**OBJECTIVE**

For this class project, we plan to work on the system integration and evaluate the performance in terms of system scalability. Specifically, our system is the behavior-based email filtering system. We already have the prospective design of the system and the detection algorithm. Briefly, we use the behavioral patterns that we have learnt from the malware in the past to detect new and unknown malware. We are going to improve the scalability on Xen by applying the flash cloning [2] and copy-on-write [3] techniques.

**BACKGROUND**

The major deficiency in the traditional signature-based systems, e.g. antivirus software (AV), is that their signatures are not tolerant to malware variants and there is a lag in the signature generation. The traditional signatures are code-based. As a result, with a simply change in the code, e.g. add a NOP instruction, AV often needs to generate the corresponding signatures. We mitigate this problem by using the behavior-based signatures for the dynamic detection. According to Kaspersky’s report [5], the newly-born malware in 2006 can be categorized into the known malware classes such as spyware, Trojan, virus, worm. Malware in each general malware class semantically shares the similar behavior. Our behavioral signatures, extracted from the malware in the past, should be able to detect new and unknown malware, as opposed to antivirus software.

The malware behavior is represented in terms of system call sequences and the associated arguments. In particular, our monitoring engine records the file system activity, registry access requests, process creation/deletion, image loading, network connectivity, and the modification in IDT. To monitor the program execution, we explicitly open every email attachment in a clean virtual machine in order to prevent any interference. This can also be extended to examine all embedded links in email messages. Then, we pass the execution records to the pattern discovery module in order to find the pattern candidates. We propose a simple criterion to select the patterns for our behavioral signatures, which will be effectively used to detect the malware. We have run some experiments to determine the detection performance in terms of accuracy and runtime overhead. The experimental suggests that with proper parameter setting our algorithm can be implemented efficiently in runtime while it also maintains high detection rate and low false positive rate. In this project, we will focus on the efficient design of our malware detector.

**RELATED WORK**

Despite many efforts in both industry and academia, malicious email still remains a major challenge. One of the reasons is that many solutions are not practical to adopt in the real system, i.e. email server. The major impediment is the performance degradation, including both detection accuracy and run-time overhead. Specifically, our design goal is try to balance between the detection correctness and the cost of operation.
Many commercial products i.e. antivirus (AV) programs [e.g. 8,12,13], can run in real time with zero false positive, but fail to discover new malware. This deficiency arises from the nature of byte-based signatures used in AV. Similarly, our detection scheme generates and uses the signatures. The difference is that our signatures are based on invariant behavior found in malware. The behavioral signatures are more resilient to the malware obfuscation. Theoretically, behavior-based approach should yield better detection precision. In the downside, behavior-based detection potentially encounters the problem of high false positive. Yet, depending on the detection techniques, the false positive can be significantly reduced. According to our experimental results, our detection algorithm yields zero false positive given the typical users applications. In the behavior-based detection paradigm, there are two major approaches – that is static and dynamic analysis. The main advantage of static analysis [e.g. 19,20,22,23,28] is the completeness because it examines every single possible path in a program. One disadvantage is that the static analysis of source code or binary takes a substantial amount of time, inhibiting the deployment in real time. Instead, some works [e.g. 20,28] derive the static control flow graph (CFG) from the binary, which results in high runtime monitoring overhead.

Forrest [19] is one of pioneers on the static analysis approach. They create the normal models to detect malware using the deviation from the models, specifically called anomaly detection. One benefit of the anomaly detection is to have broader coverage of the detection. With a careful normal model construction, the false positive should be zero. One drawback in this approach is that the normal model is closely tied with the application. In order to indicate the deviation from a particular application execution, they need to have normal representations of that application. The number of representation models depends on the complexity of applications and a number of applications. This number can be considerably large. Moreover, it is very difficult to enumerate the complete normal execution patterns of an application in a short period of time. Alternatively, we choose the dynamic analysis due to significantly less runtime overhead. Instead of detecting based on the deviation, we construct a set of prohibited patterns.

Nevertheless, Gao [17] proposes a technique that constructs the execution graph for anomaly detection, which is equivalent to CFG in the static analysis. King [24] performs dynamic analysis to construct the dependency graph for backtracking the intrusion root cause. Similarly, we perform the dynamic technique to monitor the dynamic behavior of a program execution, which are used to construct the misuse signatures, as opposed to the normal patterns. Seemingly, our technique potentially has lower detection coverage. According to the experimental results in Section 6, our technique can detect more than 95% of unforeseen malware. Specifically, our detection signatures are constructed from the observed system calls and associated arguments. Creating the behavioral patterns solely from the system calls has been done in previous works [e.g. 20,22]. Several works [e.g. 18,20,24,31] improves the detection performance by incorporating the called arguments. One major difference between our work and the existing works is that our patterns do not necessarily derive from consecutive system calls. However, the time ordering in the sequences is strictly preserved. This is important in our detection scheme because we use the patterns as the misuse signatures. This way, our behavioral signatures are more resilient to the malware obfuscation e.g. simply add irrelevant system call.

CWSandbox [27], a malware analyzer, is similar to our work, except that they utilize the virtual machine, i.e. VMware virtual machine, without any further modification. Furthermore, their detection scheme considers no behavioral patterns. Similarly, Paladin [11] observes the malware execution in VMware virtual machine with the narrow scope of the rootkit detection.

SEES [16] is a commercial system that provide the secure execution code environment. Basically, users can remotely execute any suspicious code in the SEES server. The limitation on the number of remote session can cause scalability problem since each user has to initiate a remote terminal session to run their codes. Moreover, the SEES server executes each piece of code with a low privilege. In contrast, we allow a code execution under the administrator account in order to observe the worst-possible consequences. Another technique to observe a program execution is to run a target code in a simulator [e.g.26]. Sidiroglou [10] proposes a similar email filtering system, but they only focus on the worm detection with the limited monitoring mechanism.

### TENTATIVE PLAN

<table>
<thead>
<tr>
<th>Week</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/15</td>
<td>Complete the detailed system design</td>
</tr>
<tr>
<td>10/22</td>
<td>Stabilize the modified Xen and separately implement the malware detector module</td>
</tr>
<tr>
<td>10/29</td>
<td>Individual test the functionality of each part</td>
</tr>
<tr>
<td>11/5</td>
<td>Integrate all parts together and design the experiments</td>
</tr>
<tr>
<td>11/12</td>
<td>Evaluate the system performance with synthesis data</td>
</tr>
<tr>
<td>11/19</td>
<td>Create a testbed i.e simulate the email traffic and evaluate the system with this testbed</td>
</tr>
<tr>
<td>12/10</td>
<td>Wrap up the project and write the report</td>
</tr>
</tbody>
</table>

### REFERENCE


