Security and Safety, Vol. 2, 2023023 (2023)
https://doi.org/10.1051/sands/2023023
sands.edpsciences.org

Research Article OPEN ACCESS

Other Fields

Multi-path exploration guided by taint and probability against evasive malware

Fangzhou Xu[1,3,4], Wang Zhang[1,3,4], Weizhong Qiang[1,3,4,6,*], and Hai Jin[1,2,5]

[1] *National Engineering Research Center for Big Data Technology and System, Wuhan 430074, China*
[2] *Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Wuhan 430074, China*
[3] *Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, Wuhan 430074, China*
[4] *School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China*
[5] *School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*
[6] *Jinyinhu Laboratory, Wuhan 430040, China*

Received: 8 May 2023 / Revised: 20 June 2023 / Accepted: 11 August 2023 / Published online: 05 September 2023

Abstract Static analysis is often impeded by malware obfuscation techniques, such as encryption and packing, whereas dynamic analysis tends to be more resistant to obfuscation by leveraging concrete execution information. Unfortunately, malware can employ evasive techniques to detect the analysis environment and alter its behavior accordingly. While known evasive techniques can be explicitly dismantled, the challenge lies in generically dismantling evasions without full knowledge of their conditions or implementations, such as logic bombs that rely on uncertain conditions, let alone unsupported evasive techniques, which contain evasions without corresponding dismantling strategies and those leveraging unknown implementations. In this paper, we present Antitoxin, a prototype for automatically exploring evasive malware. Antitoxin utilizes multi-path exploration guided by taint analysis and probability calculations to effectively dismantle evasive techniques. The probabilities of branch execution are derived from dynamic coverage, while taint analysis helps identify paths associated with evasive techniques that rely on uncertain conditions. Subsequently, Antitoxin prioritizes branches with lower execution probabilities and those influenced by taint analysis for multi-path exploration. This is achieved through forced execution, which forcefully sets the outcomes of branches on selected paths. Additionally, Antitoxin employs active anti-evasion countermeasures to dismantle known evasive techniques, thereby reducing exploration overhead. Furthermore, Antitoxin provides valuable insights into sensitive behaviors, facilitating deeper manual analysis. Our experiments on a set of highly evasive samples demonstrate that Antitoxin can effectively dismantle evasive techniques in a generic manner. The probability calculations guide the multi-path exploration of evasions without requiring prior knowledge of their conditions or implementations, enabling the dismantling of unsupported techniques such as C2 and significantly improving efficiency compared to linear exploration when dealing with complex control flows. Additionally, taint analysis can accurately identify branches related to logic bombs, facilitating preferential exploration.

Keywords Malware analysis, dynamic binary instrumentation, forced execution, taint analysis, evasion detection

Citation Xu F, Zhang W, Qiang W et al. Multi-path exploration guided by taint and probability against evasive malware. Security and Safety 2023; **2**: 2023023. https://doi.org/10.1051/sands/2023023

* Corresponding author (email: wzqiang@hust.edu.cn)

1 Introduction

Malware is a widespread threat to information systems, with nearly 100 million new malware samples being identified each year [1]. Various analysis methods have been proposed to identify malware samples automatically. Static analysis is a widely used method for malware detection, but obfuscation techniques, particularly those that reveal little information about behaviors before execution, such as encryption [2], can have a significant impact on its effectiveness [3] by removing static features. In contrast, dynamic analysis is more resistant to obfuscation because it monitors the behavior that is actually executed. However, malware samples often employ evasive techniques to hinder dynamic analysis [4]. These techniques extract information from the environment to detect whether the sample is being analyzed. If it is, the control flow is redirected away from malicious behaviors [5], preventing dynamic analysis from capturing these behaviors and effectively evading detection.

Therefore, the ultimate goal of defeating evasive techniques is to explore hidden paths. For the two phases of evasion, namely obtaining information and altering control flow, researchers have proposed two types of countermeasures, information-based and control-flow-based.

Active anti-evasion countermeasures are information-based methods that actively modify environmental information obtained through evasive techniques. Their purpose is to deceive evaders into believing that they are not under analysis, thus enabling the triggering of malicious operations. In comparison to passive methods that generate artifacts and mimic user behaviors in normal hosts, active anti-evasion countermeasures offer greater customization and extensibility to other user functions. Relevant works [6–9] have analyzed and defeated various evasive techniques, revealing their prevalence in malware samples. Active anti-evasion countermeasures can precisely modify detection results to satisfy the conditions imposed by evasive techniques, all without incurring additional exploration overhead [10]. However, logic bombs represent a distinctive form of evasion that relies on uncertain conditions, necessitating manual analysis to determine the conditions needed to trigger malicious behavior. Currently, there is limited research available on logic bombs, let alone automatic anti-evasion efforts.

Multi-path exploration [11–13] is a control-flow-based countermeasure aimed at achieving extensive program coverage by exploring paths that may contain malicious behaviors. It offers a generic approach that can effectively overcome various evasive techniques without prior knowledge. However, multi-path exploration can be inefficient due to the path explosion problem, which introduces significant overhead. Moreover, the lack of targeted exploration for specific evasive techniques can result in unnecessary exploration overhead.

In this paper, we propose a method that combines automatic multi-path exploration with active anti-evasion countermeasures to achieve a balance between efficiency and coverage when dealing with evasive techniques. Multi-path exploration generically defeats logic bombs and other unsupported evasive techniques. Furthermore, to improve exploration efficiency, path planning based on dynamic analysis is used to select the most probable evasion-related paths for exploration. Specifically, when dealing with logic bombs that rely on uncertain conditions, our method identifies and marks relevant environmental information and accurately switches branches based on taint analysis. For evasive techniques that are difficult to mark, it calculates branch probabilities and prioritizes those least likely to be executed for multi-path exploration. Additionally, to minimize the number of techniques that need to be explored, we have integrated several active anti-evasion countermeasures against known evasive techniques.

We have implemented Antitoxin, a dynamic analysis system for automatic evasion analysis of Windows x86 binaries, using Intel Pin [14]. The evaluation of Antitoxin is on a set of complex evasive malware samples containing diverse evasive techniques. The experimental results demonstrate that Antitoxin efficiently explores logic bombs and unsupported techniques with minimal rounds. Furthermore, Antitoxin successfully mitigates the impact of most evasive techniques through its active anti-evasion countermeasures.

In summary, we make the following contributions:

- We have conducted an in-depth study on evasive techniques to dismantle them through multi-path exploration and active anti-evasion countermeasures based on their implementations accordingly.
- We have proposed a path-planning strategy to explore the most evasion-related paths, which can effectively explore logic bombs and unsupported evasive techniques with the guidance of taints and probability.

– We have implemented a prototype called Antitoxin for automatically analyzing evasive malware. The performance evaluation of different path-planning strategies demonstrates the effectiveness of Antitoxin.

2 Related work

The static analysis identifies evasion based on statically extracted features. Branco *et al.* [15] summarized 50 static detection methods for anti-debugging, anti-disassembly, and anti-VM evasions by checking the disassembled code for the presence of specific sensitive behaviors. Static analysis is efficient but vulnerable to obfuscation techniques. Aghakhani *et al.* [3] investigated the classification performance of machine learning-based static features on packed samples and found that static information was not always indicative of actual behavior, based on experiments on 392 168 executables. Chenke *et al.* [16] found that static analysis was ineffective in obtaining behavior and control flow information under strong protection, such as encrypting the whole program using AES.

In contrast to static analysis, dynamic analysis is more robust to obfuscation. To defeat evasions, it covers multiple paths in several executions or dismantles evasive techniques in each execution. The first dynamic method is multi-path exploration. Moser *et al.* [11] proposed to explore multiple paths to identify malicious behaviors that are triggered only under specific conditions. They tracked inputs of interest and solved the input-related conditions to achieve new coverage. Other works, such as [17–20], used symbolic execution to precisely solve the conditions during exploration. However, analyzing and solving constraints for the entire sample in this manner is infeasible due to the high overhead involved [21]. To alleviate this weakness, Zhao *et al.* [22] and Sebastio *et al.* [23] proposed enabling symbolic execution on demand. Furthermore, to completely get rid of the constraint-related overhead, Peng *et al.* [12] proposed X-Force, a technique that utilizes forced execution to switch input-related branches. Experiments on ten real-world malware samples demonstrated that forced execution can effectively overcome evasive malware checks and expose malicious behavior. Building on X-Force, You *et al.* [13] developed PMP, which employs memory pre-planning to minimize memory exceptions caused by force execution without tracing the memory correlations between memory addresses, which can detect 98% more payload than X-Force.

However, multi-path exploration faces the path explosion problem [20], as the number of paths exponentially increases with each branch, leading to intractability. Thus, it is crucial to cover critical branches within limited rounds. Existing works [24, 25] identified potential triggers statically and validated them in dynamic executions. However, static analysis may be infeasible due to obfuscation. X-Force [12] and PMP [13] begin multi-path exploration from input-related data, but their high coverage necessitates numerous repeated executions. Therefore, there is a need for a more targeted design for evasive techniques. Similarly, to accelerate the multi-path exploration for software testing, Böhme *et al.* [21] employed a probability-based technique, and Wang *et al.* [26] suggested using several indicators to prioritize candidate branches.

Another dynamic method is deploying active anti-evasion countermeasures after figuring out how evasions detect analysis environments. Xu *et al.* [27] introduced taint analysis [28] for dissecting evasions based on system resource conditions such as files, mutexes, registries, windows, processes, libraries, and services. Based on prior knowledge, recent works systematically classified and dismantled various evasive techniques, creating a transparent analysis environment to trigger more malicious behaviors. Polino *et al.* [6] focused on techniques that target dynamic binary instrumentation (DBI). Maffia *et al.* [29] and Galloro *et al.* [9] actively dismantled more evasive techniques. D'Elia *et al.* [8] proposed a novel observe-check-replace design called BluePill, which reconciles transparency requirements with dissection capabilities. Experiments on 48 highly evasive samples demonstrated its ability to defeat evasive techniques and assist in manual analysis.

Defeating evasive techniques by thoroughly analyzing how they are implemented and the conditions they rely on is a practical approach, but it may fail when malware developers use new techniques and logic bombs using unknown conditions. To address this challenge, we propose a two-fold approach. Firstly, we use multi-path exploration guided by taint and probability as a generic strategy against unknown conditions and unsupported evasive techniques. Secondly, we actively and efficiently dismantle known evasive techniques to reduce the exploration overhead when full knowledge of evasions is available. Additionally,

to avoid being hindered by obfuscation, we explore evasive techniques based on dynamic information rather than statically identifying targets in advance.

3 System design

The system aims to automatically analyze and explore malware samples equipped with evasive techniques, particularly logic bombs, and unsupported evasive techniques, and trigger the guarded malicious behaviors. To achieve this, the system should possess the following features:

(i) operation without the need for preprocessing the samples, simplifying the operation and minimizing the impact of obfuscation,
(ii) efficient dismantling of logic bombs and unsupported techniques by selecting exploration paths carefully, and
(iii) extensibility to existing anti-evasion countermeasures to reduce exploration overhead.

To provide the first feature, dynamic analysis is better than static analysis. To support the second feature, multi-path exploration is needed to generically dismantle evasions when lacking prior knowledge of conditions and implementations of evasions. Furthermore, to accelerate the exploration of evasions, we study the characteristics of evasions and focus on how to explore them preferentially. To provide the third feature, we should group different evasions and leave the necessary part of them for exploration.

3.1 Taxonomy of evasive techniques

This section will group evasive techniques to reduce exploration overhead by dismantling different groups of techniques through multi-path exploration and active anti-evasion countermeasures accordingly.

The multi-path exploration covers both true and false branches affected by environmental information, thus triggering hidden paths. And active anti-evasion countermeasures directly modify the information returned to the sample to cloak the analysis environment. Compared to multi-path exploration, the latter method dismantles evasions during each execution, resulting in overhead savings. However, modifying environmental information is based on prior knowledge of evasions, namely their implementations and conditions, including how evasions obtain environmental information and which values indicate analysis environments.

To propose active anti-evasion countermeasures correspondingly, we have conducted a comprehensive study on the implementations of evasive techniques and categorized them based on the semantics of the operations, as shown in Table 1, following the approach of previous studies [6–9, 29].

Evasive techniques can be classified into four categories: *hardware*, *software*, *timing*, and *running state*. The *hardware* method captures information related to hardware devices such as CPU, disk, display device, keyboard, memory, mouse, network adapter, and their respective drivers and firmware. The *software* method queries installed or running software to identify the presence and Wear-and-tear status of analysis tools obtained from filesystems, processes, modules, and services. The *timing* method compares timing differences between virtual and real environments, such as boot time, and time intervals for certain operations. The *running status* method checks information about the running sample, such as debug status and instruction address.

After obtaining the environmental information, evasive techniques check for specific values to determine whether the sample is executing in analysis environments. Some evasive techniques only return two values, such as *IsDebuggerPresent*, which returns *True* when debuggers exist. However, other evasive techniques may have various parameters, which we should group into two types based on whether they are used to detect analysis environments. For example, *NtQueryFileAttributes* can be used for evasion by querying "*... VMware Tools*", but for legitimate purposes when querying system DLLs. Therefore, blacklists need to be created to identify the values used for evasion and modify them to dismantle evasion.

Based on full knowledge of evasions, active anti-evasion countermeasures can be proposed to monitor behaviors obtaining the environmental information and modify blacklisted values to prevent the disclosure of the analysis environment. However, logic bombs are a unique type of evasion targeting special victims based on uncertain conditions, which means that we cannot group related values into two types and create generic blacklists for logic bombs in different samples. Thus, due to the lack of prior knowledge

Table 1. Evasive techniques and the low-level implementation

Category		Instruction	System call	API
Hardware	CPU	cpuid read memory		GetSystemInfo WMI[1]
	Device		registry[2] NtUserCallTwoParam NtUserEnumDisplayDevices	GlobalMemoryStatusEx GetDiskFreeSpace GetPwrCapabilities GlobalMemoryStatusEx SetupDiGetDeviceRegistryProperty WMI WNetGetProviderName
	Driver			GetDeviceDriverBaseName
	Firmware		NtQuerySystemInformation	GetSystemFirmwaretable
	Network adapter			GetAdaptersInfo WMI
Software	File		NtCreateFile NtQueryAttributesFile NtQueryDirectoryObject	FindFirstFile
	Module		NtQuerySystemInformation	GetModuleFileName GetModuleHandle
	Process		NtQuerySystemInformation NtQueryInformationProcess	Process32Next
	Service			EnumServicesStatusExW QueryServiceConfig
	System		registry	GetEnv
	Window text		NtUserFindWindowEx	
Timing		rdtsc	NtQueryPerformanceCounter	GetTickCount
Running status	Instruction address	in int 2e		
	Debugging	read memory[3]	NtQueryObject NtQuerySystemInformation NtQueryInformationProcess	CheckRemoteDebuggerPresent GetThreadContext IsDebuggerPresent

Note: [(1)]WMI represents WMI queries through *WMI_Get("SELECT ...")*. [(2)]registry represents registry operations, including NtEnumerateKey, NtOpenKey, and NtQueryValueKey. [(3)]read memory represents accessing specific memory addresses, like PEB.

of conditions or implementations of evasions, logic bombs, and other unsupported techniques will be dismantled by multi-path exploration.

To implement countermeasures, we need to monitor and modify environmental information obtained by evasions. Although information sources have different semantic categories, they can be classified into more common sources. The fundamental source of information is instructions, followed by system calls and APIs, which are common interfaces provided by the Windows system. Other sources such as memory access, registry, and *Windows Management Instrumentation* (WMI) are integrated into the existing three categories. This is because memory access is achieved by instructions, registry queries are implemented by system calls, and WMI queries are conducted through calling APIs. So, we intercept behaviors at three levels and modify blacklisted environmental information in parameters, and return values, which are stored in memory and registers.

Some other techniques, such as time stalling and exception throwing, attempt to interrupt or delay the analysis process to prevent the disclosure of sensitive information. Since the purpose of stalling techniques is to interrupt the analysis rather than lead to multiple paths as evasions, we dismantle these techniques appropriately to proceed with the analysis.

3.2 Exploring logic bombs

After dismantling evasions with full knowledge through environmental information modification, logic bombs must be addressed. Logic bombs trigger malicious behavior only under specific conditions to evade analysis. As these conditions vary among samples, modifying environmental information to a specific value is insufficient to dismantle all logic bombs. Thus, multi-path exploration is necessary to explore multiple execution paths, and selecting a small set of paths for exploration is crucial to avoid excessive overhead. To automate analysis, the selection should depend on runtime information rather than preprocessing results from other tools, such as disassembled code, which can be impeded by obfuscation.

Table 2. Evasive techniques using uncertain conditions

Category	Technique
Time	GetSystemTime
	GetSystemTimeAsFileTime
Location	GetLocaleInfo
	GetTimeZoneInformation
Language	GetKeyboardLayout
	GetUserDefaultLangID
	GetUserDefaultUILanguage
	GetSystemDefaultLangID
	GetSystemDefaultUILanguage
	WMI(MUILANGUAGES)

Logic bombs can be categorized into three types: *time*, *location*, and *language*, as shown in Table 2. The *time* method compares the current time with a predefined time to determine subsequent behaviors, which can be used to launch attacks on different infected victims at the same time. The *location* method obtains location information to target specific regions. The *language* method identifies the language of the system and keyboard layout to target victims using specific languages.

Identifying branches altered by logic bombs is necessary for accelerating multi-path exploration. Although logic bombs alter control-flow branches based on uncertain conditions, they obtain environmental information in the same three levels as general evasive techniques. Thus, intercepting related behaviors with the mechanism mentioned above is feasible. To determine which branches are affected by logic bombs, we mark related parameters and return values rather based on taint analysis than modifying them. By contrast, since some evasive techniques have been dismantled by active anti-evasion countermeasures, we identify branches related to these techniques to reduce unnecessary exploration overhead.

Specifically, evasive techniques can be categorized into *dismantled*, *evasive*, and *suspicious*, representing no exploration, prioritized exploration, and exploration before normal paths, respectively.

The identification of *dismantled* evasion techniques involves comparing parameters and results to predefined blacklists and altering environmental information. For instance, *IsDebuggerPresent* can be dismantled by returning a false value, and querying a registry containing "VBOX" can be dismantled by modifying the parameter or returned value. Multi-path exploration excludes branches with *dismantled* taints and can be used to validate and locate the use of evasion.

Evasive techniques are logic bombs that require further exploration. For instance, the use of *GetSystemTime* to query the current time and trigger malicious behavior after a specific date. Since logic bombs obtain environmental information and compare it with uncertain conditions, the addition of *evasive* taints to the environmental information can help identify affected branches that require prioritized exploration.

Suspicious techniques are sensitive techniques that cannot be identified as evasion because they use environmental information that is not included in predefined blacklists. Finite blacklists cannot identify a new condition, resulting in no countermeasures being implemented. Therefore, we consider *suspicious* taints as supplementary information to help identify techniques that have not yet been dismantled effectively.

The malware category is not static, but rather dynamic. The category changes to *dismantled* when an *evasive* or *suspicious* technique is mitigated. However, if the malware adapts to a new condition of a previously dismantled technique, the corresponding countermeasure will no longer be effective, and the category will revert to *evasive*.

3.3 Filtering behaviors

Dynamic analysis is a behavior-based method. In an ideal scenario, only malware behaviors are analyzed while filtering out legitimate behaviors of system libraries. For instance, early-stage functions invoked by another library function are excluded since the code inside libraries has been wrapped as higher-level APIs and remains unchanged across various executions. Thus, monitoring parameters and return values of APIs is sufficient for analysis.

Table 3. Dynamic executable regions

Category	Technique
Allocated memory	NtAllocateVirtualMemory NtMapViewOfSection NtProtectVirtualMemory
Child process	CreateProcess
Injection	NtOpenProcess NtWriteVirtualMemory
Obfuscating API names	GetProcAddress GetModuleHandle LoadLibrary MapViewOfFile
Self-modifying	write memory[1]

Note: [(1)]Write memory represents modifying the instructions of the executable under analysis.

Besides excluding system libraries, we should extend the monitored region in some cases. Malware can evade analysis by performing behaviors outside the original executable, using techniques that are generally intended to hinder static analysis but can also affect dynamic analysis. Prevalent techniques include the *allocated memory* method, which writes code to memory allocated with execute permission and executes it; the *child process* method, which performs behaviors in a child process to evade dynamic analysis that only monitors the main process; the *injection* method, which writes code to another legitimate process similar to creating a child process; the *obfuscating API names* method, which dynamically resolves API addresses to prevent static analysis from correlating the address with the API; and the *self-modifying* method, which overwrites executable instructions at runtime, potentially bypassing analysis. Table 3 provides a summary of these techniques.

3.4 Logging information

To support multi-path exploration and manual dissection, it is important to log information beyond evasive techniques. Multi-path exploration requires the repeated execution of malware to uncover new paths, which means that control flow must be recorded to distinguish branches that have not been covered. Control flow can be divided into instructions, basic blocks, routines, and paths, but basic blocks are preferred since they contain single control-flow transfers. Thus, information on each basic block should be recorded, including address, size, branch, and frequency.

In addition, to control flow, it is also important to record and parse sensitive behaviors related to *strings*, *networks*, *files*, *registries*, *services*, and *processes*, which can provide insight into malicious behaviors such as connecting remote servers, modifying system settings, and creating scheduled tasks. By analyzing this information, analysts can locate critical branches that trigger malicious behaviors.

4 Analysis framework

The main workflow of Antitoxin is depicted in Figure 1. It performs analysis of a malware executable without preprocessing and selects a new path for each execution through path planning. During execution, Antitoxin records coverage, taints, and other sensitive behaviors. The path planning is guided by coverage and taints and selects branches to be preferentially explored after each execution.

4.1 The design of analysis framework

We have developed an automatic analysis system for Windows x86 programs that actively dismantle known evasive techniques through active anti-evasion countermeasures and defeats logic bombs and unsupported techniques through multi-path exploration. To achieve this, we have employed dynamic

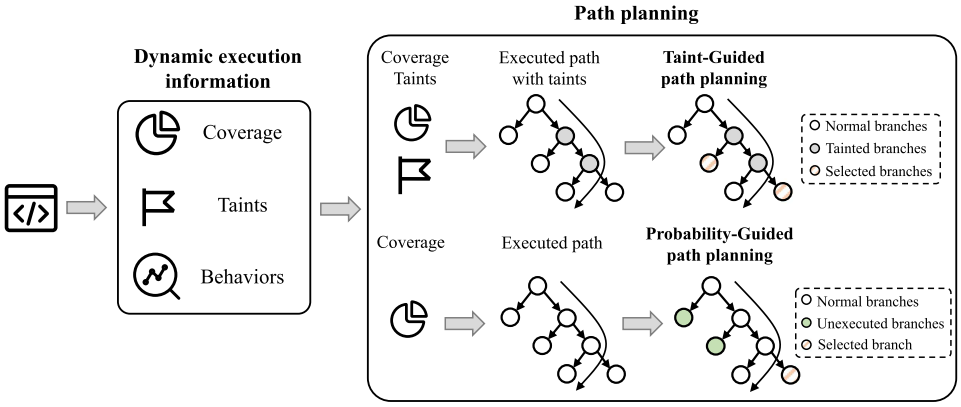

Figure 1. The workflow of Antitoxin

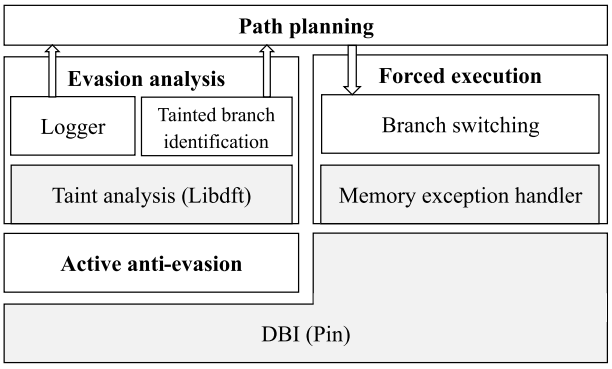

Figure 2. The overall framework

analysis. Dynamic analysis can be implemented by different techniques such as virtual machine introspection (VMI), emulator, bare-metal, DLL injection, and dynamic binary instrumentation (DBI). Among these techniques, DBI has no semantic gap and can flexibly monitor and modify environmental information obtained by evasive techniques. DBI also allows for the invocation of system libraries, which makes it easier to integrate with existing countermeasures and implement other necessary functions. Therefore, we use Intel Pin [14] to instrument behaviors and take full control of the execution.

Based on Pin, we register callback functions to analyze evasive techniques at three levels: instruction, system call, and API. Figure 2 shows the overall framework of the system, which comprises several modules that work together to detect malicious behaviors in malware samples.

The active anti-evasion module identifies and dismantles known evasive techniques based on predefined blacklists, thereby reducing the overhead of exploration. The evasion analysis module adds various taints to sensitive techniques, identifies branches affected by taints, and continuously records control flows to facilitate multi-path exploration and manual analysis. After each execution, the path planning module utilizes information from taint and coverage from the evasion analysis module to prioritize the exploration of branches based on a probability and taint-guided algorithm. The forced execution module switches selected branches and handles memory exceptions. Additionally, the framework traces all dynamically generated executable regions of malware samples to capture stealthy malicious behaviors and distinguish them from legitimate system library behaviors.

4.2 Functions and modules of framework

Based on the framework, we will demonstrate the functions of different modules and their implementations.

4.2.1 Active anti-evasion

When full knowledge of evasions is available, we implement active anti-evasion countermeasures accordingly. Specifically, we use Pin to identify and dismantle known evasive techniques by registering callbacks at three levels. These callbacks parse parameters and return values, which are then matched against predefined blacklists and modified to irrelevant or invalid values as needed. For instance, the *GetModuleHandle("pin.exe")* function checks module information and can be dismantled by replacing the module name with an invalid string. Similarly, the *GetAdaptorInfo* function checks for VM-related MAC addresses, which can be dismantled by replacing the outcomes with normal or invalid MAC addresses.

To mitigate timing checks, we utilize the "time fast forwarding" countermeasure from BluePill [8], which manually aggregates delays from multiple sources to achieve a consistent virtual time. Notably, behaviors are often implemented by other low-level APIs or system calls. For instance, the API *EnumerateKey* invokes the system call *NtEnumerateKey*, while *GetCursorPos* invokes *NtUserCallTwoParam*. To prevent duplicate checks and missing low-level behaviors, we register callbacks at the lower level.

4.2.2 Evasion analysis

To accelerate the multi-path exploration of logic bombs with the lack of prior knowledge of conditions, we first add various taints to environmental information. We use *dismantled* taints to reduce exploration overhead, *evasive* taints to identify branches related to logic bombs, and *suspicious* taints to help detect unsupported techniques. Next, we identify control-flow branches affected by tainted information to guide the multi-path exploration process.

Following BluePill [8], taint analysis is implemented based on the fork of libdft [30], which provides a byte-level mechanism for adding, querying, and propagating taints between memory and registers. When adding taints, we mark the entire structure or specific element for fixed-size data structures and calculate the length dynamically for variable-length data like strings. However, we conservatively mark the first byte of unknown information to prevent false positives.

Normally, taints propagate effectively through libdft, even when invoking APIs such as *memcpy* and *strcpy*. However, there are scenarios where propagation may be interrupted, leading to bogus propagation. In our system, we have addressed these issues. On one hand, some countermeasures may interrupt complete propagation. For instance, setting a string to empty will interrupt propagation because string functions cannot read content from an empty string. However, as these techniques have been dismantled, the interruption has little negative impact. In addition, we can temporarily deactivate corresponding countermeasures to resume propagation for dissection. On the other hand, libdft may fail to trace some APIs, such as the propagation interrupts in *WCSToMBEx*. In such cases, we manually propagate taints in these APIs to complement libdft. Additionally, bogus taints may arise in instructions like *and operand,0* and *or operand,0xffff*, where the taints of operands should be cleared, but are not considered in libdft.

Identifying the tainted branches. Evasive techniques compare environmental information with predefined values and alter the control-flow branches accordingly. For example, the *jcc* (conditional jump) jumps to different targets depending on the comparison results of *cmp* instruction for evasive purposes. We have found that flags set early by instructions like *cmp* affect the control-flow transfer when executing instructions like *jcc*. As instructions set flags based on operands, we need to decide which operands should be considered when analyzing taints. However, it is not appropriate to consider all the taints of operands as the taints of flags. For instance, instructions like *mov* and *xchg* could contain tainted operands, but they neither set flags nor affect control flow. To address this issue, we have categorized instructions that set flags into two types. For instructions with only read operands, such as *cmp* and *test*, we consider the union of taints from all operands. Otherwise, we only consider taints from write operand(s) since flags are set based on the destination operands. In addition, implicit operands are included through the interface of Pin.

As shown in Table 4, we categorize instructions related to flags into three categories: *setting*, *using*, and *resetting*. Specifically, we refer to taints of flags that derive from the operands as flag taints. The *setting* instructions modify flags, and we calculate flag taints from the operands as mentioned above. The *using* instructions are control-flow transfers that involve flags and are affected by taints if the flag taints are not empty. The *resetting* instructions are other control-flow transfers that do not depend on but modify flags, so we clear flag taints. Additionally, *setcc* and *movcc* are considered as *using* instructions, which do not directly alter control flow but should still be noticed.

Table 4. Instructions related to flags

Category	Instruction
Setting flags	adc, add, and, dec, imul, inc, mul, neg, or, sub, test, xadd, xor, bsf, bsr, bt, bts, btr, btr, btc, cmp, cmpsb, cmpxchg, cmpxchg8b, lar, lsl, rcl, rcr, rol, ror, salc, sar,shl, shr, sbb, scasb, shld, shrd
Using flags	jb, jbe, jcxz, jecxz, jl, jle, jmp, jmp_far, jnb, jnbe, jnl, jnle, jno, jns, jnz, jo, jp, jrcxz, js, jz, setb, setbe, setl, setle, setnb, setnbe, setnl, setnle, setno, setnp, setns, setnz, seto, setp, sets, setssbsy, setz, cmovb, cmovbe, cmovl, cmovle, cmovnb, cmovnbe, cmovnl, cmovnle, cmovno, cmovnp, cmovns, cmovnz, cmovo, cmovp, cmovs, cmovz
Resetting flags	call(near), call(far),ret(near), ret(far), iret, iretd, iretq

In conclusion, we save flag taints of *setting* instructions, check flag taints when executing *using* instructions, and clear the flag taints after the transfer of *resetting* instructions.

4.2.3 Path planning

Path planning selects the most likely evasion-related branches for multi-path exploration. We use multi-path exploration to dismantle logic bombs and other unsupported evasive techniques, of which taint analysis is used to identify branches related to logic bombs. However, in the case of unsupported evasive techniques where no taint information is available due to lacking prior knowledge of their implementations, the probability is calculated from coverage information to guide path planning in a more generic manner. We first implement a generic multi-path exploration algorithm to uncover new paths and then implement our path-planning algorithm to guide the exploration.

Multi-path exploration. Multi-path exploration executes the sample repeatedly and covers different paths by selecting and switching certain branches. To avoid selecting unreachable branches, we refer to X-Force [12] to adopt incremental path planning that attaches a newly selected branch to the sequence of selected branches.

Algorithm 1 presents a generic multi-path exploration algorithm. *Pool* stores multiple executions and explores one each round. Each execution is represented by a sequence of branches to be switched (*i.e.*, *switches*). A branch is defined as *(Jcc, Dst)*, indicating a conditional control-flow transfer and its destination. During the initialization phase, *Pool* contains an empty sequence representing an execution without any switching (line 1). In each round, a sequence of *switches* is selected from *Pool* using different path-planning strategies (lines 4–8). After the execution, coverage information is updated to support path planning (line 10). As described in incremental planning, if a branch after existing *switches* has not been covered, it will be attached to existing *switches* to generate a new sequence for exploration (lines 11–18). Furthermore, different path-planning strategies prioritize different branches for specific purposes (lines 4–7). The most straightforward algorithm is linear exploration, which explores all unsorted switches in *Pool* one by one (lines 4 and 5). Based on our findings, we implemented a path-planning strategy to accelerate the multi-path exploration of evasive techniques (lines 6 and 7), as shown in Algorithm 2.

Switches selection. The path-planning strategy sorts branches that have not been covered according to user-defined criteria and selects *switches* to explore. Specifically, we have developed an algorithm guided by taint and probability, which prioritizes the most likely evasion-related branches for exploration.

Taints guide multi-path exploration both directly and indirectly. Direct taints result from explicitly marked evasive techniques, requiring accurate exclusion or prioritization of related branches based on the taint category. Indirect taints, on the other hand, come from libraries outside the sample and can interrupt taint propagation for various reasons, such as unsatisfied conditions, unmonitored processing techniques, or variable value conversion. We address the first and second problems by manually propagating taints from source to destination, while for the last problem, we add an indirect taint flag to `N` basic blocks after a library returns. Unlike direct taints that propagate in memory and registers, indirect taints propagate along `N` consecutive basic blocks. To minimize false positives, we mark only the first basic block after the library return using taints.

When an operand has taints from multiple sources, branches are excluded only if the *dismantled* category exists, indicating that corresponding evasive techniques have been dismantled. Otherwise, exploration should consider the *evasive* and *suspicious* categories. Taints guide multi-path exploration with different priorities as shown in Table 5. Direct taints have higher priority than indirect taints, and *evasive* taints precede *suspicious* ones. Indirect *dismantled* taints are not propagated as they are not accurate indicators, and indirect *evasive* taints are given lower priority.

Algorithm 1: Multi-path exploration algorithm

Output: Ex – the set of execution information (including the records of behaviors, coverage of branches, evasion, *etc.*)
Input: $Executable$ – the sample to be analyzed
Definition: $switches$: $\overline{Jcc \times Dst}$ – a sequence of branches to be switched in an execution
$Pool$ – the set of switches
$ExploringSpace$ – the candidate branches

$Pool \leftarrow \{Nil\}$
$Path \leftarrow Nil$
while *Pool is not Nil* **do**
  if *Method is Linear* **then**
    $switches \leftarrow Pool.pop()$ #Linear exploration selects the last sequence from $Pool$
  else
    $switches \leftarrow Pool.pop(SelectSwitches(Pool, Path))$ #Path planning algorithm selects a sequence
  end
  Execute the $Executable$ and switch branch outcomes in switches
  Update Ex
  $Path \leftarrow$ the sequence of executed branches
  $t \leftarrow$ index of the last switch of $switches$ in $Path$
  $ExploringSpace \leftarrow$ remove the first t elements in $Path$
  foreach $(jcc, dst) \in ExploringSpace$ **do**
    $dst\_switch \leftarrow GetSwitch(jcc, dst)$ #Switching to the branch that was not executed
    if $!covered(jcc, dst\_switch)$ **then**
      $Pool \leftarrow Pool \cup switches \cdot (jcc, dst\_switch)$ #Adding branches that are not covered to $Pool$
    end
  end
end

Evasive techniques hide malicious behaviors and trigger normal behaviors when analysis environments are detected. Therefore, dynamic analysis is more likely to execute normal rather than malicious behaviors when we do not deal with evasive techniques, which means paths containing malicious behaviors have lower execution probabilities. Thus, probability guides multi-path exploration by selecting switches that are less likely to be executed, thereby enabling the dismantling of evasive techniques even in the absence of taints. That is, probability can guide multi-path exploration without prior knowledge of conditions and implementations of evasions. Drawing inspiration from the ant colony system (ACS) [31], which selects branches based on the probability calculated from execution frequency, we developed our algorithm based on the following heuristics.

Heuristic 1. Hidden by evasive techniques, the branches leading to malicious behaviors have a lower probability of execution.

Heuristic 2. Triggered after bypassing the evasive techniques, the branches leading to malicious behaviors are on longer paths.

Algorithm 2 describes a probability and taint-guided switch selection algorithm derived from ACS. Specifically, it accumulates pheromones of basic blocks (BBLs) during execution and calculates the execution probability of branches based on pheromones. The updating rules are as follows:

(i) The pheromones of basic blocks are initially set to a default value.
(ii) The pheromones of all basic blocks evaporate each round.
(iii) The pheromones are deposited into executed basic blocks each round.
(iv) The pheromones of basic blocks of the longest path evaporate every `M` rounds.

Table 5. Priority of taints

Category	Evasive	Dismantled	Suspicious
Direct taints	2	Do not explore	1
Indirect taints	1	No propagation	1

The formula for evaporating pheromone (lines 2 and 5) is as Equation (1):

$$\text{pheromone}_{\text{new}} = (1-\rho)\cdot \text{pheromone}_{\text{old}}. \tag{1}$$

The formula for depositing pheromone (line 2) is as Equation (2):

$$\text{pheromone}_{\text{new}} = \frac{Q}{\text{pathlen}} \tag{2}$$

where the length of the path is calculated as follows:

$$\text{pathlen} = \sum_{\text{BBL}\in\text{Path}} \text{BBLsize}. \tag{3}$$

Algorithm 2: Taint-guided ant colony system algorithm

Output: *SelectSwitches* – the selected switches returned
Input: *Pool* – the set of the switches
Path – the sequence of executed branches
Definition: *EvasivePool* – the switches containing branches affected by *evasive* taints
Round – execution times
BestPath – the longest path ever
TaintCategory – the category of taints

1 *EvasivePool* ← *Nil*
2 *UpdatePheromones*(*Path*) #Evaporating pheromones of all BBLs and depositing pheromones into the executed path each round
3 **if** *Round*%*M* == 0 **then**
4 | *EvaporatePheromones*(*BestPath*) #Evaporating the pheromones of the longest path each *M* round
5 **end**
6 **if** *PathLen*(*Path*) ≥ *PathLen*(*BestPath*) **then**
7 | *BestPath* = *Path*
8 **end**
9 Remove duplicate switches in *Pool*
10 **for** *switches* ∈ *Pool* **do**
11 | *category* ← *GetTaintCategory*(*switches*[−1]) #The taint category of the branch to be switched, which is the last branch of a sequence
12 | **if** *category is dismantled* **then**
13 | | *Pool.pop*(*switches*)
14 | **else if** *category is evasive* **then**
15 | | *EvasivePool.push*(*switches*) #Adding the sequence containing branches affected by *evasive* taints
16 | **end**
17 **end**
18 **if** *EvasivePool is not Nil* **then**
19 | *SelectSwitches* ← *Roulette*(*EvasivePool*) #Preferentially selecting sequences from *EvasivePool*
20 **else if** *Pool is not Nil* **then**
21 | *SelectSwitches* ← *Roulette*(*Pool*) #Selecting sequences from *Pool* based on execution probability
22 **end**
23 **return** *SelectSwitches*

Duplicate switches (line 9) and switches with *dismantled* taints (lines 12 and 13) are removed to reduce exploration overhead. Switches with *evasive* taints are prioritized by moving to *EvasivePool* (lines 14, 15 and 18, 19). Roulette-wheel method [32] selects switches from *EvasivePool* or *Pool* (lines 20 and 21) based on the probability calculated as follows:

$$P = \frac{(TaintCategory+1)^{\beta}}{\text{pheromone}^{\alpha}} \tag{4}$$

Table 6. ACS parameters

Parameter	Value	Definition
α	1	Pheromone heuristic factor
β	0	Expectation heuristic factor
ρ	0.4	Pheromone decay parameter
Q	10	Pheromone intensity
M	10	The number of ants

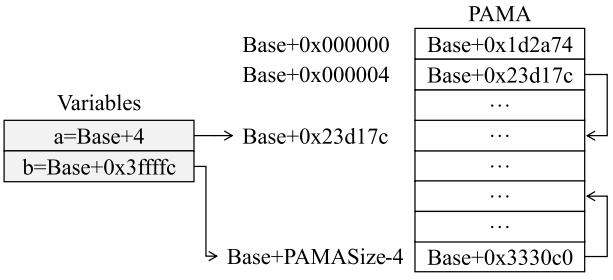

Figure 3. Memory pre-planning

where *TaintCategory* refers to the categories of taints given in Table 5, and α and β determine the weight of pheromones and taints, respectively, when calculating probabilities individually. Our ACS parameters are shown in Table 6. We set $\beta = 0$ to calculate probability solely from pheromones, as *dismantled* and *evasive* taints have already been considered in lines 12–16, and *suspicious* taints are currently used to assist manual analysis by highlighting suspicious parameters of sensitive techniques. When setting β to a positive value, branches affected by *suspicious* taints will be preferentially selected, and we will choose a reasonable value based on more experiments in the future. Users can set different parameters based on their findings.

4.2.4 Forced execution and exception handler

Forced execution explores new paths by forcefully setting outcomes of selected branches, which avoids the overhead of tracing and solving conditions. However, this approach can sometimes disrupt the original logic, such as initialization and validity checks, leading to memory exceptions. To handle such exceptions, we have implemented a preventive method and an on-demand exception handler.

PAMA. Following [13], a preventive method is used to initialize variables in advance, which helps to prevent memory exceptions. As shown in Figure 3, a memory area called PAMA (Pre-Allocated Memory Area) is pre-allocated and filled with carefully crafted random values that point to the area when interpreted as addresses. This means that initializing a variable to a PAMA address can prevent memory exceptions when accessed as a pointer. If the variable is later overwritten, the initializing will have no effect on the original behavior.

Unhandled memory exception. Although PAMA can prevent most memory exceptions related to initialization, an on-demand handler is needed to handle other exceptions.

First, a vectored exception handler (VEH) is registered to record the basic block (BBL) causing the memory exception for accurate handling. While adding checks for all instructions is feasible, it incurs significant overhead. Instead, the validity of memory accesses is checked inside the recorded BBL and invalid memory addresses are replaced with PAMA addresses. Memory addresses are calculated from registers and immediate, so the whole operand is replaced when only immediate exist. Otherwise, registers are modified so that the new address falls into the PAMA. Additionally, memory dependencies inside the recorded BBL are also captured for updating correlated variables. For example, in the code snippet in Figure 4, *eax* derives from the address *ebp+disp0*. If *eax+disp1* is an invalid address, *eax* will be modified to make *eax+disp1* a PAMA address. With dependency identification, the content of *ebp+disp0* is also modified to avoid similar exceptions in the future.

```
.text:00406434  mov  eax, [ebp+disp0]
.text:00406437  cmp  [eax+disp1], edi
.text:0040643D  jge  loc_4065DE
```

Figure 4. Memory dependency inside a basic block

4.2.5 Filtering addresses

To focus on malware behaviors, we filter out legitimate behaviors of system libraries and include executable regions dynamically generated by samples. Addresses are used to identify behaviors executed by malware. Specifically, instructions and APIs are filtered by instruction addresses and return addresses accordingly. However, system calls are commonly invoked by APIs rather than directly invoked by malware code, so we do not filter them.

Monitored regions begin with the main executable and dynamically include subsequent executable regions. We identify *allocated memory* regions by tracing the allocation and permission modification of memory and include regions where execution permission exists. The *child process* can be automatically followed by Pin with the *-follow_execv* option enabled. However, path planning for multiple processes is challenging. Therefore, we leave the multi-path exploration of child processes for future work. Additionally, we manually create child processes from code inside memory regions for analysis, as Pin may fail to follow them. The *injection* is redirected to a honeypot process under analysis, following a recent work [29]. The *self-modifying* is identified by monitoring the write operation to addresses inside the trace, which is the execution granularity of Pin.

5 Evaluation

We will now demonstrate the efficacy of Antitoxin in countering evasive techniques, such as logic bombs and unsupported techniques. Firstly, we evaluated different path-planning strategies on validation samples. Subsequently, we conducted experiments on a set of representative evasive samples from security vendors and blogs, illustrating how path planning and active anti-evasion countermeasures effectively dismantle evasive techniques.

5.1 Experiment setup

We deployed our analysis framework on both VMware and VirtualBox, using a Windows 7 64-bit/32-bit operating system, 4 CPU cores, and 8 GB of RAM. To avoid missing sensitive behaviors, we set a timer of 10 min for each execution. Although most samples exhibit evasive behaviors within the first 2 min [33], complex samples may perform more checks and behaviors. Additionally, it is worth noting that DBI and callbacks may incur overhead.

5.2 Validation

Validation sample. This is a validation sample that includes multiple evasive techniques, such as logic bombs and Command & Control (C2) that cannot be dismantled in advance. In this case study, we demonstrate the implementations of evasive techniques and evaluate the effectiveness of probability, taints, and active anti-evasion countermeasures in dismantling evasion. The simplified code for the sample is shown in Figure 5. It starts with a time-based logic bomb and disguises itself as benign software for querying the weather (lines 5–9), followed by a language-based logic bomb (lines 11–13). Next, it performs a series of checks for timing (lines 16–21), hardware, and software (line 22). The check for the number of recently used files (lines 26–32) is an unsupported technique used to evaluate multi-path exploration. Finally, malicious behaviors are triggered through C2 (lines 35–44). In our experiment, the remote server emulates the response of the "systeminfo" command (lines 39–41) to simulate cases that cannot be naturally covered by other commands in the analysis (lines 36–38, 42–44).

We executed the validation sample with Antitoxin and recorded the rounds of multi-path exploration when bypassing the evasive techniques, as shown in Table 7. Columns 1 and 2 list the different evasive

```
int main(int argc, const char **argv){
  //Logic bombs
  //Time-based logic bomb remaining dormant before 2024
  SYSTEMTIME time;
  GetSystemTime(&time);
  if (time.wYear!=2024) {
    Query_Wethear(argv[1]);
    terminate();//Cloaking itself as a benign software
  }
  //Language-based logic bomb targeting French
  HKL layout = GetKeyboardLayout(0);
  if (LOWORD(layout)!=0x040c)
    terminate();
  //Evasive techniques that can be dismantled by active anti-evasion
  //Timing check
  ULONGLONG uptime = GetTickCount64();
  LARGE_INTEGER delay;
  delay.QuadPart = -1000*100000;
  NtDelayExecution(&delay);
  if (GetTickCount64()-uptime<10)
    terminate();
  ...//Other checks related to hardware, software, and timing
  //Unsurppoted technique
  //Check if number of files in recent folder is less than 50
  int numberOfFiles = 0;
  HANDLE hFind = FindFirstFileW(recentPath);
  if (hFind!=INVALID_HANDLE_VALUE){
  do{
    numberOfFiles++;
  } while (FindNextFileW(hFind));
  if (numberOfFiles<50)
    terminate();
  ...
  //C2
  recv(Socket, command, sizeof(command),0)
  if(command=="download"){
    ...
  }
  else if(command=="systeminfo"){
    ...
  }
  else if(command=="screenshot"){
    ...
  }
}
```

Figure 5. Simplified code of validation sample

techniques. Columns 3–5 show the rounds of Antitoxin, ACS, and linear exploration. Among them, Antitoxin is guided by probability and taint, and the ACS algorithm is guided by probability alone. The comparison between Antitoxin and ACS (columns 3 and 4) demonstrates the effect of taint, which significantly accelerates the exploration of logic bombs (2, 4 rounds *vs.* 31, 32 rounds). The comparison between ACS and linear exploration (columns 4 and 5) demonstrates the effect of probability, which generically guides the exploration to cover all evasion-related branches faster than linear exploration (81 rounds *vs.* 120 rounds). Additionally, active anti-evasion countermeasures dismantled the checks for timing, hardware, and software, preventing any increase in the number of rounds required to bypass them.

Linear exploration slightly outperforms Antitoxin for file-number check and "screenshot" command. This is because the validation sample terminates execution as soon as the check fails, and linear exploration traverses the control flow from back to front, thus selecting the key branches faster. However, real-world samples have more complex control flows and may perform behaviors after the check. Therefore, Antitoxin outperforms linear exploration when exploring the "download" command. Furthermore,

Table 7. Validation sample results

Category	Evasive technique	Results (rounds)		
		Antitoxin	ACS	Linear
Time-based logic bomb	Checking the time	**2**	5	31
Language-based logic bomb	Checking the language	**4**	17	32
Timing	Checking the time interval of behaviors	4	17	32
Hardware	Checking the number of processors	4	17	32
	Checking the size of RAM	4	17	32
	Checking for specific MAC addresses	4	17	32
Software	Checking for specific DLLs	4	17	32
Unsupported technique	Checking the number of files in specific folder	43	60	**34**
	C2 "download": Downloading payload from http server	62	85	120
	C2 "systeminfo": Sending the information of victim to the server	43	60	34
	C2 "screenshot": Taking a screenshot and send to the server	**81**	83	60
Total		**81**	85	120

Table 8. Pafish results

Category	Evasive technique	Results (rounds)	
		Antitoxin	Linear
Unsupported technique	Checking the click event	**7**	71
	Checking the double-click event	7	71
	Checking the dialog event	**13**	74
	Checking the dialog event and the position of cursor	113	75
Total		113	**75**

the path planning selects branches affected by a logic-bomb-related technique, *GetSystemTimeAsFileTime*, which is used to calculate security cookies during initialization. Comparing columns 3 and 4, we can see that taints speed up the exploration of logic bombs at first, but exploring branches related to security cookies slows down the overall progress.

Pafish. It is an open-source tool for detecting analysis environments. We tested multi-path exploration against its Reverse Turning Test (RTT), while other evasive techniques have been dismantled by active anti-evasion countermeasures. Table 8 shows the rounds of bypassing different checks.

Multi-path exploration bypass two checks for mouse-click events in the 7th round with the guidance of probability, while linear exploration takes 71 rounds. The two checks register different event hooks to set the check result but share the same code for returning results. Instead of switching branches of the hooks, multi-path exploration directly switches the branch for returning different results to bypass the check without prior knowledge of conditions or implementations.

However, in the case of two checks for dialog events, they are bypassed in the 13th and 113th rounds, respectively, while linear exploration takes 74 and 75 rounds. These checks also register hooks and return results, and the result of the second check is additionally affected by a mouse-event hook. Linear exploration is faster because Pafish is a detection tool that prints the result rather than leading to different behaviors after evasive techniques. However, Antitoxin is designed to explore different paths based on probability. Since the exploration of the first dialog check has covered the branch bypassing the check in the shared code, the result could lead to malicious behaviors in real-world samples.

In conclusion, Antitoxin proves to be effective in dismantling evasive malware. The use of active anti-evasion countermeasures helps to save on exploration overhead, while the use of taints accelerates the exploration of logic bombs. Moreover, even in the absence of these two speed-up strategies, probability

```
01 //Check for debuggers
02 NtQuerySystemInformation(0x23, &Info,-)
03 if (Info.DebuggerEnabled)
04   terminate();
05 //Check for sandbox modules
06 if (GetModuleHandleA("sbie.dll")||GetModuleHandleA("aswhook")
07   ||GetModuleHandleA("snxhk"))
08   terminate();
09 //Open and enumerate subkeys, check for vm registries
10 if (check_regkey("REGISTRY\MACHINE\System\CurrentControlSet\Enum\IDE")
11   ||check_regkey("\REGISTRY\MACHINE\System\CurrentControlSet\Enum\SCSI"))
12   terminate();
13 //Check for processes (qemu-ga.exe, qga.exe, ...)
14 NtQuerySystemInformation(0x5, &Info,-)
15   if (check_process(Info))
16     terminate();
17 //Check for modules exist (vmci.s, vmusbm, ...)
18 NtQuerySystemInformation(0xb, &Info,-)
19 if (check_module(Info))
20   terminate();
```

Figure 6. Simplified code of Chaos

Table 9. Comparision of active anti-evasion countermeasures

Category	Evasive technique	BluePill	Brioscia	Pepper	Antitoxin
Running status	NtQuerySystemInformation(0x23)	DBI does not affect dubugging status			
Module	GetModuleHandle			✓	✓
Device	NtEnumerateKey(0x0, Buffer)	✓	✓		✓
	Null-pointer check of NtEnumerateKey		Unknown		✓
Process	NtQuerySystemInformation(0x5)	✓	✓	✓	✓
Module	NtQuerySystemInformation(0xb)	✓			✓

alone can effectively guide multi-path exploration. While linear exploration has an advantage in dismantling evasive techniques that terminate immediately when checks fail, path planning guided by probability is more efficient when exploring complex samples.

5.3 Active anti-evasion

Antitoxin dismantles most known evasive techniques with the active anti-evasion module, rather than exploring multiple rounds to reduce exploration overhead.

Chaos (*MD5:ba01f27b54d8db54c4ce4ece800a8d6c*)**.** It is a highly obfuscated sample. Figure 6 shows its simplified logic. It performs multiple checks for debugging status (lines 2 and 3), sandbox-related and VM-related modules (lines 6, 7 and 18, 19), processes (lines 14 and 15), and registries (lines 10 and 11). After all the checks, it injects the real payload into the process *explorer.exe*.

Regarding evasive techniques equipped by Chaos, Table 9 compares Antitoxin with existing works, including BluePill [8], Brioscia [9], and Pepper [29].

First, only Pepper and Antitoxin dismantle the module check through *GetModuleHandleA*. Moreover, Antitoxin collects more blacklist values and supports the wild-card version *GetModuleHandleW*. Second, all works identify and dismantle the device check implemented by querying the registry, except for Pepper, which only records the behavior. However, Chaos invokes *NtEnumerateKey* twice to query the size of the buffer storing information and acquire the information accordingly. Thus, the buffer is empty in the first invocation, which causes a memory exception when being parsed. Antitoxin adds a null-pointer check before parsing, avoiding this problem. Finally, BluePill and Antitoxin dismantle another module check through *NtQuerySystemInformation*. However, BluePill only includes "VBox" in the blacklist, while Antitoxin reuses the abovementioned blacklist of modules.

```
01 SYSTEMTIME time;
02 //orimodulePath is a hard-coded string initially set to NULL
03 char* orimodulePath = GetOriModulePath();
04 char* modulePath = GetModuleFileName();
05 char* dfenghPath = GetTempDir() + "dfengh.exe";
06 GetSystemTime(time);
07 if(time.wMonth!=9)
08   terminate();
09 if (orimodulePath && !exist_file(orimodulePath))
10   terminate();
11 //"C:\Users\Administrator\AppData\Local\Temp\dfengh.exe"
12 if (!CompareString(modulePath, dfenghPath)) {
13   //Set orimodulePath to the path of September
14   WriteOriModulePath(modulePath);
15   CopyFile(modulePath, dfenghPath);
16   CreateProcess(dfenghPath);
17 }
18 else{
19   DeleteFile(orimodulePath)//The path of September
20   download("gov-l.com/go/da.exe");
21 }
```

Figure 7. Simplified code of September

Table 10. September results

Category	Evasive technique	Results (rounds)		
		Antitoxin	ACS	Linear
Time-based logic bomb	Checking the time	**1**	11	7

In this experiment, Antitoxin successfully tackles static obfuscation and bypasses multiple checks. In comparison to existing works, Antitoxin performs null-pointer checks on countermeasures and establishes more comprehensive blacklists, making it more robust.

5.4 Exploring logic bombs

To defeat logic bombs based on known techniques that use uncertain conditions that could not be dismantled in advance, we employ taint to uncover branches related to sensitive information and explore them preferentially. We provide a case study to demonstrate how taints can accelerate path planning.

September (*MD5: 9f68ae8267182bf1be4e5bb6c75022b8*)**.** It is a time-aware Zbot downloader that stays dormant except in September. Figure 7 shows its simplified code snippet. It dynamically resolves the address of *GetSystemTime* and invokes the API to obtain the time. Then the member wMonth of the `SYSTEMTIME` struct is compared to 9 (line 4). If the value equals 9, the sample copies itself to the %TEMP% folder and executes it (lines 16 and 17). Otherwise, the sample terminates to evade analysis. Table 10 shows the number of rounds needed to bypass the logic bomb.

Without the guidance of tainted information, linear exploration, and probability-guided ACS require 7 and 11 rounds, respectively. Similar to the validation sample, the ACS is slightly slower because the sample immediately terminates execution when the condition is not satisfied, and linear exploration traverses the control flow simply from back to front. However, complex evasive samples will continue to perform some behaviors after checks, and the linear efficiency will be significantly affected due to the exploration of a large number of non-critical branches. ACS, on the other hand, can exclude branches generically leading to duplicate behaviors.

By contrast, Antitoxin, guided by taints, requires only one round to bypass the logic bomb. This is because Antitoxin marks the `SYSTEMTIME` data structure, enabling the identification and prioritization of the branch (line 7) responsible for checking time and altering control flow during path planning. Moreover, unlike the validation sample, this sample doesn't generate security cookies, eliminating any unnecessary exploration overhead.

Table 11. Comparison of dynamic executable regions

Category	Technique	BluePill	Brioscia	Pepper	Antitoxin
Allocated memory	Executing code in memory regions			✓	✓
Child process	Creating child processes	✓	✓	✓	✓

Dfength. This is the sample that September drops after bypassing the logic bomb. When we examine the logs recorded by Antitoxin for both September and Dfengh, we observe that the two samples behave identically up to a string comparison in memory, as illustrated in Figure 7. In conjunction with static analysis, we find that the only difference is a hard-coded string (lines 2 and 3). Antitoxin has parsed API related to strings, which reveals that the module path is compared with the path of Dfengh. Then, September releases Dfengh (lines 13–16), while Dfeng further performs malicious behaviors (lines 19 and 20). First, Dfengh calls *DeleteFile* to delete the original September sample using the hard-coded string (line 19), and then it downloads the next-stage payload through *InternetConnect* and *HttpOpenRequest* (line 20). Antitoxin captures and parses all of these behaviors.

In the experiments on September and Dfengh, Antitoxin efficiently explores the time-based logic bomb with the guidance of taints. As shown in Table 11, Antitoxin monitors a wider range of executable regions than related works, including allocated memory and child processes. Although related works mentioned the use of *-follow_execv* option of Pin, our experiments show the option fails to track child processes created from the allocated memory, which has not been previously mentioned.

5.5 Exploring unsupported techniques

Complex samples often use multiple evasive techniques, some of which are not supported or even unknown to analysis tools. Antitoxin uses probability-guided multi-path exploration to overcome these challenges. Once the diversion of execution is detected, the behavior logs can assist in identifying unsupported techniques. Additionally, by adding new taints to related environmental information, their usage can be revealed, allowing for the extension of active anti-evasion countermeasures or new taints to accelerate multi-path exploration. In the following case, we demonstrate in detail how Antitoxin assists in the dissection of unsupported techniques.

Furtim (*MD5:564ac87ca4114edd6a84a005092f1285*)**.** It is a sophisticated trojan that was first spotted in 2016 and employs 400 rigorous evasive checks, evading known sandboxes except for the bare-metal environment of Joe Security. In a thorough and detailed analysis of Furtim, Joseph [34] identified a number of evasions related to CPU, files, processes, DLLs, registries, devices, and window texts. Additionally, D'Elia *et al.* [8] pointed out that Direct3D was used for detection. After bypassing all the checks, Furtim drops *puntosw.exe* to the startup directory. We explore the last evasive technique of the complex sample, Furtim, to demonstrate Antitoxin's ability to defeat unsupported techniques. The simplified code snippet is shown in Figure 8. Furtim obtains the device ID (lines 6 and 7) and extracts the vendor ID (line 8). Then, it uses a blacklist to match VM-related IDs and terminates execution if there is a match (lines 10 and 11). Otherwise, it uses a whitelist to match machine IDs and ends the check if there is a match. Finally, if the ID does not belong to either list, Furtim performs a mouse-movement check (line 15).

Furtim uses function pointers to obfuscate APIs and encrypts the `.data` section to hinder static analysis. However, Antitoxin can dismantle and log checks for hardware, including CPU and device name, as well as software, including file, DLL, and debugging status. The information is then used to match blacklists related to analysis environments through functions such as *wcscmp*, *CompareString*, and *RtlCompareUnicodeString*. In particular, a large number of files related to virtual machines, sandboxes, and analysis tools are checked through *NtQueryAttributesFile*. The strings parsed by Antitoxin help expand countermeasures for unsupported techniques. We now examine the last check used by Furtim.

Next, we focus on the invocation of *CreateProcess* used to drop *puntosw.exe* and compare the number of rounds required for different exploration methods. Note our early experiments need to bypass all the different checks to trigger the behavior, but Furtim has several branches that can bypass the check without knowing the implementations of evasive techniques. Even if the environment information is not modified, exploration can still bypass the checks by switching branches of other stages such as initialization. First,

```
01 void Direct3D_AdapterCheck(){
02   DISPLAY_DEVICE DisplayDeviceM;
03   int monitorNum = get_monitornum();
04   int monitorIndex = 0;
05   while (monitorIndex<monitorNum) {
06     if (Enum_monitor(monitorIndex, &DisplayDeviceM, _)) {
07       deviceid = DisplayDeviceM.DeviceId;
08       int vendorid = GetVendorIDfromDeviceID(deviceid);
09       //check if vendorid in blacklist, if so, terminate immediately
10       if (vendorid==0x15AD || vendorid==0x80EE || vendorid==0x1513)
11         terminate();
12       //check if vendorid is not zero and in whitelist, if not, check cursor behavior
13       if (vendorid==0x8086 || vendorid==0x10DE || vendorid==0x1002 || vendorid==0)
14         break;
15       else check_cursor();
16     }
17   }
18   monitorIndex++;
19 }
```

Figure 8. Simplified code of Furtim

Table 12. Furtim results

Evasive technique	Switching	Results (rounds)			
		ACS	ACS (Rerun)	Linear	Adding taints
Obtaining the address of API	Obtaining fails	35	22		
Creating an object through Direct3DCreate9	Creating fails	7		291	
Obtaining the number of devices through API_1	The number is zero				
Obtaining the vendor ID through API_2	Obtaining fails		**4**		
Matching the whitelist of ID	The ID belongs to whitelist				12

Furtim obtains the address of *Direct3DCreate9* from *d3d9.dll* and then creates a querying object to invoke its interfaces. Next, an anonymous API_1 obtains the number of display devices. Then, a loop repeatedly queries the vendor ID of each display device through API_2 and tries to match predefined lists. Specifically, it terminates execution if the ID matches a blacklist and the check will be passed when matching a whitelist. To ensure logical rationality, return values are checked for validity when obtaining the API address, invoking *Direct3DCreate9*, API_1, and API_2. Thus, multi-path exploration can bypass the whole check by switching any branch related to these validity checks, as shown in Table 12. Columns 1 and 2 show different branches used to bypass the check. Columns 3–5 list the rounds needed for taint-guided ACS and linear exploration. Columns 3 and 4 result from two independent analyses that select different branches because ACS introduces roulette for randomness in branch selection besides probability priority. Column 6 lists the rounds after actively adding taints, which we will explain in detail later.

In column 3, ACS selects the branch related to *Direct3DCreate9* in the 7th round and the branch of obtaining API to bypass the check in the 35th, both triggering malicious behaviors. Additionally, in column 4, another analysis using ACS selects the branch related to API_2 in the 4th round and the branch of obtaining API in the 22nd round. In contrast, linear exploration requires 291 rounds to bypass the check of creating the object due to the complex control flow of the sample. It's important to note that bypassing any of the checks in column 1 will trigger malicious behaviors, specifically in the 7th round in column 3 and the 4th round in column 4, so it is not necessary to cover all the branches to achieve high coverage.

In addition to multi-path exploration, Antitoxin can also assist in dissecting evasive techniques through taint analysis and behavior logs. Combined with the findings of D'Elia *et al.* [8] and the logs of Antitoxin, it is found that the anonymous function API_2, which obtains the vendor ID, leverages

NtUserEnumDisplayDevices to obtain the information about display devices. Additionally, *RtlUnicodeToMultiByteN* and its underlying implementation, *WCSToMBEx*, convert the information from wide-char strings to ANSI strings. Based on this discovery, we add taints to the environment information obtained by *NtUserEnumDisplayDevices* to guide path planning. As shown in column 6, with the guidance of taints, the branch related to matching the whitelist is switched in the 12th round. Although taint-guided exploration takes more rounds, it provides comprehensive information about the evasive technique. With the help of the whitelist we discovered, a new countermeasure can be implemented by modifying the obtained vendor ID to a value in the whitelist.

In this experiment, Antitoxin explores Furtim using probability-based path planning, demonstrating its efficiency in exploring complex control flow. Moreover, the results show that multi-path exploration can overcome unsupported evasive techniques by switching between different branches without prior knowledge of their conditions or implementations. In addition, taints can be added to environmental information based on previous experience and experiments to help dissect unsupported techniques by tracing obtained information and identifying affected control-flow branches. Finally, it is feasible to propose active anti-evasion countermeasures, or retain taints, to guide the exploration of logic bombs.

In summary, the experimental results show that the probability calculated from coverage can effectively guide multi-path exploration against evasive techniques, including unsupported techniques, and achieve better efficiency than linear exploration for complex control flow. Furthermore, the control flow and behavior information recorded by Antitoxin can assist in dissecting new evasive techniques. Based on existing knowledge, we can add taints to accelerate the exploration of logic bombs and deploy active anti-evasion countermeasures to reduce exploration overhead. In addition, Antitoxin employs a series of optimization methods, such as monitoring allocated memory and child processes, which can defeat more complex malicious samples.

6 Conclusion and future work

In this paper, we propose a method using multi-path exploration to generically dismantle evasions lacking prior knowledge of conditions and implementations, particularly logic bombs and unsupported evasive techniques. To accelerate the multi-path exploration, we introduce a path-planning strategy to explore the most evasion-related paths, which effectively explores logic bombs and unsupported evasive techniques with the guidance of taint analysis and probability calculation. We apply the proposed prototype, Antitoxin, to a set of samples that contain multiple evasive techniques. The experimental results demonstrate that Antitoxin can generically identify evasion-related branches and successfully trigger hidden behaviors with fewer rounds. Furthermore, Antitoxin proves to be significantly more efficient than linear exploration methods for complex control flows.

Then, we list some open problems and future work to further improve the method's efficiency.

Taint analysis. Antitoxin accelerates the exploration of logic bombs by adding taints to environmental information obtained by specific evasive techniques, which may result in false positives. Some techniques are used for normal purposes, such as the invocation of *GetSystemTime* for initializing security cookies, which Antitoxin cannot distinguish. Nevertheless, these taints can be excluded within limited rounds because they lead to fewer paths than evasive techniques. We should exclude these normal behaviors based on dynamic execution information in the future.

Furthermore, Antitoxin adds taints based on existing knowledge, which means that we cannot explore unknown techniques through taint-guided exploration. We only monitor a small list of techniques to avoid false positives, and analysts can extend the list of techniques based on their findings. However, the extension needed is not endless and may not cover the majority of logic bombs. In addition, Antitoxin provides probability-guided exploration and behavior logs that could help identify unsupported techniques. In the future, we should aim to dynamically identify logic bombs and automatically add taints to guide multi-path exploration.

Multi-path exploration. Antitoxin is capable of monitoring and applying anti-evasion countermeasures to various regions related to the analyzed sample, including dynamically allocated memory, suspicious DLLs, and child processes. However, exploring these regions together with the main module presents a challenge due to dynamic addresses that vary across each execution. To overcome this hurdle, we can separate DLLs and child processes as modules and calculate fixed offsets. Additionally, we should consider

the control flow in memory regions, such that a unique mark is generated using the offset from the base address and the instruction address where the region is allocated.

Finally, Antitoxin only switches direct conditional branches. X-Force utilizes IDA Pro to identify indirect jumps and calls, while Antitoxin collects coverage information through Pin's interfaces during execution, avoiding the need for static analysis. In future work, we should aim to dynamically identify jump tables and support switching to multiple targets.

Conflict of Interest

The authors declare no conflict of interest.

Data Availability

No data are associated with this article.

Authors' Contributions

Fangzhou Xu wrote and constructed this paper. Wang Zhang mainly implemented the path planning algorithm and jointly conducted the experiments. Weizhong Qiang guided the overall work, corrected typos, and jointly wrote this paper. Hai Jin supervised the overall work.

Acknowledgements

We thank the anonymous reviewers for their helpful comments.

Funding

This work was supported in part by the National Natural Science Foundation of China (Grant No. 62272181).

References

[1] AV-TEST. Malware Statistics & Trends Report. https://www.av-test.org/en/statistics/malware/ (January 2023).

[2] Moser A, Kruegel C and Kirda E, Limits of static analysis for malware detection. In: The 23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10–14, 2007, Miami Beach, FL, USA, 2007, 421–430, doi: 10.1109/ACSAC.2007.21.

[3] Aghakhani H, Gritti F and Mecca F et al. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In: Network and Distributed Systems Security (NDSS) Symposium 2020. 2020, doi: 10.14722/ndss.2020.24310.

[4] Ji T, Fang B and Cui X et al. Framework for understanding intention-unbreakable malware. Sci Chin Inf Sci 2023; **66**: 142104.

[5] Chen X, Andersen J and Mao ZM et al. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24–27, 2008, Anchorage, Alaska, USA. IEEE Computer Society, 2008, 177–86.

[6] Polino M, Continella A and Mariani S et al. Measuring and defeating anti-instrumentation-equipped malware. In: Polychronakis M and Meier M (eds.). Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2017. Lecture Notes in Computer Science. Vol. 10327. Cham: Springer, 2017.

[7] D'Elia DC, Coppa E and Nicchi S et al. SoK: using dynamic binary instrumentation for security (and how you may get caught red handed). In: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. ACM, 2019, 15–27.

[8] D'Elia DC, Coppa E and Palmaro F et al. On the dissection of evasive malware. IEEE Trans Inf Forensics Secur 2020; **15**: 2750–65.

[9] Galloro N, Polino M and Carminati M et al. A systematical and longitudinal study of evasive behaviors in windows malware. Comput Secur 2022; **113**: 102550.

[10] Afianian A, Niksefat S and Sadeghiyan B et al. Malware dynamic analysis evasion techniques: a survey. ACM Comput Surv 2019; **52**: 1–28 .

[11] Moser A, Kruegel C and Kirda E, Exploring multiple execution paths for malware analysis. In: 2007 IEEE Symposium on Security and Privacy (SP'07). IEEE, 2007.

[12] Peng F, Deng Z and Zhang X et al. X-force: force-executing binary programs for security applications. In: 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA. 2014.

[13] You W, Zhang Z and Kwon Y et al., Pmp: cost-effective forced execution with probabilistic memory pre-planning. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020, 1121–38.

[14] Intel. Pin – A Dynamic Binary Instrumentation Tool. www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html (January 2023).

[15] Branco RR, Barbosa GN and Neto PD. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Black Hat 2012; **1**: 1–27.

[16] Chenke L, Feng Y and Qiyuan G et al. Anti-reverse-engineering tool of executable files on the windows platform. In: 2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC). Vol. 1. IEEE, 2017, 797–800.

[17] Cha SK, Avgerinos T and Rebert A et al. Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy. IEEE, 2012, 380–94.

[18] Chipounov V, Kuznetsov V and Candea G. S2E: a platform for in-vivo multi-path analysis of software systems. Acm Sigplan Notices 2011; **46**: 265–78.
[19] Saudel F and Salwan J, Triton: a dynamic symbolic execution framework. In: Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes. 2015, 31–54.
[20] Shoshitaishvili Y, Wang R and Salls C et al. Sok: (state of) the art of war: offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016, 138–57.
[21] Böhme M, Pham VT and Roychoudhury A. Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016, 1032–43.
[22] Zhao L, Duan Y and Yin H et al. Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing. In: Network and Distributed Systems Security (NDSS) Symposium 2019. 2019, doi: 10.14722/ndss.2019.23504.
[23] Sebastio S, Baranov E and Biondi F et al. Optimizing symbolic execution for malware behavior classification. Comput Secur 2020; **93**: 101775.
[24] Wang X, Yang Y and Zhu S. Automated hybrid analysis of android malware through augmenting fuzzing with forced execution. IEEE Trans Mobile Comput 2019; **18**: 2768–82.
[25] Park K, Sahin B and Chen Y et al. Identifying behavior dispatchers for malware analysis. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. ACM, 2021, 759–73.
[26] Wang Y, Jia X and Liu Y et al. Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23–26, 2020. The Internet Society, 2020.
[27] Xu Z, Zhang J and Gu G et al. Autovac: automatically extracting system resource constraints and generating vaccines for malware immunization. In: 2013 IEEE 33rd International Conference on Distributed Computing Systems. IEEE, 2013, 112–23.
[28] Schwartz EJ, Avgerinos T and Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on Security and Privacy. IEEE, 2010, 317–31.
[29] Maffia L, Nisi D and Kotzias P et al. Longitudinal study of the prevalence of malware evasive techniques. CoRR. Preprint arXiv:2112.11289 (2021).
[30] Kemerlis VP, Portokalidis G and Jee K et al. Libdft: practical dynamic data flow tracking for commodity systems. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments. ACM, 2012, 121–32.
[31] Dorigo M and Gambardella LM, Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Trans Evol Comput 1997; **1**: 53–66.
[32] Goldberg DE, Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison Wesley, 1989.
[33] Küchler A, Mantovani A and Han Y et al. Does every second count? Time-based evolution of malware behavior in sandboxes. In: Proceedings 2021 Network and Distributed System Security Symposium. 2021.
[34] Landry J. Malware Discovered – SFG: Furtim Malware Analysis. https://www.sentinelone.com/blog/sfg-furtims-parent/ (January 2016).

Fangzhou Xu is currently a master's student in cyberspace security at Huazhong University of Science and Technology (HUST), Wuhan, China. He received a B.E. degree in information security from HUST, Wuhan, China, in 2020. His research interests include malware analysis and evasion detection.

Wang Zhang is currently a master's student in cyberspace security at Huazhong University of Science and Technology (HUST), Wuhan, China. He received a B.E. degree in information security from HUST, Wuhan, China, in 2021. His research interests include adversarial malware detection and encrypted traffic classification.

Weizhong Qiang received a Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, in 2005. He is a professor at HUST. His topics of research interests include system security about virtualization and cloud computing.

Hai Jin received a Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994. He is a Cheung Kung Scholars Chair Professor of computer science and engineering with HUST. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.