GOLDENEYE: Efficiently and Effectively Unveiling Malware's Targeted Environment

Zhaoyan Xu[1], Jialong Zhang[1], Guofei Gu[1], and Zhiqiang Lin[2]

[1]Texas A&M University, College Station, TX
{z0x0427, jialong, guofei}@cse.tamu.edu
[2]The University of Texas at Dallas, Richardson, TX
zhiqiang.lin@utdallas.edu

Abstract. A critical challenge when combating malware threat is *how to efficiently and effectively identify the targeted victim's environment*, given an unknown malware sample. Unfortunately, existing malware analysis techniques either use a limited, fixed set of analysis environments (not effective) or employ expensive, time-consuming multi-path exploration (not efficient), making them not well-suited to solve this challenge. As such, this paper proposes a new dynamic analysis scheme to deal with this problem by applying the concept of speculative execution in this new context. Specifically, by providing multiple dynamically created, parallel, and virtual environment spaces, we speculatively execute a malware sample and adaptively switch to the right environment during the analysis. Interestingly, while our approach appears to trade space for speed, we show that it can actually use less memory space and achieve much higher speed than existing schemes. We have implemented a prototype system, GOLDENEYE, and evaluated it with a large real-world malware dataset. The experimental results show that GOLDENEYE outperforms existing solutions and can effectively and efficiently expose malware's targeted environment, thereby speeding up the analysis in the critical battle against the emerging targeted malware threat.

Keywords: Dynamic Malware Analysis, Speculative Execution

1 Introduction

In the past few years, we have witnessed a new evolution of malware attacks from blindly or randomly attacking all of the Internet machines to targeting only specific systems, with a great deal of diversity among the victims, including government, military, business, education, and civil society networks [17,24]. Through querying the victim environment, such as the version of the operating system, the keyboard layout, or the existence of vulnerable software, malware can precisely determine whether it infects the targeted machine or not. Such *query-then-infect* pattern has been widely employed by emerging malware attacks. As one representative example, advanced persistent threats (APT), a unique category of targeted attacks that sets its goal at a particular individual or organization, are consistently increasing and they have caused massive damage [15]. According to an annual report from Symantec Inc, in 2011 targeted malware has a steady uptrend of over 70% increasing since 2010 [15], such overgrowth has never been slow down, especially for the growth of malware binaries involved in targeted attacks in 2012 [14].

To defeat such massive intrusions, one critical challenge for malware analysis is how to effectively and efficiently expose these environment-sensitive behaviors and in

further derive the specification of environments, especially when we have to handle a large volume of malware corpus everyday. Moreover, in the context of defeating targeted attacks, deriving the malware targeted environment is an indispensable analysis step. If we can derive the environment conditions that trigger malware's malicious behavior, we can promptly send out alerts or patches to the systems that satisfy these conditions.

In this paper, we focus on *environment-targeted malware*, i.e., malware that contains *query-then-infect* features. To analyze such malware and extract the specification of their targeted environment, we have to refactor our existing malware analysis infrastructure, especially for dynamic malware analysis. Because of the limitation of static analysis [38], dynamic malware analysis is recognized as one of the most effective solutions for exposing malicious behaviors [38,37]. However, existing dynamic analysis techniques are not effective and efficient enough, and, as mentioned, we are facing two new challenges: First, we need *highly efficient techniques* to handle a great number of environment-targeted malware samples collected every day. Second, we require the analysis environment to be more *adaptive* to each individual sample since malware may only exhibit its malicious intent in its targeted environment. (More details are explained in Section 2.)

As such, in this paper we attempt to fill the aforementioned gaps. Specifically, we present a novel dynamic analysis scheme, GOLDENEYE, for agile and effective malware targeted environment analysis. To serve as an efficient tool for malware analysts, GOLDENEYE is able to *proactively* capture malware's environment-sensitive behaviors in progressive running, *dynamically* determine the malware's possible targeted environments, and *online* switch its system environment *adaptively* for further analysis.

The key idea is that by providing several dynamic, parallel, virtual environment spaces during a single malware execution, GOLDENEYE proactively determines what the malware's targeted environment is through a specially designed speculative execution engine to observe malware behaviors under alternative environments. Moreover, GOLDENEYE dynamically, adaptively switches the analysis environment and lets malware itself expose its target-environment-dependent behaviors. Although GOLDENEYE trades space for speed, interestingly our experimental results show that GOLDENEYE could actually use less memory space while achieving much higher speed than existing multi-path exploration techniques.

In summary, this paper makes the following contributions:

- We present a new scheme for environment-targeted malware analysis that provides a better trade-off between effectiveness and efficiency, an important and highly demanded step beyond existing solutions. As a preliminary effort towards systematic analysis of targeted malware, we hope it will inspire more future research in targeted and advanced persistent threat defense.
- We design and implement GOLDENEYE, a new lightweight dynamic analysis tool for discovering malware's targeted environment by applying novel speculative execution in dynamic, parallel, virtual environment spaces. The proposed approach can facilitate the analysis on new emerging targeted threats to reveal malware's possible high-value targets. Meanwhile, it also facilitates conducting large volumes of malware analysis in a realtime fashion.
- We provide an in-depth evaluation of GOLDENEYE on real-world malware datasets and show that GOLDENEYE can successfully expose malware's environment-sensitive behaviors with much less time or fewer resources, clearly outperforming existing approaches. We also show that GOLDENEYE can automatically identify and provide correct running environment for tested well-known targeted malware families. To further improve the accuracy and efficiency, we also propose a distributed deployment scheme to achieve better parallelization of our analysis.

2 Background and Related Work

2.1 Objectives

The focal point of this paper is on a set of malware families, namely *environment-targeted malware*. In our context, we adopt the same definition of *environment* in related work [36], i.e., we define an *environment* as a system configuration, such as the version of operating system, system language, and the existence of certain system objects, such as file, registry and devices.

Environment-targeted malware families commonly contain some customized environment check logic to identify their targeted victims. Such logic can thus naturally lead us to find out the malware's targeted running environment. For instance, Stuxnet [13], an infamous targeted malware family, embeds a PLC device detection logic to infect machines that connect to PLC control devices. Banking Trojans, such as Zeus [21], only steal information from users who have designated bank accounts. Other well-known examples include Flame [6], Conficker [43] and Duqu [4].

As a result, different from the traditional malware analysis, which mainly focuses on malware's behaviors, environment-targeted malware analysis has to answer the following two questions: (1) Given a random malware binary, can we tell whether this sample is used for environment-targeted attacks? (2) If so, what is its targeted victim or targeted running environment?

Consequently, the goal of our work is to design techniques that can (1) identify possible targeted malware; (2) unveil targeted malware's environment sensitive behaviors; and (3) provide environment information to describe malware's targeted victims.

2.2 Related Work

Research on Enforced/Multi-path Exploration. Exposing malicious behaviors is a research topic that has been extensively discussed in existing research [33,30,27,23,36,47,46].

One brute-forced path exploration scheme, forced execution, was proposed in [46]. Instead of providing semantics information for a path's trigger condition, the technique was designed for brute-force exhausting path space only. Most recently, X-Force [42] has made this approach further by designing a crash-free engine. To provide the semantics of the trigger, Brumley *et al.* [25] proposed an approach that applies taint analysis and symbolic execution to derive the condition of malware's hidden behavior. In [34], Hasten was proposed as an automatic tool to identify malware's stalling code and deviate the execution from it. In [35], Kolbitsch *et al.* proposed a multipath execution scheme for Java-script-based malware. Other research [29,46] proposed techniques to enforce execution of different malware functionalities.

One important work in this domain [37] introduced a snapshot based approach which could be applied to expose malware's environment-sensitive behaviors. However, this approach is not efficient for *large-scale* analysis of environment-targeted malware: it is typically very expensive and it may provide too much unwanted information, thus leaving truly valuable information buried. This approach essentially requires to run the malware multiple times to explore different paths. After each path exploration, we need to rewind to a previous point (e.g., a saved snapshot), deduce the trigger condition of branches and explore unobserved paths by providing a different set of input, or sometimes enforce the executing of branches in a brute-force way. Obviously this kind of frequent forward execution and then rolling back is very resource-consuming, thus making it not very scalable to be applied for analyzing a large volume of malware samples collected each day. Moreover, this scheme is a typical sequential model which

makes the analysis hard for parallel or distributed deployment, e.g., in a cloud computing setting. Last but not least, the possible path explosion problem [37] is another important concern for this approach.

Research on Malware's Environment-Sensitive Behaviors. Another line of research [27,23,28,36,44,40] discusses malware environment-sensitive behaviors. These studies fall into three categories: (1) Analyzing malware's anti-debugging and anti-virtualization logic [23,28]; (2) Discovering malware's different behaviors in different system configurations [36]; (3) Discovering behaviors in network-contained environment [32]. The main idea in these studies is to provide *possible target environments* before applying the traditional dynamic analysis. The possible target environment could be a running environment without debuggers [28], introspection tools [23], or patched vulnerabilities involved.

In a recent representative study [36], the authors provided several statically-configured environments to detect malware's environment sensitive behaviors. While efficient (not carrying the overhead of multi-path exploration), this approach is not effective, i.e., the limitation is: *we cannot predict and enumerate all possible target environments in advance*. In particular, in the case of targeted malware, we often are not able to predict malware's targeted environments before the attack/analysis.

Summary. We summarize the pros and cons of previous research in Table 1. We analyze these techniques from several aspects: *Completeness, Flexibility, Prerequisites, Resource Consumption, Analysis Speed, Assisting Techniques, and Deployment Model*.

Approach Category	I	II
Representative Work	[25,37,46]	[36,23]
Completeness	High	Low
Flexibility	High	Low
Prerequisites	Low	High
Resource Consumption	High	Low
Analysis Speed	Slow	Fast
Assisting Techniques	Symbolic Execution, Tainted Analysis, Execution Snapshot	Trace Comparison
Deployment Model	Sequential	Sequential/Parallel

Table 1: Summary of Existing Techniques

As illustrated, the first category of solution, such as [37,25], has theoretically full-completeness but with high resource consumption. It requires the execution to periodically store execution context and roll back analysis after one-round exploration, thus very slow. Meanwhile, it requires some assisting techniques, such as symbolic execution which is slow and has some inherent limitations [22]. Last but not least, it is not designed for parallel deployment, making it not able to leverage modern computing resources such as clouds.

For the second category, such as [23,36], these approaches support both sequential and parallel deployment. Meanwhile it has less resource consumption and fast analysis speed. However, all the environments require manual expertise knowledge and need to be configured *statically beforehand*. Hence, it is not flexible nor adaptive. More importantly, it is incomplete, limited to these limited number of preconfigured environments, and has a low analysis coverage.

3 Overview of GOLDENEYE

An overview of our approach is presented in Figure 1. As illustrated, our scheme consists of three phases. In phase I, we screen malware corpus and identify the possible targeted malware samples. In phase II, we employ dynamic environment analysis to iteratively unveil the malware candidates' targeted running environments. In phase III, we summarize the analysis result with detailed reports. The reports contain the information about malware's sensitive environments and their corresponding behavior differences.

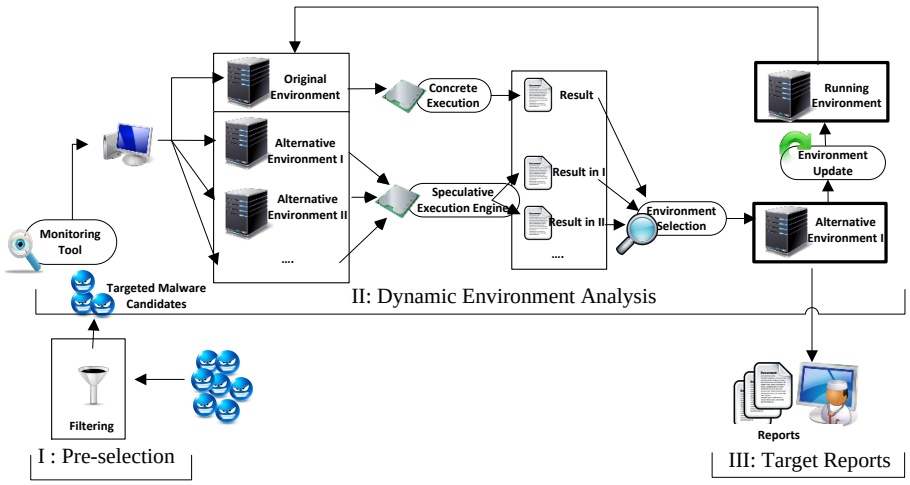

Fig. 1: Overview of GOLDENEYE

In this section we briefly overview the basic idea of our key novel design in GOLDENEYE, i.e., *progressive speculative execution in parallel spaces*, and leave the rest system details to Section 4.

The first key design of GOLDENEYE is to *dynamically* construct parallel spaces to expose malicious behaviors. To overcome the limitation of previous work [36], which statically specifies multiple analysis environments beforehand, our design is to dynamically construct multiple environments based on malware's behaviors, *the call of environment query APIs*. In particular, through labeling these APIs beforehand, we can understand all possible return values of each environment query. For each possible return value, we construct one environment for that. For example, if we find the malware queries system call `GetKeyboardLayout`, we can prepare multiple return values such as 0x0004 for Chinese and 0x0409 for United States, and simulate two parallel running environment with Chinese and English keyboards for analyzing malware behaviors. As shown in Figure 2, the parallel environments is constructed alongside with malware's execution, therefore, it prevents running the same sample by multiple times. As long as our API labeling (introduced in Section 4) can cover the environment query, we believe GOLDENEYE can automatically detect/expose all environment-sensitive behaviors of samples.

Our second novel design is to apply speculative execution in these parallel environments. Observing the limitation of existing work [37], which consumes a huge amount of time and memory on rolling back the analysis on alternative paths, we apply the concept of speculative execution [31], which refers to the situation when a computer system performs some task that may not be actually needed but to trade off some other optimize needs. The merit of applying speculative execution in our context is to keep the execution forward as far as possible. Thus, we consider to construct multiple possible environments online and speculatively execute malware in each environment instance. Through determining the most possible malicious execution path, we can also determine what the running environment is in order to reach certain path.

To embrace speculative execution in our new problem domain, we need to solve new technical challenges. First, since the executed instructions in each environment vary,

it is practically infeasible to predict the execution in an online dynamic fashion. We solve this challenge by *progressively* performing the speculative execution at the *basic block level*. In particular, we execute each basic block in all alternative environment settings. Since there is no branch instruction inside each basic block, the instructions are the same for all environments. When we reach the branch instruction at the end of a block, we apply several heuristics to determine which is the possible malicious path. Consequently, we reduce the space by only keeping the settings that most likely lead to the desired path.

Second, speculative execution is essentially a trade-off scheme between speed and space (i.e., trading more memory consumption for speedup) [31]. In our design, we also try to reduce the memory consumption by two novel designs: (1) We only speculatively execute the instructions that generate different results for different environments. We choose to employ taint analysis to narrow down the scope to the instructions which operate on environment-related data. (2) We monitor the memory usage to prevent the explosion of alternative environments.

In general, we introduce the following speculative execution engine: We conduct speculative execution at the granularity of code block to emulate the malware's execution in multiple parallel environment spaces. We first *prefetch* a block of instructions. Next, we apply taint analysis on the pre-fetched instructions and taint each byte of the API/instruction output (*environment element*) as the tainted source. The reason to use taint analysis is to filter those instructions that are not related to the environment, which can reduce the overhead of full speculative execution. We propagate tainted labels and when we hit one instruction with the tainted operands, we accordingly update the execution context in all alternative environments. We continue such propagation until we reach the end of the block, which is a branch instruction. For the branch instruction, we determine whether it could be affected by the tainted bytes or not. If it is an environment-sensitive branch, we continue to the next step, i.e., branch selection and update. If not, speculative execution will start a new pre-fetch operation.

For environment-sensitive branches, we attempt to prevent the overhead caused by roll-back operation in [37]. We design our scheme to proactively select the branches based on the information provided in the speculative execution. The intuition is: if we can tell which branch is more likely the malware author's intended branch, we can dynamically adjust the environment to enforce the malware to only execute some designated branch. In principle, whenever we find a branch that belongs to a possible malicious path, we will re-examine the alternative environments and only select the environment that could be used to enforce the desired branch. Our solution to find the possible malicious branch is to apply similar techniques as *branch evaluation* [47] to predict the possible malicious branches. The detail will be presented in Section 4.2.

To differentiate GOLDENEYE with other approaches, we illustrate the high-level idea of GOLDENEYE in Figure 2 by comparing with an existing multi-path exploration approach [37]. For the multi-path exploration approach (the left part in Figure 2), the redundant overhead comes from exploring all the possible paths by depth-first execution and storing roll-back snapshots for all deviation branches. GOLDENEYE works in a different way. It applies branch prediction that follows a breath-first execution scheme to quickly locate possible malicious paths, which saves the effort of exploring all possible paths. Second, it enumerates all the possible alternative environments, e.g., ABCD in Figure 2, dynamically. It ensures the analysis continuously keep forward and saves the roll-back operations. Thus, it is not necessary to store snapshots for every branch. Lastly, we use taint analysis to skip many non-environment-sensitive branches to further save the exploration overhead.

Meanwhile, from the figure we can also notice how the speculative execution technique is performed in parallel environments. Essentially, our speculative execution is

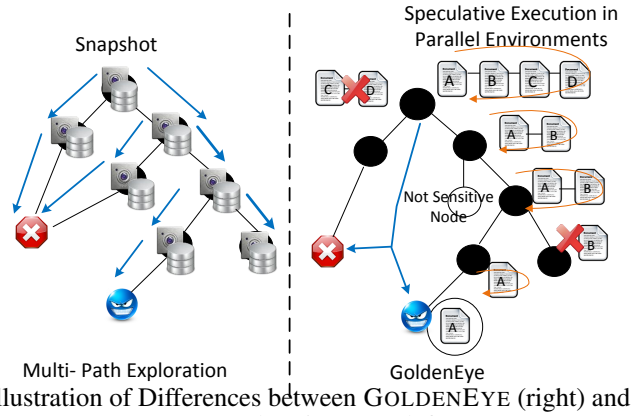

Fig. 2: Illustration of Differences between GOLDENEYE (right) and Multi-path Exploration [37] (left)

periodically progressing for each code block. We need to iterate all the environments and synchronize their running results for each instruction in the code block. At the end of a basic block, the parallel space will be curtailed and GOLDENEYE clears all the environments settings that unlikely lead to targeted paths.

4 Detailed Design

4.1 Phase I: Pre-selection of Malware Corpus

The first phase of GOLDENEYE is to quickly obtain the malware samples which are candidates of environment-targeted malware. As defined in Section 2.1, our criteria for the pre-processing is to find any malware that is sensitive to its running environment.

Our scheme of pre-selection is achieved by tainting the return values of certain environment query API/instructions and tracking whether the tainted bytes affect the decision on some branch instructions, such as changing `CFlag` register. If the tested sample is sensitive to its environment querying, we keep the sample for further steps.

API Labeling. The most common way for malware to query its running environment is through certain system APIs/instructions. To capture malware's environment queries, we need to hook these APIs/instructions. Furthermore, it is important to derive all possible return values of these APIs/instructions because these return values are used to define parallel denvironments. In GOLDENEYE, we label three categories of environment queries:

- *Hook system-level environment query APIs*. The operating system provides a large set of system APIs to allow programmers query the running environment. They have also been commonly used by malware to achieve the similar goal.
- *Hook environment-related instructions*. Some X86 instructions such as `CPUID` can also be thought as a way to query environment information.
- *Hook APIs with environment-related parameter(s)*. Some system files/registries can be used to store environment configuration. Thus, we also hook file/registry operation APIs and examine their parameters. If the parameters contain some keywords, such as `version`, we also treat as a query attempt.

For each labeled API/instruction, we examine its return value as the reference to initialize parallel environments. In general, we construct one speculative execution context

for each possible return value. To narrow down the alternative choices of the environment, we define the following four basic sets of return values.

- *BSET(n)* defines a two-choice (binary) set. One example for `NtOpenFile` is *BSET(0)* for the return value `NTSTATUS`, which accepts 0 (success) or other value (failure).
- *SET([...])* defines a normal enumeration of values, such as enumeration for `LANGID` in the default system language.
- *RANGE(A, B)* set contains a range of possible return values.

Based on these three sets, we construct the parallel contexts. For example, we simply construct two parallel contexts for *BSET(n)* element. Note that a large amount of system objects, whose querying API returns -1 as *non-existence* and *random value* as the *object handle*, belong to this type. We consider all these objects as *BSET(n)* element.

For *SET([...])* with n different values, we accordingly initialize n parallel settings based on the context.

For *RANGE(A, B)* set, we examine whether the range set can be divided into some semantically independent sub-ranges. For example, the different range of native call `NtQuerySystemInformation`'s return specifies different type of the system information. For these cases, we construct one context for each semantically-independent sub-range. Otherwise, we initially construct one context for each possible value.

One current limitation of our labeling is that we cannot provide parallel environments for API functions whose return values are not enumerable. For example, some malware logic may depend on the content of certain file. However, it is not possible for us to construct all possible (correct) file contents in advance. One possible solution is to combine symbolic execution [22] in the analysis step at the cost of extra analysis overhead. However, to achieve better balance between efficiency and effectiveness, we do not adopt such solution at the moment.

4.2 Phase II: Dynamic Environment Analysis

Dynamic environment analysis is the main component of GOLDENEYE. In this section, we present its detailed design. We use Conficker [43] worm's logic as a working example. As illustrated in Figure 3, in this example, Conficker worm queries the existence of specific `mutex` and the version of the running operating system. The malicious logic is triggered only after the check succeeds.

Initialization of Malware Environment Analysis. After the preprocessing, we first initialize the analysis by constructing parallel environments when we find malware's environment query. We define a running environment with a set of environment elements as

$$env = \{e_1, ..., e_i, ...e_n\}$$

For each e_i, it is defined as a tuple:

$$< identifier, API, type, value >$$

where $identifier$ uniquely denotes the name of each environment element, such as the `mutex` name or the title of GUI windows; API is the invoked API to query the element; $type$ specifies the type of element, such as system setting (system language, os version, etc.) or system objects (the existence of files, registries, etc.); and $value$ states *what are possible values of each element*, such as true/false or a set of hex values.

Context Maintenance of Speculative Execution. After GOLDENEYE captures malware's environment query, a set of initialized environment contexts are maintained by our speculative execution engine. The main overhead of our speculative execution comes from continuously maintaining those parallel contexts.

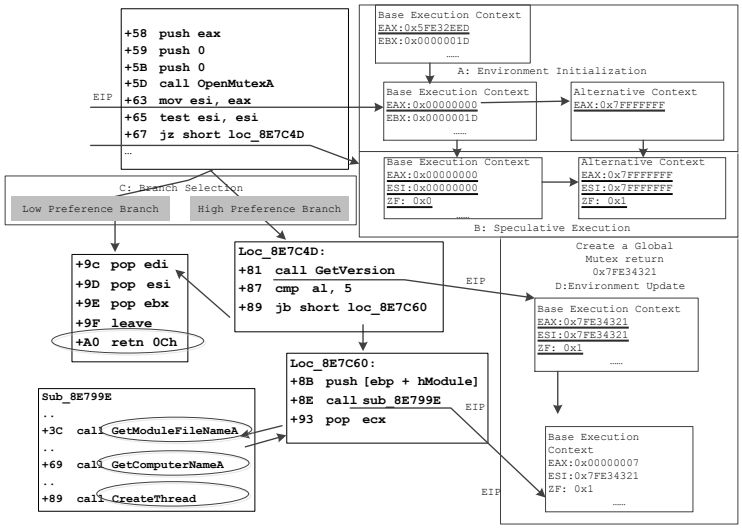

Fig. 3: Working Example of GOLDENEYE

To save space, the key design for context maintenance is based on our progressive execution scheme. Since the execution in parallel can be naturally synchronized by each instruction (it follows the same code block(s)), we choose to only record the modification of parallel contexts. As illustrated in Figure 3 Step A and B, we have no need to maintain the full execution context, such as all general registers value and execution stack, in each parallel space. We only track the different data, which is `EAX` and `ESI` in the example. We maintain such alternative contexts using a linked list. When an environment update operation starts, we only update the *dirty* bytes that have been modified since the previous block(s). In further, we organize each progressive context using linked-list to track the modified bytes.

Taint-assisted Speculative Execution. Another key design to prevent redundant overhead is to applying taint tracking on environment-sensitive data. In particular, we taint each byte of the environment query's return and propagate the tainted labels by each instruction. When we encounter an instruction without tainted operation, we continue with concrete execution. Otherwise, when we encounter an instruction with the tainted operands, we accordingly update the execution context in all alternative environments. We continue such propagation until we reach the end of a basic block. For the branch instruction, we also determine whether it could be affected by the tainted bytes or not (whether `CFlag` has been tainted or not). If it is an environment-sensitive branch, we continue the branch selection and environment update. If not, speculative execution starts a new pre-fetch operation to continue analyzing a new code block.

The advantage of using taint analysis is to efficiently assist the analysis in three ways: (1) Our speculative execution is only conducted on the instructions whose operands have been tainted. It allows us to skip (majority) untainted instruction for speculative execution to save analysis effort. (2) Tainted propagation can help us to determine the environment-sensitive branches. Our environment prediction/selection is based on the correct identification of these sensitive branches. (3) Tracking the status of the tainted label helps us to maintain parallel environment spaces and delete/merge untracked environments.

Heuristics for Branch Selection. Next, we present how we evaluate the branches and determine which branch is more possible in the targeted environment. In GOLDEN-

EYE, we apply three heuristics to determine what is a possible branch in the targeted environment:

- If a branch contains a function call that calls some exit or sleep functions, such as `ExitProcess`, `ExitThread`, and `sleep`, it means this branch may terminate the program's execution. We treat another branch as the possible targeted branch.
- If a branch contains function calls that create a new process or thread, such as `CreateProcess` and `CreateThread`, or start network communication, such as `socket` and `connect`, we treat this branch as the possible targeted branch. Similar function calls could be some representative malicious calls, such as functions for process injection, auto-booting, and kernel hijacking [45].
- If a branch directly interacts with the environment, we treat this branch as the possible targeted branch. For example, if malware creates a file before the branch, we treat the branch that directly operates on the created file as the targeted branch. Essentially, if one branch contains instructions intensively operating on tainted data, we consider it as the targeted branch.

After examining these three heuristics, if we still cannot decide the possible targeted branch in a given time window or we find some conflicts among different heuristics, inspired by the multi-path exploration work [37], we will save the snapshot at the branch point and conduct the concrete execution for both branches. While this may lead to more overhead (as in [37]), our experimental result shows that such cases are very rare (less than 5% cases require rolling back).

Determining Targeted Branch. Our scheme of branch evaluation is to *foresee* k (e.g., $k = 50$) instructions and find whether any of them contains code of our interest. The foreseeing operation is conducted by statically disassemble the code in the blocks after the addresses of two branches. It is gradually processed until we have collected enough evidence for predicting the branch.

In particular, we start with disassembling one code block at a time. We also need to disassemble all the possible branches after each code block. Then we scan each instruction to check whether it (1) has a `CALL` instruction or (2) operates on some tainted data.

For the first case, we need to examine the destination address of `CALL`. Beforehand, we need to maintain two API address lists: the first records the address of possible malicious functions such as `CreateProcess` and `socket`, and the second records the dormant/termination functions such as `sleep` and `ExitProcess`. Thus, if `CALL`'s destination address belongs to either of the lists, we set the corresponding preference to the explored branch.

For the second case, we examine each instruction along two alternative paths to see whether any instruction operates on the tainted data or not. We achieve that by examining the source operands of each instruction. If the source operand is tainted before, we consider the instruction operates on tainted data. Then we deduce the path that contains more instructions using tainted data.

If we cannot make a decision after we examine k instructions, we apply enforced execution [46] to explore both branches. In this case, we need an extra round of analysis for each branch (which is very rare in practice, as shown in our evaluation).

In Figure 3 Step C, we illustrate our strategy by evaluating two branches after the `JZ` instruction. As shown in the left branch, the execution may direct to `leave` and `retn` while the right branch exhibits possible malicious logic, such as `CreateThread`. Then, we choose the right branch and identify the alternative context as our preferred execution context.

Environment Update. The result of target branch prediction is to decide whether to remain in the current running environment or to switch to another alternative environment.

If the environment switching is needed, there are three basic environment switching operations: (1) Creation, (2) Removal, (3) Substitution.

The key requirement of our design is to update the environment online. Hence, our environment update step is performed directly after the speculative execution engine has committed its execution context.

Creating an element is a common case for an environment update. Especially when malware tries to query the existence of certain system object, we would thus create such an object to ensure that the following malware operation on this object will succeed. To this end, we create a dummy object in the system, such as creating a blank file with certain file name or creating a new registry entry. Accordingly, deleting the element is the opposite operation and we can simply achieve that by deleting the corresponding existing system object. While the dummy objects may not always work because fundamentally we may not have the exact same knowledge as malware and its targeted environment to fill the actual content, this scheme works quite well in our evaluation. And we leave a full discussion of GOLDENEYE limitations in Section 7.

The substitution operation usually occurs when malware requires different system configuration from the current running environment. A main approach to find out the correct environment setting is through the result of the speculative execution. Since the speculative execution tells us the condition to ensure the selected branch, we can concretely set up the value to satisfy this requirement. For example, we can modify some registry entries to modify certain software version. As a more generic solution, we design an API manipulation scheme. When a substitution occurs, we hook the previously captured APIs or instructions, and return a manipulated value to malware for every query.

The environment update for our working example is illustrated in Figure 3 Step D. The first step is to update the base execution context as the selected context. In the example, we first update the `ESI` and `ZF` register. Secondly, since `EAX` is the object handle of the `mutex` object, we need to create the `mutex` for current context and bind `EAX` to the `mutex` handle. In our implementation, we do not concretely create the `mutex`. Instead, we record the handle value and when any system call operates on the handle, we enforce the `SUCCESS` to emulate the existence of the object.

Handling Space Explosion. As one notable problem for parallel space maintenance in the speculative execution engine, explosion of parallel spaces could dramatically increase the overhead of GOLDENEYE, especially when the combination of multiple environment elements happens (Cartesian Effect). We solve the problem by periodically pruning the parallel spaces. More specially, we enforce the space adjustment when current memory occupation exceeds some predefined threshold, ρ_h. During the analysis, we associate a timestamp T with each environment element. The time stamp denotes the last instruction that accesses the corresponding taint label of the element. When the current memory usage overflows ρ_h, the speculative execution engine fetches the environment element with the oldest time stamp. Then, the update operation merges all the parallel spaces which have different values for the pruned elements. This process is recursively performed till the current memory capacity is below a predefined lower bound, ρ_l. In practice, among all of our test cases, the average number of concurrent parallel spaces is below 200. It means that, with minor memory overhead (below 500B) for each space, the space pruning rarely occurs in the practical analysis task.

5 Distributed Deployment of GOLDENEYE

While the above scheme works very well in practice (as shown in our experiment), there are still some concerns: (1) To prevent rolling-back, we adopt branch evaluation

to select the most likely malicious branch, which might not always be accurate. (2) Our environment update step is conducted online. Thus, some analysis is possibly conducted on a set of inconsistent environments. (3) The possible environment explosion may overburden one analysis instance.

To further improve the accuracy and efficiency, we propose a distributed deployment scheme of GOLDENEYE. The scheme is essentially taking advantage of parallel environments created by the speculative engine and distributing them to a set of worker machines for further analysis.

In detail, when the speculative engine detects an environment-sensitive branch, it can choose to push a request R into a shared task queue and allow an idle worker (virtual) machine to handle the further exploration. The worker machine monitoring (WMM) tool pulls each request and updates the environment settings before analyzing a malware sample. After the booting of a malware sample, the WMM tool will monitor the execution status and enable the speculative execution if some unobserved malicious logic has occurred.

There are two tasks for each WMM: (1) *Updating analysis environment*, which is a set of operations to update its environment before analaysis, such as *create/delete environment element* or *modify current environment value.* After that, we create one customized environment for each analysis task. (2) *Starting speculative execution*, which is to conduct a series of EIP and basic context registers comparison before restarting the speculative execution. By skipping the instructions which have been analyzed before, we can focus on exploring new malicious behaviors.

The merits of our design are twofold. First, the analysis environment is dynamically changed and the setting is dynamically generated based on the malware execution and analysis progress. It is essentially different from the parallel/distributed deployment of existing analysis techniques [36,23] because their settings are statically preconfigured. Second, it saves a huge amount of system resources including memory and storage. Snapshot-based schemes such as [33,37] are mainly used as sequential execution. If one attempts to parallelize its deployment, a great deal of resources need to be used to store/transmit/restore the snapshots. In our design, each worker machine just maintains *one* initial snapshot locally and consumes little memory to transmit the environment request. In this sense, our scheme achieves a better balance between effectiveness and efficiency.

6 Evaluation

We have implemented GOLDENEYE, which consists of over 4,000 lines mixed C and python code. Our monitoring tool, taint tracking tool, and speculative execution engine are implemented based on the open-source binary instrumentation framework, DynamoRIO [5] by first translating the X86 instructions into an intermediate language BIL [26], then performing data and control flow analysis afterwards. We write our semantic rule module as an independent C library, which receives the output of the monitoring tool and parses each instruction. Our environment selector is based on an open source disassembly library, *distorm*[3]. We also implement a lightweight emulated execution engine inside the module to perform branch evaluation. In addition, our environment update module is implemented as an API hook tool based on DynamoRIO and a set of dummy object creation/deletion scripts, which can be directly invoked by our environment update module. In this section, we present our evaluation results.

6.1 Experiment Dataset

Our test dataset consists of $1,439$ malware samples, collected from multiple online malware repositories such as Anubis [1] and other sources [10]. This dataset is randomly collected without any pre-selection involved. We analyze these $1,439$ malware using a free virus classification tool [20] and classify them into 417 distinct malware families. Analyzing the classification result, we further categorize these 417 malware families into four classes: *Trojan*, *Worm*, *Spyware/Adware*, and *Downloader*. The statistics about our dataset is listed in Table 2. Meanwhile, we also collect a small dataset that includes some well-known malware samples which are environment-targeted, such as Conficker [43], Duqu [4], Sality [12], and Zeus [21]. For each malware family, we collected several variant samples.

Category	# Malware Samples	Percent	Distinct Families
Trojan	627	43.57%	263
Adware/Spyware	284	19.73%	59
Worm/Virus	185	12.85%	27
Downloader	343	23.83%	68
Total	1,439	100%	417

Table 2: Malware's Classification from VirusTotal

6.2 Experiment Setup

In our experiment setting, we manually labeled 112 system/library APIs with 122 output parameters, and hooked them in our analysis. All our experiments are conducted in a machine with Intel Core Duo 2.53GHz processor and 4GB memory.

6.3 Experiments on General Malware Corpus

We conduct the following experiments to evaluate GOLDENEYE on the larger malware dataset with $1,439$ samples.

Measurement of Effectiveness First, we study the effectiveness of our approach in terms of the code coverage in analysis. To measure that, we first collect a baseline trace by naturally running each malware sample in our virtual environment for 5 minutes. Then we apply GOLDENEYE to collect a new trace in the adaptively-changing environment(s). In our evaluation, we measure the relative increase in the number of native system calls between the base run and analysis run. The distribution of increased APIs among all malware samples is shown in Figure 4. As seen in Figure 4, over 500 malware samples exhibit over 50% more APIs in the new run. It shows that our system can expose more malware's environment-sensitive behaviors. From the result, we also find that over 10% Adware/Spyware exhibits 100% more behaviors. It may imply that Spyware is more sensitive to the running environment compared with other malware categories. This is reasonable because Spyware normally exhibits its malicious behavior after it collects enough information about the infected user. This further proves the usefulness of our system. Examining the quantitative results of other categories, it is evident that our system can efficiently discover malware's environment-sensitive functionalities.

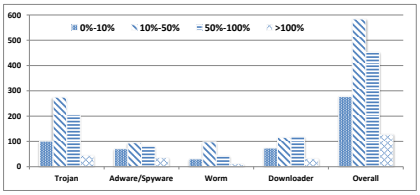

Fig. 4: Relative Increase of Native APIs

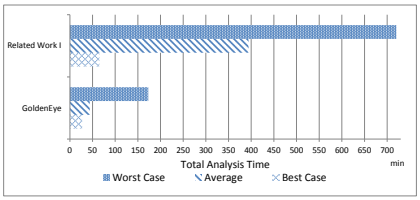

Fig. 5: Analysis Time Comparison

Comparison with Related Work The last set of our experiment is to compare the effectiveness and efficiency of GOLDENEYE with other approaches. To this end, we first implemented the approach presented in the related work [37] (labeled as Related Work I), which needs to explore multiple possible paths of environment-sensitive branches. Secondly, we configure four virtual environments according to the descriptions in related work [36] (labeled as Related Work II). We test malware samples in all four environments and choose the best one as the result. Then we randomly select 100 malware samples from each category of malware and collect the traces generated by GOLDENEYE, Related Work I, and II, respectively. When collecting each execution path trace, we terminate the analysis if no further system calls are observed for 30 seconds (e.g., sample terminates or sleeps), or if it reaches maximum analysis time which we set as 300 seconds (5 minutes) for GOLDENEYE and Related Work II. For Related Work I, since it needs to explore all possible paths, we have to let it run for a much longer time. However, it could possibly take forever. Hence, in this experiment we limit its maximum analysis time to 12 hours.

Approach	Malware	Percent of Increased APIs				# of Rolling Back			Memory/Disk Usage		
		<10%	10%-50%	50%-100%	>100%	<50	50-500	>500	<1MB	1MB-5MB	>5MB
GOLDENEYE	Trojan	31%	36%	27%	6%	74%	26%	0%	67%	33%	0%
	Adware/Spyware	29%	34%	28%	9%	86%	14%	0%	56%	44%	0%
	Worm	39%	47%	11%	3%	84%	16%	0%	24%	76%	0%
	Downloader	43%	29%	24%	4%	69%	31%	0%	32%	68%	0%
Related Work I[37]	Trojan	21%	34%	29%	16%	0%	2%	98%	0%	0%	100%
	Adware/Spyware	16%	32%	33%	19%	0%	1%	99%	0%	0%	100%
	Worm	27%	28%	37%	8%	0%	0%	100%	0%	0%	100%
	Downloader	19%	41%	23%	17%	0%	2%	98%	0%	0%	100%
Related Work II[36]	Trojan	94%	5%	1%	0%	-	-	-	-	-	-
	Adware/Spyware	99%	0%	1%	0%	-	-	-	-	-	-
	Worm	96%	4%	0%	0%	-	-	-	-	-	-
	Downloader	98%	2%	0%	0%	-	-	-	-	-	-

Table 3: Performance comparison with two representative existing approaches

The result is presented in Table 3. We use the following metrics for the comparison:

– Increased APIs. For each of three approaches, we pick the longest trace during any single run to compare with the normal run. For each approach, we record the percentage of malware samples whose increased APIs belonging to $0-10\%$, $10-50\%$, $50-100\%$, or 100% and above. From the result, we can see that Related Work I performs the best among all approaches, which is obvious because this approach blindly explores all possible paths and we select the path with most APIs in the comparison. Meanwhile, in our test, pre-configured environment (Related Work II) can seldom expose malware's hidden behaviors; on average it only increase 5%

more APIs. Thus, even though pre-configured environment has no extra overhead for the analysis, it cannot effectively analyze targeted malware. It further confirms that it is impractical to predict malware's targeted environment beforehand. Our approach clearly performs significantly better than Related Work II, and very close to Related Work I.

- Number of Rolling Backs, which is a key factor to slow down analysis. For exploring both branches, Related Work I has to roll back the execution. In theory, for each environment-sensitive branch, it requires one roll back operation. From the result, we can see that most of the samples have to roll back over 500 times to finish the analysis. However, our GOLDENEYE can efficiently control the number of rolling back because it only occurs when branch prediction cannot determine the right path to select. The largest number of rolling back in our test is 126 and median number is 39. It means that we can save more than 90% overhead when compared with multi-path exploration.
- Memory Usage. According to the description in [37], average snapshot for rolling back consumes around 3.5MB memory. Considering their approach needs to recursively maintain the context for branches and sub-branches, the memory/disk overhead should be over 5MB. However, the highest memory/disk usage of GOLDENEYE is only around 1-2MB, which is much less than half of the memory overhead in Related Work I. Hence, for memory usage, our system also outperforms the compared solution.

Finally, we also compare the total time to complete analysis for GOLDENEYE and Related Work I. For each malware, both GOLDENEYE and Related Work I may generate multiple traces and we sum up all the time as the total time to complete the analysis of the malware. The result is summarized in Figure 5. As we can see, for GOLDENEYE, the average analysis time per malware is around 44 minutes, while the average time for Related Work I is 394 minutes, which is around 9 times slower. Furthermore, the worst case for GOLDENEYE never exceeds 175 minutes while there are 12% of tested malware takes longer than 12 hours for Related Work I (note that if we do not set the 12 hour limit, the average for Related Work I will be much longer). This clearly indicates that GOLDENEYE is much more efficient.

In summary, it is evident that our approach has better performance regarding the trade-off of effectiveness and efficiency. We believe the main reason that other solutions have a higher overhead or lower effectiveness is because they are *not* designed to analyze malware's targeted environment. In other words, our approach is more *proactive* and *dynamic* to achieve the goal of targeted malware analysis.

6.4 Experiment on Known Environment-Targeted Malware Dataset

In this experiment, we aim to verify that our system can extract known targeted environments for malware samples. We began our experiment from collecting the ground truth of some malware set. We look up multiple online resources, such as [43], for the documentation about our collected malware samples. In particular, we first verified that all of them are environment-targeted malware, which means they all need to check some environments and then expose their real malicious intention. Secondly, we manually examine their analysis report and summarize their interested environment elements. We group them into five categories: *System Information*, *Network Status*, *Hardware*, *Customized Objects*, and *Library/Process*. For instance, if one sample's malicious logic depends on some system-wide mutex, we consider it as sensitive to *Customized Objects*. We record our manual findings about our test dataset in Table 4(a).

There are several caveats in the test. First, if the documentation does not clearly mention the sample's MD5 or the sample with the specific MD5 cannot be found on-

	System	*Network*	*Hardware*	*Customized Object*	Library Process
Conficker[43]	√	√		√	√
Zeus[21]	√	√	√	√	√
Sality[12]	√	√			
Bifrost[2]	√	√	√	√	
iBank[7]	√	√	√	√	√
nuclearRAT[9]	√	√	√	√	√
Duqu[4]	√	√	√	√	√
Nitro[16]	√	√	√		
Qakbot[11]	√			√	√

(a) Ground Truth

	System	*Network*	*Hardware*	*Customized Object*	Library Process
Conficker	√	√		○	√
Zeus	√	√	√	√	○
Sality	√	√			
Bifrost	√	√	√	○	
iBank	√	√	×	×	√
nuclearRAT	√	×	√	○	○
Duqu	○	√	√	√	√
Nitro	○	√	○		
Qakbot	○			√	√

(b) GOLDENEYE Environment Extraction Result

√: Correctly Extracted, ○: Similar Element

×: Not Extracted

Table 4: Test on Targeted Malware

line, it may bring some inaccurate measurement for the result. One example is the Trojan iBank [7] case. We analyze some of its variants and they may not exhibit the same behaviors as the documented states. Second, we conclude the extraction result in three types: (a) *Correctly Extracted* means GOLDENEYE can extract the exact same environment element. (b) *Similar Element* means GOLDENEYE finds some element that acts the similar functionality as mentioned in the document, but such element may have different name as the document described. We suspect it is probably because the element name is dynamically generated based on different information. For this type, we consider GOLDENEYE successfully extracts the environment information, because the correct element name could be derived through further manual examination or automatic symbolic execution [22]. (c) *Not Extracted* means GOLDENEYE fails to extract the environment element.

From the result, we can see that our GOLDENEYE can correctly detect most of the targeted environment elements (41 out of 44) within the 5-min analysis time limit. However, our system fails to extract 3 elements out of 44 cases. After we manually unpack the code and check the reason of the failures, we find there are two main reasons: (1) Some hardware query functions are not in our labeled API list (e.g., in the case of iBank). This could be solved if we improve our labeled API list. (2) Some element check only occurs after the malware successfully interacts with a remote (C&C) server (e.g., in the case of nuclearRAT). However, these servers may not be alive during our test thus we fail to observe such checks.

6.5 Case Studies

Next, we study some cases in our analysis. We list several environment targets which may trigger malware's activities.

Targeted Location. For Conficker A, GOLDENEYE successful captures the system call `GetKeyboardLayout` and automatically extracts malware's intention of not infecting the system with Ukrainian keyboard [43]. For some variants of Bifrost[2], GOLDENEYE finds they query the system language to check whether the running OS is Chinese system or not, which is their targeted victim environment. For these cases, GOLDENEYE can intelligently change the query result of APIs, such as `GetKeyboardLayout`, to make malware believe they are running in their targeted machine/location.

User Credentials. We found several malware samples target at user credentials to conduct their malicious activities. For example, we found that Neloweg[19] will access registry at `Microsoft/Internet Account Manager/Accounts key`,

which stores users' outlook credentials. Similar examples also include Koobface[8], which targets at user's facebook credentials. GOLDENEYE successfully captures these malicious intentions by providing fake credentials/file/registry to malware and allowing the malware to continue execution. While the malware's further execution may fail because GOLDENEYE may not provide the exact correct content of the credential, GOLDENEYE can still provide enough targeted environment information to malware analysts.

System Invariants. In our test, GOLDENEYE extracted one mutex from Sality [12] whose name is `uxJLpe1m`. In the report, we found that the existence of such mutex may disable Sality's execution. This turns out to be some common logic for a set of malware to prevent multiple infections. Similar logic has also been found in Zeus [21] and Conficker [43]. For these cases, even though the clean environment, which does not contain the mutex, is the ideal environment for analysis, we can still see that GOLDENEYE's extracted information is useful, potentially for malware prevention, as discussed in [48].

Displayed Windows and Installed Library. iBank [7] Trojan is one example that is sensitive to certain displayed windows and installed library. In particular, GOLDENEYE detects that IBank tries to find the window `"_AVP.Root"`, which belongs to Kasperky software. Meanwhile, it also detects that IBank accesses `avipc.dll` in the home path of Avira Anti-virus software. Our GOLDENEYE further detects if such library or window exists, the malware exhibits more behaviors by calling the function `AvIpcCall` in the library to kill the AV-tools. IBank samples tell us that if our analysis is performed in an environment without AV tools installed, we will miss these anti-AV behaviors. Hence, as a side effect, GOLDENEYE could be a good automatic tools for analysts to detect malware's anti-AV behaviors.

Others. Last but not least, we always assume exposing more malicious behaviors is better. However, detecting some path with less malicious behaviors may be also interesting. One example we find in our dataset is Qakbot [11]. The malware exhibits some behaviors related to some registry entry. This malware tries to write `_qbothome_qbotinj.exe` into a common start up registry key `CurrentVersion\Run`. The further logic for Qakbot needs to check the existence of such registry entry and if it fails, malware goes to sleep routine without directly exhibiting some malicious behaviors. This case is interesting for us because we find that by changing environment setting, we could even observe some *hidden dormant* functionality. Discovering such hidden dormant functionality may help defenders to make some schemes for slowing down the fast-spreading of certain malware.

6.6 Experiment on Distributed Deployment of GOLDENEYE

Finally, we evaluate the performance overhead of our distributed deployment of GOLDENEYE. In this experiment, we measure three cases:

- Case I: Generate a parallel task for all environment-sensitive branches.
- Case II: Generate a parallel task only when the branch evaluation cannot decide a branch after measuring the branch selection heuristics.
- Case III: Do not generate a parallel task and do not conduct rolling back, i.e., using a single machine instead of distributed deployment (for undetermined paths, we select the default environment as desired).

We use additional four worker (virtual) machines for this measurement (Case I and II). Each virtual machine installs original unpatched Windows XP SP1 operating system. We randomly select 100 malware samples and run each sample for at most 300

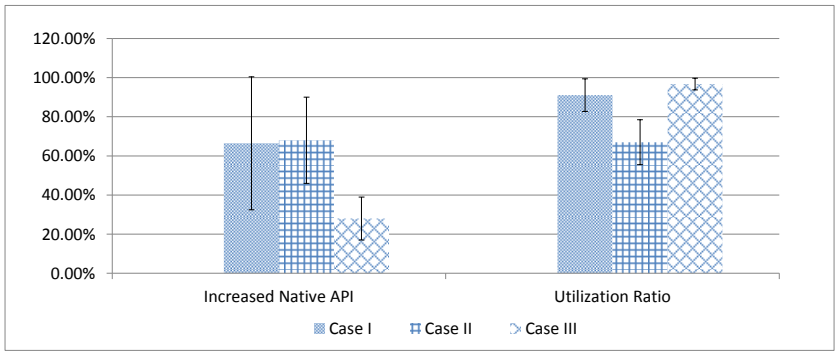

Fig. 6: Measurement of Distributed GOLDENEYE

seconds in each configuration. We compare performance with the baseline case, which is running each malware in the default environment.

The result is summarized in Figure 6. As seen in the figure, we study the effectiveness by measuring the increased ratio of native APIs. As expected, Case I and II expose over 30% more behaviors than Case III. However, the standard deviation of Case I is higher than Case II. It shows that, with the same analysis time, the first approach may not outperform the second case because exploring all environment-sensitive paths is not efficient enough. We also measure the utilization ratio of the analysis machine(s), which is defined as the percentage of time for an analysis machine to run the analysis task within the given 300 seconds. The average utilization ratio from VMs in Case I is over 90%, which is much higher than Case II. In short, we conclude that Case II configuration of GOLDENEYE, i.e., combining the branch selection scheme with the distributed deployment, seems to achieve the best balance between effectiveness and resource consumption among the three cases.

7 Discussion

Exposing malicious behaviors of environment-targeted malware is a challenging research task for the whole malware defense community. As a new step towards systematic environment-targeted malware analysis, our solution is not perfect and not targeting to completely solve the problem. We now discuss limitations/evasions below.

Correctness of Path Selection/Prediction. One limitation of our approach is that the correctness of our branch evaluation depends on whether malware's behavior fits our heuristics. One solution for this problem is to explore all possible branches by multi-round snapshot-and-recover analysis, as in [37]. However, this scheme may cause much higher overhead because of the *path explosion* problem. Hence, to trade off the performance, we choose to apply snapshot-and-recover only when we cannot apply the heuristics. Other dynamic analysis approaches such as previous work [39,41] can also be applied to make the analysis more efficient.

Possible Problems of Taint Analysis. In our scheme, we apply taint analysis at the stages of preprocessing and speculative execution. For preprocessing, taint analysis can help us filter out the malware which are not sensitive to the environment. For speculative execution, taint analysis helps to save execution overhead from multiple aspects. However, as discussed in related work [22], taint analysis could have limitations of *over-tainting* and *under-tainting*. Even though it may cause the problem of imprecise

results, for our cases, the limitation can seldom affect our analysis. This is because: (1) Even though over-tainting costs more overhead for speculative execution, our scheme is still more lightweight than existing approaches. (2) The under-tainting problem may mislead our branch prediction. However, by using stricter branch selection criteria, we could avoid such wrong branch. Meanwhile, conducting more roll-backing operations on some critical branches can also improve the overall accuracy. (3) Our analysis can be independently conducted even without taint analysis. In this case, our speculative execution engine has to be executed at all branches to truncate undesired environments. Even though it may cause more overhead, we believe it still outperforms other approaches because it prevents unnecessary rolling-back.

Evasion through Misleading the Analysis. The implementation GOLDENEYE is built upon on binary instrumentation, and because of the similar limitation as VMM-based approaches[28], it is possible for malware to detect the existence of GOLDENEYE.

By knowing our heuristics for branch selection, the attacker could mislead our analysis through injecting some certain APIs in the branches. However, some heuristics (e.g., environment interaction, process termination) are relatively hard to be evaded because otherwise they will be against the malware's execution intention. We note that even in the worst case (we have to rewind to explore another branch, similar to existing multi-path solutions), our solution is still better than a blind multi-path exploration scheme.

Another way to evade the analysis is to query environment information and process it at a very later time. To handle this issue, we could increase the capacity of parallel spaces and track the tainted environment elements throughout the whole analysis by paying a little more analysis overhead.

Malware can insert some dormant functions such as `sleep` because GOLDENEYE may not prefer to choose branches in which malware could enter a dormant status. To handle such cases, GOLDENEYE can examine more code blocks in the foreseeing operation in order to make a more accurate branch selection or could simply generate a parallel task for another worker machine.

Last but not least, current implementation of GOLDENEYE does not handle implicit control flow, a common issue to many dynamic analysis systems. Hence, malware authors may evade the analysis by including implicit control flow. However, this issue could be partially solved by conducting symbolic execution on indirect branches. We leave it as our future work.

Environment-Uniqueness Malware. A recent study [27] discussed a novel anti-analysis technique, which applies environment primitives as the decryption key for the malware binary. In the real world, flashback [18] malware has exhibited similar interesting attributes. To the best of our knowledge, there is no research or tool can automatically analyze such kind of malware. Even though our approach cannot provide correct analysis environment for the captured sample, we believe our analysis can still discover more information than traditional automatic analysis techniques. For example, our approach can detect malware's query for system environment and deduce what are likely environment elements that compose the decryption key. We leave the analysis of such malware to our future work.

8 Conclusion

In this paper, we have presented a new dynamic analysis system, GOLDENEYE, to facilitate targeted malware analysis by efficiently and effectively exposing its targeted environments. To achieve our goal, we design several new dynamic analysis techniques based on speculative execution, such as parallel environment spaces construction and branch evaluation, to solve the technical challenges faced by targeted malware analysis.

To further improve the accuracy and efficiency, we deploy GOLDENEYE onto a distributed computing model. In the evaluation, we show that our scheme can work on a large real-world malware corpus and achieve a better performance trade-off compared with existing approaches. While not perfect, we believe this is a right step towards an interesting new topic, i.e., targeted threat analysis and defense, which needs further research from the community.

9 Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant CNS-0954096 and the Air Force Office of Scientific Research under Grants FA9550-13-1-0077 and FA-9550-12-1-0077. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and AFOSR.

References

1. Anubis: Analyzing unknown binaries. http://anubis.iseclab.org/.
2. Bifrost. http://www.symantec.com/security_response/writeup.jsp?docid=2004-101214-5358-99.
3. Disassembler library for x86/amd64. http://code.google.com/p/distorm/.
4. Duqu. http://www.kaspersky.com/about/press/major_malware_outbreaks/duqu.
5. DynamoRIO . http://dynamorio.org/.
6. Flame. http://en.wikipedia.org/wiki/Flame_(malware).
7. IBank. http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Troj~IBank-B/detailed-analysis.aspx.
8. Koobface. http://www.symantec.com/security_response/writeup.jsp?docid=2008-080315-0217-99&tabid=2.
9. NuclearRAT. http://en.wikipedia.org/wiki/Nuclear_RAT.
10. Offensive Computing. http://www.offensivecomputing.net/.
11. Qakbot. http://www.symantec.com/connect/blogs/w32qakbot-under-surface.
12. Sality. http://www.symantec.com/security_response/writeup.jsp?docid=2006-011714-3948-99.
13. Stuxnet. http://en.wikipedia.org/wiki/Stuxnet.
14. Symantec intelligence quarterly. http://www.symantec.com/threatreport/quarterly.jsp.
15. Symantec: Triage analysis of targeted attacks. http://www.symantec.com/threatreport/topic.jsp?id=malicious_code_trend.
16. The Nitro Attacks: Stealing Secrets from the Chemical Industry. http://www.symantec.com/security_response/whitepapers.jsp.
17. Trends in targeted attacks. http://www.trendmicro.com/cloud-content/us.
18. Trojan BackDoor.Flashback. http://en.wikipedia.org/wiki/Trojan_BackDoor.Flashback.
19. Trojan.Neloweg. http://www.symantec.com/security_response/writeup.jsp?docid=2012-020609-4221-99.
20. Virustotal. https://www.virustotal.com/.
21. Zeus Trojan horse. http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99.
22. 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'10*, 2010.

23. D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient detection of split personalities in malware. In *Proc of NDSS'10*, 2010.
24. L. Bilge and T. Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proc. of CCS'12*, 2012.
25. D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In W. Lee, C. Wang, and D. Dagon, editors, *Botnet Analysis and Defense*, volume 36, pages 65–88. Springer, 2008.
26. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proc. of CAV'11*, 2011.
27. P. Royal C. Song and W. Lee. Impeding automated malware analysis with environment-sensitive malware. In *Proc. of HotSec'12*, 2012.
28. X. Chen, J. Andersen, M. Mao, M. Bailey, and J. Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Proc. of DSN'08*, 2008.
29. P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Krugel, and S. Zanero. Identifying dormant functionality in malware programs. In *Proc. of S&P'10*, 2010.
30. A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proc of CCS'08*, 2008.
31. J. Gonzlez and A. Gonzlez. Speculative execution via address prediction and data prefetching. In *Proc. of ICS'97*, 1997.
32. M. Graziano, C. Leita, and D. Balzarotti. Towards network containment in malware analysis systems. In *Proc. of ACSAC'12*, December 2012.
33. 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.
34. C. Kolbitsch, E. Kirda, and C. Kruegel. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proc. of CCS'11*, 2011.
35. C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of S&P'12*, 2012.
36. Martina L, Clemens K., and M.Paolo. Detecting Environment-Sensitive Malware. In *Proc. of RAID'11*, 2011.
37. A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proc. of S&P'07*, 2007.
38. A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. of ACSAC'07*, 2007.
39. Y. Nadji, M. Antonakakis, R. Perdisci, and W. Lee. Understanding the Prevalence and Use of Alternative Plans in Malware with Network Games. In *Proc. of ACSAC'11*, 2011.
40. A. Nappa, Z. Xu, M. Z. Rafique, J. Caballero, and G. Gu. Cyberprobe: Towards internet-scale active detection of malicious servers. In *Proc. of NDSS'14*, 2014.
41. M. Neugschwandtner, P. M. Comparetti, and C. Platzer. Detecting Malware's Failover C&C Strategies with SQUEEZE. In *Proc. of ACSAC'11*, 2011.
42. Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-force: Force-executing binary programs for security applications. In *Proceedings of the 2014 USENIX Security Symposium*, San Diego, CA, August 2014.
43. P. Porras, H. Saidi, and V. Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. `http://mtc.sri.com/Conficker/`, 2009.
44. S. Shin, Z. Xu, and G. Gu. Effort: Efficient and effective bot malware detection. In *Proc. of INFOCOM'12 Mini-Conference*, 2012.
45. M. Sikorski. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.
46. J. Wilhelm and T. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proc. of RAID'07*, 2007.
47. Z. Xu, L. Chen, G. Gu, and C. Kruegel. PeerPress: Utilizing enemies' p2p strength against them. In *Proc.of CCS'12*, 2012.
48. Z. Xu, J. Zhang, G. Gu, and Z. Lin. AUTOVAC: Towards automatically extracting system resource constraints and generating vaccines for malware immunization. In *Proc. of ICDCS'13*, 2013.