2013 7th International Conference on Software Security and Reliability

Divergence Detector: A Fine-grained Approach to Detecting VM-Awareness Malware

Chia-Wei Hsu [†] Fan-SyunShih [‡] Chi-WeiWang [†] Shiuhpyng Winston Shieh [†]

Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan (R.O.C.)
[†] {hsucw, cwwang, ssp}@cs.nctu.edu.tw, [‡] jpeanut750908@gmail.com

Abstract*—Virtualized execution has become an effective mechanism to analyze malware in a dynamic way. To conceal its malicious behaviors, VM-aware malware probes the execution environment for analysis-resistance. These malware programs hide their malicious behaviors if they are launched in a virtual machine (VM). VM awareness becomes a barrier for malware analysis due to the concealment of malicious behaviors. In this paper, we discover that uncertain factors have significant influence on the effectiveness of malware detection. To cope with the problems, a new VM-aware detection scheme, namely *Divergence Detector*, is proposed to address the swindle of the evolved malware. Unlike conventional schemes, the Divergence Detector reduces the *uncertain factors* at instruction level, and can detect the divergence of multi-execution traces across heterogeneous virtual machines. The proposed Divergence Detector is implemented across the three commonly used VM platforms, that is, QEMU, Bochs and Xen. It compares the code coverage of the execution traces on various VM platforms to discover the deviation of behavior, thereby precisely detecting the VM-awareness. We will formally predict the effectiveness of Divergence Detector by constructing a mathematic model, which shows the maximum false positive rate is exponentially decreased with respect to the number of multi-executions. Representative samples utilizing seven types of commonly used VM-aware techniques were also employed for evaluation. The evaluation results indicate that the maximum false positive rate complies with our prediction. The uncertain factors play the major role in the VM-awareness detection. To reduce uncertain factors causing false positives, a method is proposed for VM-aware detection. The Divergence Detector can also enable the identification of new types of malware since the benign programs do not need to be aware of execution environment.

Keywords—Virtual Mashine; VM-awareness; Malware

I. INTRODUCTION

In recent years, virtual machines (VMs) [24] have been widely adopted for malware analysis. Due to the nature of VM, malware behaviors can be monitored, profiled and analyzed in the well protected and controllable environment. Much research work has been studied by using VM-based analysis, such as tracking the data dependency [1][2], recording system calls patterns of malware [3] recovering code obfuscation [4], and unpacking executable [5]. Moreover, many VM-based analysis tools [6]-[12] were developed to cope with these cyber-crimes.

To disguise its malicious behavior, a malware program may be equipped with a new capability to detect the presence of the virtual machines [13][14]. If they are launched in VM environments, the malicious behaviors will be concealed by changing its control flow, thereby pretending to be a benign executable. According to the study [15], more than 40% of malware programs contain anti-virtualization or anti-debugging functionalities, and have the ability to distinguish the VM environments from physical machines. For readability, we refer to these malware programs as *VM-aware* malware, and the methodologies they used as *VM checks*. In this paper, the programs which equip VM checks are temporarily regarded as malware since the benign program do not need to be aware of execution environment.

VM checks detect the VM environments by discovering the differences between the virtual machines and the physical machines, such as the time cost of execution, the specific identities of emulated devices, and the instruction-emulation bugs [13][16]. The VM checks can be classified into the following three categories:

System Fingerprint Checks: The system fingerprint check can recognize the specific identities in VM. For instance, VMware uses the name “Logic BT-958” and “pcnet32” for the virtual VGA adapter and network adapter [15][17][18], respectively. A program can be VM-aware by checking these hardware identities. If there is a match, the system fingerprint is recognized.

Unfaithful Execution Checks: This kind of VM checks can discover the differences of machine states after executing a series of instructions on both the VMs and the physical machines. Most of unfaithful executions [12][13][16] are caused by the instruction translation bugs (e.g. instruction CMPXCHG8B and undocumented instruction ICEBP on QEMU). Malware can differentiate the VM environments from physical machines if the execution results are not as expected. The unfaithful execution checks are hard to be addressed. The emulation or virtualization bugs are always discovered at time goes on. Even the VM authors cannot know what bugs are exploited by coming malware.

Execution Timing Checks: Execution timing checks measure the time cost of instructions and compare with the anticipated answers. In general, the computation time in VMs is higher than that in physical machines. In general, an instruction running on a guest system is usually translated into several instructions on the host. Moreover, the VM, which executes the guest system binaries directly on the host, can be also detected by using the timing attack [20] due to the hardware limitation

978-0-7695-5021-3/13 $26.00 © 2013 IEEE
DOI 10.1109/SERE.2013.23

IEEE computer society

(e.g. TLB access time profiling on Intel VT [19] based virtualization). Worse, the malware can query the remote server for the time of a serious execution. This kind of VM checks is hard to address today.

Conventional VM-aware detection schemes mainly investigated the search for the deviation of system call sequences. Only the system-call information will suffer from the mimicry attack to cause a false negative. Additionally, the uncertain factor, which may cause execution divergence, is not addressed so far. To cope with the problems, a new VM-aware detection scheme is proposed to address the unfaithful execution checks and the partially execution timing checks. The proposed VM-aware malware detection scheme, namely Divergence Detector, aims to discover *VM-aware divergent points* (VMDP), thereby detecting VM-aware malware. Contributions of the proposed scheme are three folds:

VM-aware Divergent Points Discovery: The VM-aware techniques are used to disguise malicious instruction-level behaviors. The code coverage comparison of multi-executions can explore the divergent points of the traces across VMs. However, the *uncertainty attributed divergent points* (UADPs) are caused by uncertain factors instead of probing the execution environments. In this paper, the proposed scheme reduces the UADPs so that the VM-awareness will be accurately identified. These VMDPs can enable further analysis of VM-aware malware. For example, the malign VMDPs can be extracted for signature-based detection.

Uncertainty-Attributed Divergences Reduction: In general, a comparison-based VM-aware detection, either system calls or instructions sequences, suffer false positives easily while the referenced environment contains uncertain factors. The uncertainty of execution, usually uncontrollable factors, such as randomization and interrupts changes control flows even in the same machine. Comparison-based VM-aware detections will have a false positive when the uncertainty is regarded as VM-awareness. Some research eliminates the uncertainty by execution replay systems. The replay system must be based on prior knowledge of VM and malware, and may have false negative if the system replays the data used for VM-awareness. The proposed Divergence Detector distinguishes the uncertainty-attributed divergences from the VM-aware branches by a general technique. Multiple execution traces are applied to reduce the uncertainty attributed divergent points effectively.

Comprehensive Detection: Current VM-aware malware is aware of the commonly used VMs including VMware, QEMU [22], Bochs [23], and Xen [30]. Except VMware, all of them are open source VM that can be modified for malware analysis. It is reasonable to choose these VM platforms for VM-aware detection. VM checks are applicable on these platforms for analysis-resistance. By comparing the executed code coverage on each VM, VMDPs can be found due to the differences of VM implementation. Emulation VMs such as QEMU and Bochs suffer from unfaithful execution checks that virtualization VM does not. However, virtualization VM such as Xen can be detected by execution timing checks. We leverage the diversity of VMs to develop the methods for VM awareness.

The rest of the paper is organized as follows. Section II introduces related work, and section III is for the problem definition and assumptions. In section IV, a new VM-aware detection scheme, namely Divergence Detector, is proposed. The design and implementation will be elaborated in section V, and the evaluation will be given in section VI. Section VII will conclude the paper.

II. RELATED WORK

Due to the existence of VM-aware malware, some studies have attempted to explore the methodology for analysis-resistance. Coarse grain comparison of system calls being invoked at the kernel level was conducted to show the deviating behaviors of suspicious samples in VM as opposed to that in physical machines [15]. They used BackTracker [25] to record the system calls (e.g. disk read/write, process fork, etc.). This scheme can successfully and quickly discover the differences of system call sequences. However, they cannot know the cause of the execution differences. The similar work [31] proposed a reliable replay system to reveal the VM-awareness. By replaying the system calls recorded on a reference system, the stealthy behaviors of malware can be captured, or their VM-awareness can be discovered. Although both research works investigate the system call deviations of the processes to recognize their VM-awareness, system call sequences can be easily fooled by sophisticated attackers. It is possible to invoke the same system call sequences after the VM checks, but malicious instructions only generate harmful arguments to the branch passed the VM checks. Moreover, the replay system based on prior knowledge may make mistake when it replays the data used for VM-awareness.

Due to the VM-check techniques, VM-aware malware can disguise itself as a benign program. Consequently, the advanced malware analysis tools [6][12] will produce false negatives. Discrepancy of the instruction traces was analyzed in an attempt to ensure emulation resistant at instruction-level [21], where suspicious samples were executed in both TEMU [27] and Ether [12] at the same time. Ether, a Xen-based analysis tool, is the hardware virtualization rather than emulation. Therefore the emulation-resistant malware will reveal their malicious behavior in the Ether. However, there is a virtualization-resistant technique which uses TLB access time profiling [20] or system fingerprinting checks for detecting virtualization VM, such as Xen. A correct VM-based analysis cannot be made until these VM checks are addressed by a comprehensive analysis. Our approach leverages on different VM implementations to discover the VM-awareness.

All of studies mentioned above detect the VM-awareness by dynamically executing the malware. The scope of collected code coverage is the most important issue on dynamic analysis. Our approach launches the target executables multiple times to maximize the code coverage. However, the execution deviation may happen even executing the applications which are not VM-aware. These deviations come from inconsistent execution environment, which includes the time-related event and memory states, etc. Some execution replay systems [32][33] are proposed to address the non-determinism in VM. Previous works record all of non-deterministic events and replay them in later execution. However, the instruction-level replay system is not easy to be imple-

mented and evaluated in practice. Small timing differences possibly cause temporary deviations while replaying even on the same VM environment. Hence, uncertainty is hard to be eliminated across heterogeneous platforms unless we can exploit the nature of different implementation of VMs.

III. Problem Definitions and Assumptions

For a fine-grained VM-aware analysis, our approach discovers divergences of executions across heterogeneous platforms by instruction-level comparison. However, the execution of many benign applications may be affected by "uncertain factors". The uncertain factors, which cause the divergence of traces, are non-deterministic and may change the control flow. These factors can be time-related events or external inputs. The experiment results may be inaccurate unless the uncertain factors are eliminated. For example, CPU clocks may not be synchronized across heterogeneous platforms. With unsynchronized clocks, the control flow of a program may diverge. This divergence is not caused by VM checks. Instead, uncertain factors play a pivotal role for reducing the false positives of VM-aware detection. Uncertain factors may include system time, random numbers, network packets, etc. They may lead to the alteration of control flow though they are irrelevant to VM checks. As a result, the false positives will be raised due to the divergence caused by the uncertain factors.

A VM-aware program contains three key components, namely the malicious binary code, the VM checks, and the divergent points. All VM-aware programs share a common feature, that is, they need a conditional branch to conceal their malicious behaviors. A VM check usually contains at least one branch statement to execute either the malicious code or the camouflage. Finding the closest conditional branch right before malicious code is the key for VM-aware detection. At the binary instruction level, a VM check may consist of a series of branches. We regard the last branch of a VM check as *VM-aware checkpoint* (VC) and the rest of branches of the VM check as *status checkpoint* (SC). In other words, VC is the branch to the VMDPs, and SC is the branch to the UADPs.

The execution code coverage will be distinct while the malware program is executed in a VM and in a physical machine, respectively. The discrepancy of execution can be presented by the divergence of the instruction traces. Each VC is followed by two distinct execution paths. The VM-aware divergent points (VMDP) we name are the binary instructions right after the VC when two traces run distinct paths. Both the benign and malign divergent points succeed a VM-aware checkpoint VC which decides the branching. VMDPs are reliable indexes, and can be used to identify a VM-aware program. Once the VMDP is found, the hidden malicious code can be discovered. Both the information of VMDPs and VCs are very useful for malware analysis.

Next, we will illustrate the difference between uncertain factors and VM checks. Both of them may affect control flows. In Figure 1, line 3 is a branch affected by the random input k, and line 4 is a VC, dependent on the result of detect_QEMU(). **k** is an uncertain factor and does not depend on any VM-check. Without an effective way for differentiation, the program may be misinterpreted and two VC candidates (branches on lines 3 and 4) will be identified in the sample. In fact, the conditional branch on line 4 contains a VC using the VM-aware technique to detect the QEMU environments. The other is a parity check. The program goes to the true statement when parity **k** is odd. The chance of executing the code block on line 3 is 50 percent possibility regardless of the execution platforms.

```
1.  int k;
2.  k= ( rand() % 2 );   //0-1
3.  if (k) { /* do something */ };
4.  if (detect_QEMU())
5.  printf("QEMU\n");
6.  else
7.  printf("Not QEMU\n");
8.  return 0;
9.  }
```

Figure 1.Sample program containing an uncertainty branch.

Without eliminating the uncertain factors, the comparison-based VM-aware analysis will fall into misjudgment. Malware programs can make use of a large number of uncertain factors to hide its usage of VM-aware techniques. It will incur high cost to distinguish divergent points of VM-aware from uncertain factors. Here, we name the branch attributed to uncertain factors as *uncertainty-attributed divergence points* (UADPs). In contrast to VMDP, UADP succeeds an uncertainty branch, rather than a VC. The UADPs can be reduced by testing their repeatability.

The analysis of a program is very complicated, and can be treated as the halting problem in computational theory. To simplify the VM-aware detection problem, the studies of this area usually implicitly made the following two assumptions:

A.1: Target VM-aware malware program will execute VMchecks while being analyzed. We assume that the VM-aware malware defined in this paper always check the execution environment for analysis-resistance. This assumption implies that a target VM-aware malware program will execute VM checks while being analyzed. In other words, a malware program will eventually execute VM checks. However, there is no guarantee when the checks will be conducted. Just like the halting problem in computational theory, it is hard to determine the exact time the executable will execute the binary of VM checks. That means complete code coverage cannot be collected unless the execution is terminated immediately. Without the complete code coverage, the divergence of execution traces may not appear because the instructions of VM checks are not executed. To narrow down the scope of the problem, we make this assumption for the VM-aware malware.

A.2: A single VM check cannot detect all types of referenced VMs within an atomic instruction. This assumption implies that there is always a divergence of execution after VM checking across heterogeneous VMs. Most of known VM checks are detecting specific artifacts in VMs. Except the elaborate execution timing checks, it is hard to perform a single instruction to detect all types of VMs due to their discrepant implementations. A combinative VM check may detect all types of reference platform at the same time. However, it consists of a series of VM-aware checkpoints (VCs), which separately probe environment and diverge control flows. Due to the nature of atomicity at the instruction level, a branch can only perform a single check, but cannot combine multiple checks into a branch. The two diver-

gent points to be executed in the if-else statements will be determined by each conjoined VCs. If every VC satisfies the assumption A.2, the combinative VM check must appear divergences of the execution. Furthermore, a VM check is very likely to return a unique outcome for each type of VMs. Most types of VMs, due to their diverse implementations, do not share the same features, thereby not returning the same value in response to a VM check. For example, we discover that two famous types of VM, namely Xen and QEMU, may return the same identity value to a VM check although one is the emulation VM and the other is virtualization VM. This is because Xen used part of the QEMU code to emulate the hard disk devices, the same implementation. This shared code will lead to false negative when testing some malware samples by related works using the two VMs to identify hard disk devices for VM awareness. Thus, another kind of VMs, Bochs, has been added to extend the code coverage comparison. By thoroughly examining Bochs' code, it has different features from QEMU and Xen. More VMs can be selected such as VMware, Parallel Desktop, and Virtual PC as reference environment. Our approach can be extended to a more comprehensive system by adding the VM platforms mentioned above.

IV. Proposed Scheme

In this section, the proposed Divergence Detector will be elaborated. The Divergence Detector aims to detect the VM-aware malware in a way that new challenging issues such as the mimicry attack toward system-level comparison, and the uncertainty-caused false positives can be mitigated. VM-aware detection cannot only rely on the trace in one specific type of VMs. Instead, it should leverage different varieties of VMs for discovering VCs. Hence, a novel approach is proposed to detect VM-aware malware with a predicable false positive rate. By comparing the code coverage in different types of VMs, our system can distinguish the VMDPs from UADPs.

Following is the formal description for VM-aware detection. A binary executable $\boldsymbol{S}$ can be executed on several different VM platforms x, y and z, or more. The executable $\boldsymbol{S}$ contains a set of basic blocks $\boldsymbol{B}$. We call $\boldsymbol{S}$ is VM-aware when it contains at least one conditional branch for VM-awareness. According to assumption ***A.1***, the branch must be executed and always jumps to the same execution path when the platform is the same. Based on assumption***A.2***, some of VM platform will pass the check but the others will not. The divergence can be regarded as a set of executed basic block ***VMDP***s$=\{b_i \mid b_i \in \boldsymbol{B}$, where b_i is executed on some platforms every time but never be executed on the others$\}$.

We will formalize the target executable running in the VMs, discovering the UADP, eliminating uncertainty, determining the VC, and predicting the false positive rate. The target executable launched in VM x for n times generates the set of instruction traces $T_x = \{t_{x,i} \mid \text{for all } i, 1 \leq i \leq n\}$. Each instruction trace $t_{x,i}$ can be translated into code coverage and denoted as $C_{x,i}$, which represents the collection of executed instructions on VM x in the i-th execution round for all i, $1 \leq i \leq n$. The first step of our algorithm, namely code coverage collection, is to unite the code coverage of each execution in VM x to attain an union, $U_x = C_{x,1} \cup C_{x,2} \cup \cdots \cup C_{x,n}$, which will be used in the later steps.

For intra-VM uncertainty reduction, the code coverage union is divided into two subsets of code coverage: the certain code coverage CC_x and the uncertain code coverage UC_x. The certain code coverage, $CC_x = C_{x,1} \cap C_{x,2} \cap \cdots \cap C_{x,n}$, stands for the collection of instructions which were always executed in each round of execution. The remainder of code coverage is uncertain code coverage, $UC_x = U_x - CC_x$, which was executed sometimes. The uncertain code coverage UC_x in VM x can be used for *inter-VM uncertainty reduction* in other VMs. Suppose that there are three kinds of VM x, y and z (In our system, x, y and z represent QEMU, Bochs and Xen, respectively). After the uncertainty reduction, the certain code coverage in VM x is denoted as CC_x', and $CC_x' = CC_x - (UC_y \cup UC_z)$.

In the third step, the comparisons of certain code coverage are done for extracting the differences of the code coverage across VMs, and they are marked as D_x, D_y and D_z, respectively. The code coverage defined as $D_x = (CC_x' - CC_y') \cup (CC_x' - CC_z')$ and the rest may be deduced by analogy. The details will be described later.

Finally, these differences of the code coverage D_x, D_y and D_z are used to discover the VM-aware checkpoints. This code coverage is aligned back to the original instruction traces T_x, T_y and T_z. The codes in CC_x', CC_y' and CC_z' can be mapped into their traces and located in several positions. Then the backward traversal is applied to discover the closest conditional branch. All of the branches can be marked as B_x, B_y and B_z, separately. The branches used in VM checkpoints are denoted as VC where $VC = B_x \cap B_y \cap B_z$.

A. Collection of Code Coverage

Two possible approaches, namely path coverage comparison and statement coverage comparison can be chosen for code coverage comparison. The path coverage comparison [28] is commonly used technique for code comparison. The path coverage comparison can genuinely discover the divergences of a program, but it is time-consuming and it may contain many irrelevant divergence points. For instance, the different length of inputs will cause the divergences in loops. The path coverage comparison takes the difference of looping times into account as a divergence point. However, our main concern is whether the code was executed with certainty or not. Therefore, the methodology of comparison adopted in our scheme is the statement coverage comparison instead of the path coverage comparison.

For comparison of statement coverage, the executed code was recorded during the execution time. We logged the trace of the samples in QEMU, Bochs and Xen, respectively. The design and implementation will be described later in this paper. By multi-executions, the sample can generate the instruction trace $T_x = \{t_{x,i} \mid \text{for all } 1 \leq i \leq n\}$ where x is the VM identity. Each instruction trace $t_{x,i}$ is translated into code coverage $C_{x,i}$ by sorting with the EIP (Extended Instruction Pointer). EIP is the program counter which refers to an instruction in the memory space. Therefore, we can collect all executed instructions and the code coverage U_x of running time where $U_x = C_{x,1} \cup C_{x,2} \cup \cdots \cup C_{x,n}$.

B. Uncertainty Reduction

Eliminating the uncertain code coverage can reduce the false positives. Uncertainty can cause the divergent points which are

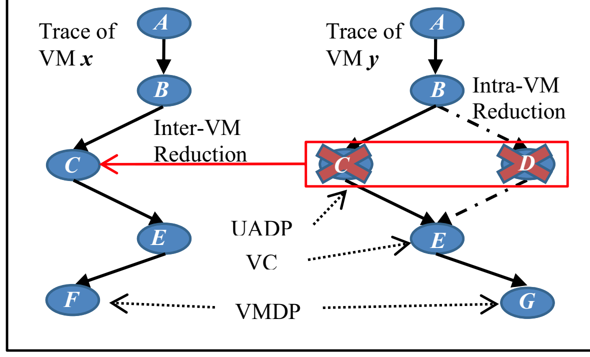

Figure 2. Illustrate theVM-aware detection.

irrelevant to VM awareness. The uncertainty reduction consists of two steps: intra-VM uncertainty reduction and inter-VM uncertainty reduction.

1) Intra-VM Uncertainty Reduction

The branching of VCs will be consistent in the same VM no matter how many times the malware was executed. Otherwise, the VM-checks contravene their purposes. Thus, the code succeeding VCs must appear in every execution round. In other words, the code should be in the intersection of the code coverage CC_x where $CC_x = C_{x,1} \cap C_{x,2} \cap \cdots \cap C_{x,n}$. The certain code coverage CC_x is extracted from repeated execution. The remainder is the uncertain code coverage UC_x where $UC_x = U_x - CC_x$. As an example in Figure 2, there are two sets of code coverage for a sample. Both instructions C and D were executed without certainty, and they succeed the common conditional branch B. In this case, $CC_y = \{A, B, E\}$ and $UC_y = \{D, E\}$. At the beginning, $C_{y,1}$ might be {A, B, C, E, F}. After multi-executions, there will be a $CC_{y,i} = \{A, B, D, E, F\}$ where $1 \le i \le n$. $CC_{y,i}$ can be used for intra-VM reduction since the branching of B is uncertain. The step can be repeated many times for better accuracy.

2) Inter-VM Uncertainty Reduction

After a limited number of execution, some instructions may belong to the certain code coverage of VM x, but at the same time belong to the uncertain code coverage of VM y. In Figure 2, the statement C is in certain code coverage of VM x, but it belongs to uncertain code coverage of VM y. This can formally expressed as $C \in CC_x$ and $C \in UC_y$. The scenario would occur if the uncertainty was not triggered in a limited number of execution rounds in VM x. The potential UADP C in VM x should be removed.

At this point, we use the observed UADPs from VM y for reduction. The new certain code coverage CC_x' was performed by subtracting the certain code coverage of other VMs. In our system, traces of three types of VM were collected, including TQEMU, TBochs andTXen. The inter-VM reduction can be written by $CC_x' = CC_x - (UC_y \cup UC_z)$. After the reduction, we derive the certain code coverage, CC_x', CC_y' and CC_z', which were always launched in VMs for every execution round.

C. Comparison of Certain Code Coverage

The certain code coverage extracted from VMs contains the VMDPs which are caused by VM checks. In Figure 2, the certain code coverage are $CC_x' = \{A, B, E, F\}$ and $CC_y' = \{A, B, E, G\}$. Next, the different certain code coverage were computed by the subtraction of code coverage. The different code coverage $D_x = CC_x' - CC_y'$, that is, the sets $\{F\}$ and vice versa. $D_x = \{F\}$ and $D_y = \{G\}$ are the instructions which took place depending on the type of VM platform.

D. Discovery of VM-aware Checkpoints

Even if we found the hidden instructions which were protected from VM-based analysis tools, the methodologies of VM check are still not clear. Thus, the reason of the VMDPs should be understood by investigating the causal branching VCs. Figure 2 shows that the VC is located at statement E. In this step, the D_x and D_y are used to search for the conditional branch E. First of all, we separately aligned the instructions in both D_x and D_y back to the traces T_x and T_y. Then our system traversed backward from these instructions in T_x and T_y, and stopped at the closest conditional branch. All the branches are denoted as B_x and B_y, which are the candidates of VCs. The B_x may not be identical to the B_y (in Figure 2, B_x is equal to B_y that is $\{E\}$) due to their different behaviors; one is part of hidden activities, and the other is camouflage. Finally, the branches used for VM check decision are the intersection of Bx and By, and they are represented as VC where VC = $B_x \cap B_y$. The point E was recognized as a VC used in VM checks. As our observation, the branches applied to VM checks can be conditional branches, indirect jump and exceptions. The first two can be used for string comparison or switch statement. The exceptions are caused by some emulation bug for VM awareness. For example, QEMU multi-REP prefix check always triggers exceptions to behave differently.

E. Prediction of False Positive Rate

The Divergence Detector is able to discover the uncertainty by multi-executions, thereby lowering false positive rate. The remaining false positive rate is due to the misjudgment of branches that are unconcerned with VM checks. The accuracy of Divergence Detector will be affected by the number of execution rounds n and the number of types of reference VM k. In accordance with the prediction of false positives rate, the following analysis will prove that it is exponentially decreased with respect to the number n and k.

We assume that each divergence point has a probability p branching to the true statement, and a probability of $1 - p$ to the false statement, where $0 \le p \le 1$. And the false positives will occur when the following situation takes place:

The false positive of the comparison of certain code coverage in VMs can be divided into two groups. One group contains the traces of VMs which are always branched into the true statement from the branch caused false positive. Another group always jumped into the false statement even if the branch is unrelated to VM checks.

As mentioned above, we can formulate the probability of false positives. With k types of VMs, n execution rounds and the probability p, the formula of false positives rate is elaborated as follows:

We divided k types of VMs into two groups; one executes true statement with probability p, and the other goes into false statement with $(1 - p)$. Each chosen VM goes to the specific

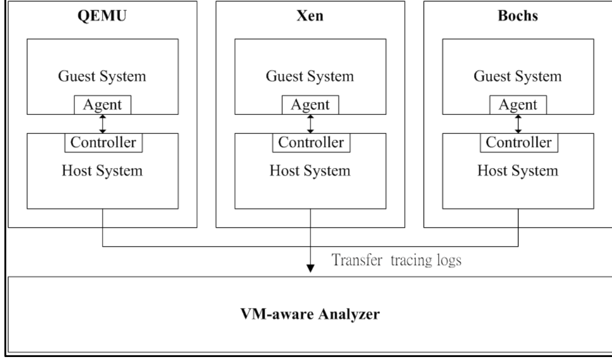

Figure 3.The architecture of the VM-aware detection system.

statement n times, so the probability of chosen VM traces and remainder are $p^n$and $(1-p)^n$, respectively.

The k-combination can be used for this case. The probability of false positive rate ***P*** can be expressed as follows:

$$\boldsymbol{P} = \binom{k}{1} p^n (1-p)^{n(k-1)} + \binom{k}{2} p^{2n} (1-p)^{n(k-2)} + \cdots + \binom{k}{k-1} p^{n(k-1)} (1-p)^n = \sum_{i=1}^{k-1} \binom{k}{i} p^{in} (1-p)^{n(k-i)} \quad (1)$$

This formula indicates how the group can be created. When $i = 1$, the k VMs is divided into 1 and $k - 1$. The false probability is$\binom{k}{1} p^n (1-p)^{n(k-1)}$. When i =2, the sizes of two groups are 2 and $k - 2$, and the probability is$\binom{k}{2} p^{2n} (1-p)^{n(k-2)}$. The i cannot be equal to 0 and k because there is only one group. Thus $\boldsymbol{P} = \sum_{i=1}^{k-1} \binom{k}{i} p^{in} (1-p)^{n(k-i)}$ is induced.

The P has a maximum while the p is equal to $(1 - p)$, that is, $p = 1/2$. Then, the max false positive rate $\boldsymbol{Pmax}$can be calculated that

$$\boldsymbol{Pmax} = \binom{k}{1} p^{nk} + \binom{k}{2} p^{nk} + \cdots + \binom{k}{k-1} p^{nk} = \sum_{i=1}^{k-1} \binom{k}{i} p^{nk} \quad (2)$$

$$\binom{k}{1} + \binom{k}{2} + \cdots + \binom{k}{k-1} = 2^k - \binom{k}{0} - \binom{k}{k} = 2^k - 2 \quad (3)$$

$$\boldsymbol{Pmax} = \sum_{i=1}^{k-1} \binom{k}{i} p^{nk} = (2^k - 2) p^{nk} \approx p^{(n-1)k} \quad (4)$$

Therefore, the maximum false positive rate is exponentially decreased with respect to the number n and the number of VMs k. This result represents that our scheme can effectively lower the false positive rate

V. Design and Implementation

The implementation of our systems is elaborated in this section. Figure 3 shows the architecture of the VM-aware detection system. The Guest System Agents are installed on the guest systems in QEMU, Bochs and Xen, respectively. The guest system agent is a startup program and it is responsible for following tasks.

(1) Setup the samples: To receive the samples outside the VM and place the samples in the specified directory path.

Table 1. The known VM-aware samples

Categories	VM-aware Samples	Addresses of branches in a VM check	Target
Hardware fingerprint check	QEMU hard disk check	0x42e8e4 0x42dbd4	QEMU, Xen
	Bochs hard disk check	0x42dbd6 0x454a94	Bochs
System fingerprint check	Bochs BIOS check	0x45e014 0x42ed15	Bochs
	Xen CPUID check	0x40132f 0x4013e0	Xen
Unfaithful Execution check	multi-rep prefix check	0x4012ff	QEMU
	QEMU, Bochs opcode bswap check	0x4014cf	QEMU, Bochs
Execution Timing check	RDTSC timing check	0x401327	QEMU, Bochs

(2) Launch the samples: To execute the samples on the guest OS when the VM is ready to record the instruction traces of the samples.

The VM controllers master the VM's execution for analysis; it is responsible for the following tasks.

(1) Restore the VM status: Preventing the influence from the samples such as modification of the file system, the VM status needs to be rolled back to the initial status to remove the influences of the system. In the final execution round, the controller will shut down the VM and reboot it from the clean system status.

(2) Export the trace logs: The controller export the trace logs of the samples to our system for VM-aware detection.

With the collaboration between guest system agent and VM controller, the traces of the samples in the VMs can be collected automatically. Next, we will introduce how to recognize the trace of the launched sample. When running the sample, a new process identity will be created by the guest system. The process identity should be memorized for saving its trace. The trace recorder monitors the target process and extracts the trace.

A. Process Identification

To record the trace of a specific process, the process identification is necessary. Each running process in the operating system has a unique Page Directory Base (PDB), and the physical address of the PDB is stored in the CR3 register. Thus we can make use of the CR3 register to dump the traces belonging to the monitored process. As the suspicious sample is executed in the guest system, the guest OS (Windows XP SP2) invokes system calls (i.e. *NtcreateProcessEx*) to create a new process. Next, these system calls used the non-exported kernel function *mmcreatePeb* to allocate a process environment block for the process. Our system intercepted the kernel function call for the PDB address which will be allocated by the operating system. Then the PDB address of the sample was sent to our trace recorder.

B. Trace Recorder

The trace recorder examines the value of the CR3 register whether is identical to that of the target sample. If there is a match, the trace recorder starts to collect the trace of this sample.

```
1.  int main(intargc, char *argv[]){
2.  char hd_str[100];
3.  int result;
4.  hd_str = get_hdver();
5.  result = strcmp (hd_str, "Generic 1234");
      //strcmpentry address 0x454a80
6.  If (result==0)
        //0x42dbcfcmpresult, 0
        //0x42dbd6jnz 0x42dbe7
7.  printf("Bochs HD emulation\n");
8.  else
9.  printf("Not Bochs\n");
```

Figure 4.Source Code of Bochs Hard Disk Check

In our system, the trace recorder was implemented in QEMU and Bochs, which are emulation VMs. It was invoked during the emulated CPU fetching instructions. In Xen, we use the Ether [12], a Xen-based analysis tool, to achieve the recording. We take program counter (EIP) and opcode into account for code coverage comparison. The record of opcode is used for the self-modifying malware which obfuscated the instructions resident in memory. Without the record of opcode, self-modifying malware can achieve different behaviors by the same code coverage. Thus, the comparison of certain code coverage will be incomplete.

C. Time Cheating Module

The execution timing checks utilize the local time source, namely real-time clock, to examine the execution environment. There will be no execution divergence while all of the reference VMs are failed to pass the timing checks. To address the problem, we implement a time cheating module in the Bochs, which is an emulator. In general, the emulators must fetch instructions and translate them into target machine code, hence the sample needs the additional time to run. The time cheating module will store the clock value of CPU before each instruction fetching and the trace dumping. Then it recovers the clock value recorded previously when the instruction is being executed. Hence, our system can partially pass the timing checks and reveals the malicious behavior.

VI. Evaluation

In this section, a series of experiments were performed to evaluate the correctness of Divergence Detector. We examined the known VM checks including system fingerprinting checks (Bochs hard disk check) and unfaithful execution checks (QEMU multi-rep prefix check). Then the false positive rate of uncertainty reduction is evaluated.

The three kinds of most popular VMs used for analysis including QEMU-0.9.1, Bochs-2.4.2 and Xen-3.1.0, were chosen to be part of Divergence Detector. To keep the consistency of guest system, each of the VMs is equipped with 512MB memory and an emulated hard drive with 2GB capacity. The guest OS is Microsoft Windows XP service pack 3. After each execution round, the hard disk image will be replaced by a clean one, thereby removing any modification to system status by the sample.

```
EIP:opcode                                  | EIP:opcode
                                            |
00454a80: mov  0x4(%esp,1),%edx             | 00454a80: mov  0x4(%esp,1),%edx
00454a84: mov  0x8(%esp,1),%ecx             | 00454a84: mov  0x8(%esp,1),%ecx
00454a88: test $0x3,%edx                    | 00454a88: test $0x3,%edx
00454a8e: jne  0x454acc                     | 00454a8e: jne  0x454acc
00454a90: mov  (%edx),%eax                  | 00454a90: mov  (%edx),%eax
00454a92: cmp  (%ecx),%al                   | 00454a92: cmp  (%ecx),%al
00454a94: jne  0x454ac4                     | 00454a94: jne  0x454ac4
00454ac4: sbb  %eax,%eax                    | 00454a96: or   %al,%al
00454ac6: shl  %eax                         | 00454a98: je   0x454ac0
00454ac8: add  $0x1,%eax                    | 00454a9a: cmp  0x1(%ecx,1),%ah
00454acb: ret                               | 00454a9d: jne  0x454ac4
                                            |
              ....                          |               ....
                                            |
0042dbcf: cmpl $0x0,0xfffffb9(%ebp)         | 0042dbcf: cmpl $0x0,0xfffffb9c(%ebp)
0042dbd6: jne  0x42dbe7                     | 0042dbd6: jne  0x42dbe7
0042dbe7: push $0x482c84                    | 0042dbd8: push $0x482c90
0042dbec: call 0x42c130                     | 0042dbdd: call 0x42c130
      //printf("NotBochs\n");               |      //printf("Bochs HD emulation\n");
              ....                          |               ....
                                            |
      Trace in Xen , QEMU                   |      Trace in Bochs
```

Figure 5. The branches in the traces of the Bochs hard disk check.

There is no commonly used benchmark for VM-aware detection, yet the known VM check samples and real VM-aware malware were collected to evaluate the correctness of detection. Both the VM-aware techniques and the real malware used herein were also used in the related work. Two types of VM-aware packers are detected and two case studies of known VM-aware techniques are demonstrated to show that Divergence Detector can correctly identify the VM-awareness.

A. Evaluate Correctness with VM-Check Sample

Seven known types of VM check samples are listed in Table 1. Each sample uses a different kind of checking techniques to detect the specific virtual machines. The third column shows the memory address of branches in the VM check. The target platforms of the VM check are listed in the fourth column. The QEMU hard disk check is aware of the QEMU and Xen simultaneously. Since Xen use part of QEMU's code for hard disk emulation, they both have the same hard disk identity. The VM check can be used for both emulation-resistance (QEMU) and virtualization-resistance (Xen). QEMU and Xen cannot be differentiated in this case. Therefore, a third VM Bochs is needed.

We choose two samples as case studies to explain the VC discovery. The selected samples are Bochs hard disk check and QEMU multi-rep prefix check, respectively.

1) Sample1: Bochs HardDisk Check: Figure 4 shows the source code of the sample. It used the function get_hdver() to get the identity of the hard disk and then compared the returned string with a string "Generic 1234" which is the default hard disk identity of Bochs. If the strings are identical, the sample will output "Bochs HD emulation" in the standard out. Otherwise, it prints "Not Bochs". The execution traces of this sample are listed in Figure 5. Figure 5 contains two execution traces, one for Xen and QEMU, and the other for Bochs. There are two branches in this binary execution traces. The branch (0x454a94) underlined on line 7 in Figure 5 is an SC, located inside the function *strcmp*(). The divergence occurs because the identities of the hard disk in Xen and QEMU are not the same as that in

```
004012f1: pushl  %fs:0x0                     004012f1: pushl  %fs:0x0
004012f8: mov    %esp,%fs:0x0                004012f8: mov    %esp,%fs:0x0
004012ff:     illegal instruction            004012ff: repz   nop
(Exception happened)                         0040130f: mov    %esp,%eax
(Exception handler executes)                 00401311: mov    %eax,%fs:0x0
00401290: push  %ebp                         00401317: add    $0x8,%esp
00401291: mov   %esp,%ebp                    0040131a: movl   $0x403021,(%esp,1)
00401293: sub   $0x8,%esp                    00401321: call   0x401870
00401296: movl  $0x403000,(%esp,1)
0040129d: call  0x401870
        Xen , Bochs                                     QEMU
```

Figure 6. VC of the QEMU multi-rep prefixes check

Table 2. Evaluation Result of Real Malware Samples

Malware labels	Address of VC
Trojan-Dropper. Win32. Agent.mu (tElock packed)	0x4134f7:***icebp***
Trojan-Downloader. Win32. VB.ang (tElock packed)	0x4144de:***icebp***
Backdoor.Win32.Rbot.ahst (Armadillo packed)	0x466945:je0xb013a7: ***iret***
Trojan.Win32.Pakes.bmf	0x40917c:jmp*%edx

Bochs. The second branch (0x42dbd6) on line 13 in Figure 5 is a VC, located at the branch which decides to print “Bochs HD emulation” or “Not Bochs”. The EIP 0x42c130 is the address of function call *printf*().

2) *Sample2: QEMU multi-rep prefix check:* This sample program was collected from the appendix of Ether. It utilizes the emulation bug in QEMU by placing fifteen rep prefixes before a single-byte instruction (NOP). This makes the total length of the opcode to be 16 bytes long which exceeds the maximum instruction length of ia-32 instruction set, but it is a valid instruction format in QEMU. While the sample is executed in Xen and Bochs, an exception is raised and the handler in the sample was triggered. In contrast, the sample was executed normally in QEMU, printing “QEMU Detected”. Figure 6 shows the divergence point discovered by our system.

B. *Evaluate Correctness with VM-aware Malware:*

We take three real malware packed with tElock and Armadillo, which are regarded as emulation-resistant packers, to evaluate our system. Moreover, we collected another suspicious VM-aware malware which can crash the QEMU emulator. Table 2 shows the results of the experiment. The labels of malware are achieved from Kaspersky Anti-Virus Database [29]. Column 2 is the memory address of a VC. The first two samples in Table 2 use the VM-aware packer *tElock*. It can detect QEMU by an undocumented opcode “icebp,” which is used for hardware-level debugging. On modern machines, the instruction will raise an interrupt with the vector of 0x1. Instead, QEMU uses this instruction to debug itself. While executing the opcode icebp, QEMU will stop emulation to wait for user inputs. The malware can be aware of QEMU environment by trapping the interrupt thrown by real hardware. The tElock is applied in Trojan-Dropper.Win32.Agent.mu and Trojan-Downloader.Win32.VB.ang.

Table 3. Uncertainty elimination of Benign Samples

	Before elimination	**After *n* execution round**	
Benign Samples	***Number of False Positives***	***Number of False Positives***	***Execution times n***
Notepad	0	0	1
Wordpad	5	0	2
Microsoft Messenger	6	0	2
Putty	3	0	2
Firefox	22	0	4

These malware programs only executed 105,839 instructions in QEMU, but executed more than 2,968,000 instructions in Xen and in Bochs which implies that the malware programs stopped unpacking when it was aware of the presence of QEMU. The malware packed with Armadillo uses an illegal opcode to crash the QEMU. Divergence Detector discovered the VM-awareness of these malware programs.

C. *Evaluate Uncertainties Reduction*

One of our contributions is to reduce uncertainty by the comparison of multi-execution code coverage in a generic way. First of all, we examine the benign programs which do not need to detect the execution environment, and the result was as expected. Secondly, the uncertainty generated by random number, which is not one of the VC checks, was examined to show that our false positive rate is as expected.

1) *Evaluate Irrelevant DPs Reduction with Benign Samples:* This experiment was the examination of five benign applications. Most of them are multi-thread since they can easily generate different code coverage to produce the UADPs. Moreover, the samples Microsoft Messenger, Putty and Firefox have the network communications. Since the time and the value of network inputs are unpredictable, the timing of packets sent and received is an uncertainty. To show the effect of uncertainty reduction, we executed the application in both Ether and QEMU, and evaluate the differences of code coverage for finding VMDPs. Table 3 shows that Firefox, which is more complex than other applications, has the more divergences which are irrelevant to VM checks. Note that, the divergences are the branches of the VM-awareness or the others. However, in our system, these branches to UADPs can be reduced in 4 execution rounds. The result of the experiment is as expected that the benign applications should not have any VC for VM-awareness.

2) *Evaluate Irrelevant DPs Reduction with random branch samples*: The evolved VM-aware malware may use many irrelevant random branches to produce the false positives against analyzing its VM awareness. We used the sample code in Figure 1, and inserted 5,000 uncertain branches that repeat lines 2 and 3. The predicted false positive rate is $(2^k-2)p^{nk}$, where k is the number of VMs, p is the probability of true statement branching and n is the number of execution rounds. In this experiment, k = 2, p = 0.5 and n = 1 to 10. The expected values are listed in Table

4, second row. For example, the expected value of only 1 execution round is $2(1/2)^{2\times1}\times 5{,}000+1=2{,}501$. The probability of each uncertain branch is 0.5. There were 5,000 uncertain branches and one VC which always cause the divergence of trace. In the first row, the result of execution round n is the average of 10,000 times execution. As the table shows, the maximum false positive rate is exponentially decreased with respect to the number of multi-executions. The result is very close to the expected value as shown in Table 4. From the result of this experiment, we demonstrate that UADPs can be removed effectively in our system.

Table 4.The predicted value of uncertainty-attributed branch.

Execution round n	1	2	3	4	5	6	7	8	9	10
Divergence Detector	2,483	620	157	39	13	4	2	1	1	1
Expected value	2,501	626	157	40	10	2	1	1	1	1

VII. Conclusion

As virtual machines are widely used for malware analysis in recent year, a new kind of malware arises for analysis-resistance. These malware programs can probe the execution environment so that they can behave harmlessly in VM-based analysis tools. Some of the VM-aware techniques introduced in this paper are actually applied to real malware. Conventional schemes are not suitable to address the VM-aware malware for the following reasons. First, the false negatives will occur because the selected VM platforms cannot identify some VM checks, such as QEMU hard disk check. The behaviors of execution in both Xen and QEMU are identical while the malware examining its execution environment. The other problem with conventional schemes is that the uncertainties will incur high false positives because the branch irrelevant to VM awareness will be also treated as VM checks. The uncertain divergences of execution frequently occur in benign applications such as the Firefox, Microsoft Messenger and Microsoft Office Word. These applications were tested and verified to demonstrate the results.

To resolve the first problem, we integrated three common VM platforms, which are QEMU, Bochs and Xen, into our system for comprehensive VM-aware detection. We discover that a single VM check cannot detect all VMs at the same time. Secondly, the approach for uncertainty reduction was proposed for decreasing uncertainty-attributed false positives. By multi-execution rounds, code coverage comparison can filter out the instructions not being launched in every round. The reduction is effective to remove uncertainty factors such as multi-thread programming, random branching, and network communication.

Seven kinds of VM-aware techniques and four real malware programs were investigated and used for evaluation. The proposed Divergence Detector successfully discovered the VM-aware divergent points in each test case. The traces in different kinds of VMs were presented to illustrate the discovery of VM awareness. Moreover, we also evaluated the false positive rate by inserting 5,000 random branches in the sample. The maximum of the false positive rate in our experiments was close to the prediction formula $\boldsymbol{Pmax} = (2^k - 2)p^{nk}$ as expected. With sufficient execution rounds, the proposed Divergence Detector can reduce a large amount of uncertainty-attributed false positives and the maximum false positive rate is exponentially decreased with respect to the number of multi-executions.

VIII. ACKNOWLEDGMENT

This work was supported in part by TRUST Center of UC Berkeley, National Science Council, NCP, ITRI, III, Chung Shan Institute of Science and Technology, Chunghwa Telecomm., Bureau of Investigation, HTC, Promise Inc., D-Link, the International Collaboration for Advancing Security Technology (iCAST) and Taiwan Information Security Center (TWISC), Ministry of Education (R.O.C.), respectively.

IX. Reference

[1] Newsome, J. and Song, D. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Network and Distributed System Security Symposium (NDSS) (2005).

[2] Yin, H., Song, D., Egele, M., Kruegel, C. and Kirda, E. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. Proceedings of the 14th ACM conference on Computer and communications security (2007), 116–127.

[3] Martignoni, L., Stinson, E., Fredrikson, M., Jha, S. and Mitchell, J. A layered architecture for detecting malicious behaviors. Recent Advances in Intrusion Detection 78–97.

[4] Linn, C. and Debray, S. 2003. Obfuscation of executable code to improve resistance to static disassembly. Proceedings of the 10th ACM conference on Computer and communications security (2003), 290–299..

[5] Royal, P., Halpin, M., Dagon, D., Edmonds, R. and Lee, W. 2006. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual (2006), 289–300..

[6] Anubis: Analyzing unknown binaries, http://anubis.iseclab.org/.

[7] CWSandbox - an automated malware analysis tool, http://www.cwsandbox. org/.

[8] Willems, C., Holz, T. and Freiling, F. 2007. Toward automated dynamic malware analysis using cwsandbox. IEEE Security & Privacy. (2007), 32–39.

[9] Norman sandbox whitepaper, http://download.norman.no/whitepapers/whitepaper_Norman_SandBox. pdf.

[10] ThreatExpert - automated threat analysis tool, http://www.threatexpert.com/.

[11] Bayer, U., Kruegel, C. and Kirda, E. 2006. TTAnalyze: A tool for analyzing malware. 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR) (2006)..

[12] Dinaburg, A., Royal, P., Sharif, M. and Lee, W. 2008. Ether: Malware analysis via hardware virtualization extensions. Proceedings of the 15th ACM conference on Computer and communications security (2008), 51–62.

[13] Ferrie, P. 2007. Attacks on more virtual machine emulators. Symantec Advanced Threat Research. (2007).

[14] Raffetseder, T., Kruegel, C. and Kirda, E. Detecting system emulators. Information Security. 1–18.

[15] Chen, X., Andersen, J., Mao, Z.M., Bailey, M. and Nazario, J. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. IEEE International Conference on Dependable Systems and Networks (2008), 177–186.

[16] Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D., di Milano, U.S. and di Udine, U.S. 2009. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. Proceedings of the USENIX Workshop on Offensive Technologies (WOOT) (2009).

[17] Carpenter, M., Liston, T. and others 2007. Hiding virtualization from attackers and malware. IEEE Security & Privacy. (2007), 62–65.

[18] Rutkowska, J. Red Pill... or how to detect VMM using (almost) one CPU instruction. Retrieved, from http://invisiblethings. org/papers/redpill. html.

[19] Neiger, G. 2006. IntelŴVirtualization Technology: Hardware Support for Efficient Processor Virtualization. Intel Technology Journal. 10, (Aug. 2006).

[20] Rutkowska, J. and Tereshkin, A. 2007. IsGameOver () Anyone. Black Hat, USA. (2007).

[21] Kang, M.G., Yin, H., Hanna, S., McCamant, S. and Song, D. 2009. Emulating emulation-resistant malware. Proceedings of the 1st ACM workshop on Virtual machine security (2009), 11–22.

[22] Open Source QEMU Emulator, http://wiki.qemu.org/.

[23] Bochs: The Open Source IA-32 Emulation Project, http://bochs.sourceforge.net/

[24] VMware Virtualization Software, http://www.vmware.com/

[25] King, S.T. and Chen, P.M. 2005. Backtracking intrusions. ACM Transactions on Computer Systems (TOCS). 23, 1 (2005), 51–76.

[26] Wagner, D. and Soto, P. 2002. Mimicry attacks on host-based intrusion detection systems. Proceedings of the 9th ACM Conference on Computer and Communications Security (2002), 264.

[27] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P. and Saxena, P. 2008. BitBlaze: A new approach to computer security via binary analysis. Proceedings of the 4th International Conference on Information Systems Security. (2008), 1-25.

[28] Beizer, B. 2002. Software testing techniques. Dreamtech Press.

[29] Kspersky Anti-Virus. http://www.kspersky.com

[30] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A. 2003. Xen and the art of virtualization. Proceedings of the nineteenth ACM symposium on Operating systems principles (New York, NY, USA, 2003), 164–177.

[31] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In Proceedings of the 17th Annual Network and Distributed System Security Symposium (2010).

[32] George W. Dunlap , Samuel T. King , Sukru Cinar , Murtaza A. Basrai , Peter M. Chen, ReVirt: enabling intrusion analysis through virtual-machine logging and replay, Proceedings of the 5th symposium on Operating systems design and implementation, December 09-11, 2002, Boston, Massachusetts.

[33] Chow, J., Lucchetti, D., Garfinkel, T., Lefebvre, G., Gardner, R., Mason, J., Small, S., Chen, P.M., Multi-stage replay with crosscut. In: Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (2010).