32%\ sdra64.exe	04a93b1f08a1675c67c9975a7024c3d6

In the evaluation, we analyzed all 1, 716 malware in a controlled environment. In total, we generated 536 HoneyResource that belong to 210 malware samples. The result is presented in Table 11.4. For each column, we classify the HoneyResource as full deception or partial deception (Type-I to Type-IV). We also list the statistics on the HoneyResource distribution among different resource types in Table 11.4. Among all HoneyResource, we find that 373 HoneyResource have static identifiers, and 163 samples have *algorithm-deterministic* or *partial static* identifiers.

Table 11.4 Evaluation on HoneyResource generation

Resource	Full	Type-I	Type-II	Type-III	Type-IV	All
File	31	19	17	110	61	238
Registry	10	11	3	72	19	115
Mutex	5	3	3	16	3	30
Process	2	5	2	18	5	32
Windows	0	4	3	8	3	18
Library	19	5	1	10	19	54
Service	7	4	0	17	21	49
Total	74	51	29	251	131	536

Table 11.5 HoneyResource statistics on different Malware families

Type	Backdoor	Trojan	Worm	Adware	Downloader	Virus
File	33%	27%	24%	30%	45%	81%
Registry	15%	29%	21%	13%	20%	19%
Windows	3%	14%	0%	47%	11%	0%
Mutex	8%	12%	29%	0%	2%	0%
Process	8%	7%	14%	0%	10%	0%
Library	26%	9%	4%	0%	7%	0%
Service	7%	2%	8%	10%	5%	0%
Deployment						
Direct	67%	79%	63%	69%	69%	84%
Daemon	33%	21%	37%	31%	31%	16%

To zoom-in the details of these HoneyResource, we select 10 representative samples and describe them in Table 11.3. We can see that most of these HoneyResource stop several logic of malware's infections. In some cases, different operations on the resources can even cause different effects on malware's logic. For example, for the last malware in Table 11.3, we find that the failure of *creating* a file will stop malware's process hijacking logic, and the failure of *writing* a file will crash the malware process (Table 11.4).

For the generated 536 HoneyResource, we also combined their types with the 210 malware's classification information to see what is the common HoneyResource type for different kinds of malware. The result is shown in Table 11.5. From this table, we can see that the file resources are the common HoneyResource for many malware families. Meanwhile, the windows resource HoneyResource is better suitable for adware because the windows resource HoneyResource is attempting to prevent adware from creating their malicious windows. If such operations fail, adware will possibly stop their further action. Last but not least, mutex HoneyResource works better for worm and backdoor malware. This is also reasonable, because these malware highly depend on the mutex to prevent duplicate infection.

We also report the statistics of our delivery for these 536 HoneyResource. As shown in Table 11.5, direct injection is the most common way to deploy HoneyResource on end hosts. Also, only about 20%–30% HoneyResource need a daemon for the deployment.

11.6.4 Case Studies

Next, we present two representative case studies to illustrate in greater detail on how each of our resource access-based HoneyResource can be used for malware infection immunization. In such case, our HoneyResource can work as malware HoneyResource to stop malware infection.

11.6.4.1 File-Based HoneyResource

One HoneyResource for Zeus/Zbot [6] family is a static file named `sdra64.exe` which is stored in the `system32` directory. We observe that if Zeus successfully creates this file, it will continue writing malicious bytes into that file using bytes in its resource and start a new process using this file.

Delivery: We deliver a HoneyResource by deliberately creating `sdra64.exe` at an end host. This file is owned by a super user and does not allow any creation operation by others. In this way, our HoneyResource prevents Zeus's attempt to start the malicious process.

11.6.4.2 Mutex-Based HoneyResource

One mutex HoneyResource is for Conficker, which is an algorithm-deterministic HoneyResource. This mutex HoneyResource can efficiently stop Conficker's infection at its initialization stage.

Several other mutex examples include `_AVIRA_21099`, `_AVIRA_2109`, `_AVIRA_2108`, which belong to Zeus/Zbot[6] malware. This set of HoneyResource can stop multiple malware logic such as kernel injection, process hijacking, and network communication.

Delivery: Direct injection is an efficient approach to deliver mutex HoneyResource. We simply create a deterministic `_AVIRA_` mutex in the system to prevent

Zbot's injection. For Conficker, we run the HoneyResource slice once at the end host and generate the mutex name for each computer.

11.6.5 HoneyResource Effect Analysis

In this test, we evaluate the effect of our HoneyResource on the malware samples. As reported in Sect. 11.6.3, our HoneyResource can stop or weaken 210 samples' malicious behaviors. In this test, we run these 210 samples in both deployed environment and the normal infection environment for 5 min. Then, we compare the differences of their native system calls (all the NT native calls) in these two environments. We define a metric Behavior Decreasing Ratio, $BDR = \frac{N_n - N_d}{N_n}$, where N_n is the number native system calls in the normal environment, while N_d is that number in the deployed environment. The larger BDR is, the more reduction of functions by the HoneyResource. In Fig. 11.4, we report the distribution of BDR according to different effectiveness type.

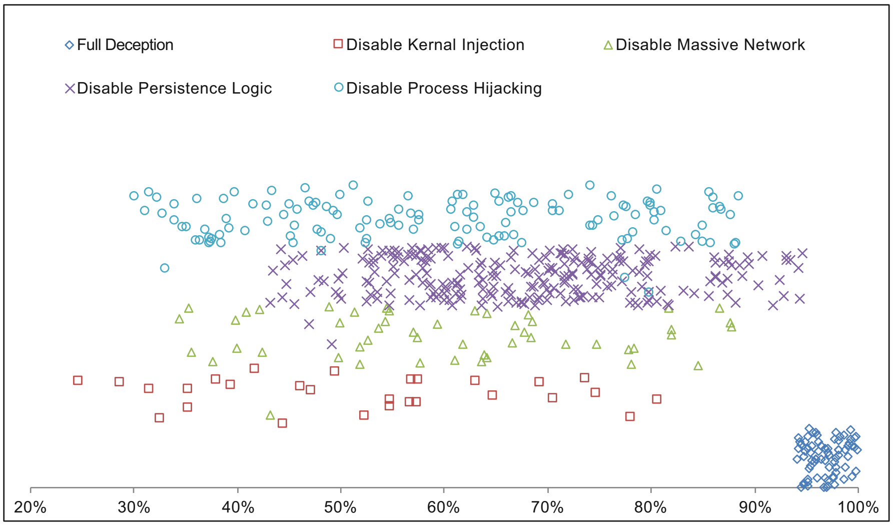

Fig. 11.4 Distribution of BDR

From this figure, we can see that the full deception HoneyResource are obviously the most effective ones and they all terminate the execution of malware (the reason why their BDR is not 100% is simply because of their initial executions before exit that also have some native system calls). Our partial deception HoneyResource all effectively achieve their goals by disabling key functions in the malware (through a careful manual examination, we confirm that all unwanted malicious logic has been disabled). One such example for Zeus is shown in Table 11.6. Even in the *worst* case

Table 11.6 Example of a high-profile Malware HoneyResource

Malware	HoneyResource	Type	Impact description
Zeus/Zbot	_AVIRA_2109	Mutex	Stop process hijacking

in terms of BDR, our partial deception HoneyResource can still reduce at least 24% malware's *important* system call activities. Note that BDR will certainly increase if we keep running the malware sample in a longer time period.

To further verify that our HoneyResource are effective for different variants in the same malware family, we choose 6 high-profile malware samples and perform another test. These samples are high-profile malware such as Conficker, Zeus/Zbot, and Sality, and for these 6 samples we have extracted a total of 17 different HoneyResource in our previous test. We then further collect 5 variants (binaries are different from what we have collected in the original dataset) belonging to each family (thus 30 new variants in total). Then, we run the 30 newly collected variants in both normal and deployed environments, similar to the previous experiment. We carefully analyze the execution differences and manually verify that whether the injected HoneyResource have achieved the goal or not. The result is showed in Table 11.7. Note that the 4th column indicates the number of malicious functions that can be stopped if ideally these HoneyResource work for all variants, the 5th column indicates the actual number from our test, and the 6th column shows the percentage of success.

From the result, we can see that overall our HoneyResource can take effect in almost all variants. However, we do find that some HoneyResource can work for some variants but fail on others. One example is the file HoneyResource `sdra64.exe` which we did not find its use in 2 other Zbot variants. Fortunately, for each malware, we have extracted more than one HoneyResource. Thus, even some may not be effective for all variants, the combination of these HoneyResource can still achieve satisfiable results. We believe that this test also highlights the importance of using an automatic tool (such as our AUTOVAC) to analyze malware samples to extract as many HoneyResource as possible, a goal otherwise very hard to achieve through manual analysis.

Table 11.7 Effectiveness evaluation on Malware variants

Malware	HoneyResource#	Type	Ideal case	Verified	Ratio
Zeus/Zbot	6	Mutex, file	30	23	77%
Conficker	2	Mutex	10	10	100%
Qakbot	2	Registry	10	10	100%
IBank	1	File	5	5	100%
Sality	3	Mutex, file	15	12	80%
PosionIvy	3	Mutex, file	15	10	67%
Total	17		85	70	82%

11.6.5.1 False Positive Test

Our next test is on the false positive evaluation, i.e., whether our generated HoneyResource will affect the normal program executions. We design a simple malware clinic test as mentioned in Sect. 11.4.4.

First, we install 5 different virtual machines running over 40 benign software (which includes the most common software typically seen on normal users' computers such as all kinds of browsers, programming environments, multimedia applications, Office toolkits, IM and social networking tools, anti-virus tools, and P2P programs). Then, we equally inject our HoneyResource into each test machine and monitor their system logs over a period of a week. The result shows that our HoneyResource did not cause any problem to our running environments.

One could argue that this automatic test may underestimate users' interaction. Hence, we conduct another test to install 200 HoneyResource on 4 lab machines. All these four machines are for normal everyday use. The result also shows that our generated HoneyResource did not cause any trouble for the operation of existing benign programs. While our clinic test could have a limited scope, we believe that a well-designed clinic test is still helpful to refine our automatically generated HoneyResource in a real-world scenario.

11.6.6 Performance Overhead

11.6.6.1 HoneyResource Generation Overhead

First, we measured the overhead of the automatic extraction. We run our test on machines with Intel Core i5 CPU and 6GB memory.

- **Generating the HoneyResource** In our test, we measure the time spent on analyzing the function traces, extracting the identifiers and filtering out common identifiers using search engine and pre-built whitelist. For each sample, it took 789 s to fulfill all these tasks on average. For backward slicing, we find that it took 214 s on average for each identifier. Meanwhile, the longest case is 530 seconds and the shortest case is 30 s.
- **Impact Analysis** We measure the overhead of our offline parsing part to handle two execution traces with 1-min malware running time. The overhead for 500 cases is around 24 h. It means that for each case, it takes around $2 - 3$ min to verify its impact.

We note that the generation is a one-time effort in the analysis environment. The more important overhead that users care about is the one on their end hosts.

11.6.6.2 Deployment Overhead

We now report the deployment overhead on each end host.

For *static and algorithm-deterministic HoneyResource*, the overhead is negligible (almost zero) because in most of the time we only need to install some system resource or replay the resource-identifier-generation slice *for one time*. In our experiment, it takes only 34s to install all the 373 static HoneyResource onto one end-host machine. It includes copying/activating the resources and correctly setting up their privileges. For 44 algorithm-deterministic HoneyResource, we need to run program slices on the machine. It takes 1,131s (25.70s for each HoneyResource on average) to deploy all the HoneyResource. Note these HoneyResource are packed with installation scripts and there are no user interactions involved.

For *partial static* HoneyResource, it adds a little more overhead to the end host. The overhead mainly comes from the identifier comparison after we intercept the call. In our test, the highest extra overhead is below 4.5% for injecting 119 partial static HoneyResource. Among 4.5% overhead, around 3.9% comes from the function hooking, which is relatively stable even the HoneyResource number increases. Hence, it could be expected that even the number of partial static HoneyResource has been expanded by 10 times, we could still efficiently control the overhead under 12% for each host. More importantly, in most cases, we do not need to inject all the HoneyResource at the same time (to be discussed in Sect. 11.7).

11.7 Limitations and Future Work

Our system is not perfect. In this section, we discuss its limitations and outline our future efforts.

11.7.1 Evasions from Malware

It is possible to evade our HoneyResource if malware authors are aware that we are using certain resource as the HoneyResource. They can drop the specific resource checking logic or change the resource name in the new version. However, the former will possibly lead to re-infection and thus may be not desired. While the latter approach is possible, if we consider the wide and random propagation of worm or botnet malware, our HoneyResource still makes the malware harder to decide whether the system has actually been infected or not. Hence, if the malware binary cannot run when over two instances on the same machine, our HoneyResource can bring the malware into a dilemma that the target system may have actually been infected before or it has installed our HoneyResource system. Even though malware can run with multiple instances, periodically changing the identifiers may finally result in multiple instances running in one machine. It also creates extra risks of being detected.

Certainly, malware authors could obfuscate the malware code to frustrate our HoneyResource generation such as using control dependence to propagate data [24]. In fact, in some cases, there is actually no propagation chain and the conditional check is directly operated with the resource values. While future malware could deliberately introduce additional data propagation and obfuscate through control dependence, to address such problem will be one of our future efforts.

11.7.2 Limitation on Dynamic Analysis

In AUTOVAC, we intensively apply multiple data flow tracking techniques such as taint analysis and program slicing. Therefore, AUTOVAC unavoidably suffers from the problems brought by these dynamic analysis techniques [13]. For instance, in our candidate selection/analysis, our taint analysis could cause *overtainting* [7] thus resulting in more candidate resources to analyze. Fortunately, due to our impact analysis and exclusive analysis, we can still easily filter out those unsuitable HoneyResource.

In addition, some imprecise interpretation of differential function calls may cause the underestimation of the actual impact of certain resources. Some previous work [22] has discussed several approaches to gain a better understanding of malware's high-level behaviors. We could leverage these techniques to refine our result in future work.

11.7.3 Potential False Positive

Some of our automated analysis techniques (e.g., the use of search engine) may also return incomplete/inaccurate results. Meanwhile, our exclusiveness analysis and clinic test may not cover all benign programs such that it is possible to have some resource collision between our HoneyResource and some benign programs. Improving these issues is our another venue of future work.

11.7.4 Deployment Issues

One concern for the HoneyResource deployment is that injecting a large number of HoneyResource into end hosts may annoy the user. Note that most generated

HoneyResource in practice are just some files, mutexes, and registry entries, whose sizes are tiny or even with 0 byte. This is pretty lightweight compared with the case that AV tools typically store millions of signatures on an end host. In addition, as mentioned before, as a complementary technique to existing solutions, our prevention scheme can be mainly used for some high-profile, large-scale, and severe malware infections, instead of for all malware.

11.7.5 Deception Goals

In this work, we mostly focus on how to use HoneyResource to stop or impact the proper infection/execution of malware for the defense purpose. While these are important deception goals, many times we are also interested in deceiving malware into continuing the normal execution so that we can observe more activities, understand the intention from the malicious operators behind malware, or even mislead them for the defense purpose. Our future work will investigate more on this direction.

11.8 Related Work

11.8.1 Immunization-Based Defense

In [14], Manuel et al. proposed an end-to-end approach to make end hosts immune from fast-propagating worms through collaborative worm detection and self-certifying alerts. Packet Vaccine [32] followed this direction and derived the network signatures of malicious packets to be used at the network level to filter unwanted packets. Different from these previous work, AUTOVAC does not investigate the exploits nor vulnerabilities that malware targets, and instead it analyzes the system resource constraints of malware and attempts to extract effective HoneyResource to immunize a clean system from future malware infection.

In a concurrent study, Andre et al. [28] proposed the idea of using infection markers to prevent malware infection. While both are inspired by the biological vaccine concept, we systematically explore this problem and our HoneyResource are more general and broader than simple infection markers. Employed techniques are also substantially different; instead of treating the malware as a black box, AUTOVAC conducts more fine-grained binary analysis on malware internals, performs more analysis (e.g., exclusiveness, impact) in the automatic HoneyResource generation, and has more delivery/deployment options.

11.8.2 Dynamic Malware Analysis

Due to the severe threat of malware, tons of research has been carried out on analyzing malware behavior (e.g., [9, 12, 16, 20, 22]) and classifying malware (e.g., [15, 19, 30]). Certainly, AUTOVAC complements these techniques by exploring a new direction to stop malware infections.

In AUTOVAC, we design several dynamic binary analysis techniques to automate the production of malware HoneyResource. There has been a significant amount of work [11, 16–18, 21, 29] on dynamic binary analysis. In particular, prior research [23, 29] has explored the enforced execution and reverting to trigger malware's dormant functions [23, 29]. Our enforced execution applies similar techniques introduced in the forced execution [29] but we focus on these *environment/system resource-sensitive* branches.

We also leverage taint analysis and program alignment techniques. Different from full taint analysis in the previous work [17, 18] and block-level program alignment [25], our proposed solution avoids the overhead caused by full execution tracking with a particular focus on the targeted malware behavior in our problem domain.

11.9 Conclusion

In this chapter, we present AUTOVAC, a new complementary malware defense scheme that aims to automatically extract malware HoneyResource from given malware samples. Our evaluation shows that it is an appealing approach that works on many real-world malware families. In particular, the HoneyResource can be used to deceive malware for stopping its infection. To demonstrate the real-world practicability, we have implemented our prototype system using several dynamic program analysis techniques, and conducted empirical evaluations on a large set of real-world malware samples. Our experimental results show that we can successfully extract working HoneyResource for many malware families including Conficker, Sality, and Zeus.

11.10 Exercise

Ex. 1 Discuss the reasons why malware authors want to avoid duplicate infection? What is the effect of duplicate infection?

Ex. 2 Analyze the following assembly code. Is there any memory/register that will be tainted by AUTOVAC after the execution? Why?

Listing 1: Code Example

```
msg     byte    "mutex_test", 10
handle  dword   ?

        section .text
go:
        push    msg
        push    dword 0
        push    dword 0
        call    _OpenMutex
        xor      eax, eax
        push    -11
        call    _GetStdHandle
        mov     handle, eax
```

Ex. 3 Review the reference paper [7] and Sect. 11.7, and discuss how the limitation of tainted analysis may affect the effectiveness of AUTOVAC.

Ex. 4 Similar to biological HoneyResource which commonly has some side effect for patients, *malware HoneyResource* could also have some side effect on a user's system. Discuss possible side effects and how to prevent them.

Acknowledgements An early version of this chapter appeared in ICDCS'13 [31] . This research is partially supported by NSF (Grant No. CNS-0954096), AFOSR (Grant No. FA9550- 13-1-0077), and DARPA (Grant No. 12011593). All opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF, AFOSR, or DARPA.

References

1. Anubis: Analyzing Unknown Binaries. https://seclab.cs.ucsb.edu/academic/projects/projects/anubis/.
2. DynamoRIO . http://dynamorio.org/.
3. malc0de. http://malc0de.com/database/.
4. Temu . http://bitblaze.cs.berkeley.edu/temu.html.
5. Virustotal. https://www.virustotal.com/.
6. Zeus Trojan horse. http://en.wikipedia.org/wiki/Zeus_(Trojan_horse).
7. T. Avgerinos, E. Schwartz, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. of IEEE S&P 2010*.
8. A.Zeller. Isolating cause-effect chains from computer programs. In *Proc. of the 10th ACM SIGSOFT symposium on Foundations of Software Engineering*, 2002.

9. U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. of NDSS'09*, 2009.
10. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proceedings of Computer Aided Verification (CAV)*, July 2011.
11. J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proc. of ACM CCS'09*, 2009.
12. Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proc. of International Symposium on Software Testing and Analysis*, 2012.
13. L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA 2008*.
14. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proc. of SOSP'05*, pages 133–147, Brighton, United Kingdom, 2005.
15. M. Fredrikson, J. Somesh, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, 2010.
16. S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of ACM Symposium on Operating Systems Principles*, October 2003.
17. C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. of USENIX Security'09*, 2009.
18. C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proc. S&P'10*, 2010.
19. J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006.
20. A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proc. of the 17th ACM CCS*, 2010.
21. Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, February 2010.
22. L. Martignoni, E. Stinsony, M. Fredrikson, S. Jhaz, and J. C.Mithchelly. A layered architecture for detecting malicious behaviors. In *RAID 2008*.
23. A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proc. S&P'07*, 2007.
24. M.Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proc. NDSS'08*, 2008.
25. N.Johnson, J.Caballero, Z.Chen, S.McCamant, P.Poosankam, D.Reynaud, and D.Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.
26. P. Porras, H. Saidi, and V. Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. http://mtc.sri.com/Conficker/, 2009.
27. I. Trestian, S. Ranjan, A. Kuzmanovic, and A. Nucci. Unconstrained Endpoint Profiling (Googling the Internet). In *ACM SIGCOMM'08*.
28. A. Wichmann and E. Gerhards-Padilla. Using infection markers as a vaccine against malware attacks. In *Proc. of the 2nd workshop on Security of Systems and Software resiLiency*, 2012.
29. J. Wilhelm and T. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proc. of RAID'07*, 2007.
30. H. Xin, C. Tzi-cker, and S. Kang G. Large-scale malware indexing using function-call graphs. In *Proc CCS '09*, 2009.
31. Z. Xu, J. Zhang, G. Gu, and Z. Lin. Autovac: Towards automatically extracting system resource constraints and generating vaccines for malware immunization. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS'13)*, Philadelphia, July 2013.
32. X.Wang, Z.Li, J.Xu, M.Reiter, C.Kil, and J.Choi. Packet vaccine: black-box exploit detection and signature generation. In *Proc CCS'06*, 2006.