

# Malware Analysis on the Cloud: Increased Performance, Reliability, and Flexibility

Graduate Capstone

Master of Science - Computer Science and Systems

Michael Schweiger<sup>1</sup>, Sam Chung<sup>1</sup>, and Barbara Endicott-Popovsky<sup>2</sup>

<sup>1</sup>Institute of Technology, University of Washington

<sup>2</sup>Center for Information Assurance and Cybersecurity, University of Washington

## Abstract

Malware has become an increasingly prevalent problem plaguing the Internet and computing communities. According to the 2012 Verizon Data Breach Investigations Report, there were 855 incidents of breach reported in 2011 with a massive 174 million records compromised in the process; 69% of those breaches incorporated malware in them some way, which was 20% higher than those breaches that used malware in 2010.

Clearly, the need to effectively and efficiently analyze malware is needed. Unfortunately, there are two major problems with malware analysis; Malware analysis is incredibly resource intensive to deploy en masse, and it tends to be highly customized requiring extensive configuration to create, control, and modify an effective lab environment. This work attempts to address both concerns by providing an easily deployable, extensible, modifiable, and open-source framework to be deployed in a private-cloud based research environment for malware analysis.

Our framework is written in Python and is based on the Xen Cloud Platform. It utilizes the Xen API allowing for automated deployment of virtual machines, coordination of host machines, and overall optimization of resources available. Each part of the malware analysis process can be identified as a discrete component and this fact is heavily relied upon. Additional functionality and modifications are completed through the use of custom modules. We have created a sample implementation that includes basic modules for each step of the analysis process, including traditional anti-virus checks, dynamic analysis, tool output aggregation, and classification. Each of these modules can be expanded, disabled, or completely replaced. We show, through the use of our sample implementation, an increase in the performance, reliability, and flexibility compared to an equivalent lab environment created without the use of our framework.

# 1 Introduction

Malware defense and detection is an incredibly hot topic among computer security researchers, industry, government, and even the lay population. There is merit behind the hype as malware has become an increasingly prevalent problem throughout the Internet and the computing community at large. Motivation has shifted from general mischief to that of major financial gain, prompting an explosive growth in the rate of new malware released and an advancement of the techniques utilized by malware authors to thwart analysis and detection [MICR07],[PALE10],[EGEL08]. “The widespread diffusion of malicious software is one of the biggest problems the Internet community has to face today. [PALE10].” As we can see from trends in malicious software, we are clearly losing the battle between those who are detecting and defending against malicious code and those who are creating and releasing it.

Traditional malware detection measures have relied on signatures and signature-based anti-virus/malware products [MICR07]. This approach has some serious drawbacks such as a severe time delay between the release of malware into the wild and it coming to the attention of analysts; there is also the delay for analysis, signature creation, and signature dissemination. This makes traditional anti-virus methods less effective and has brought to the attention of security researchers the necessity of researching other methods for analyzing and detecting malware [PALE10],[EGEL08],[IDIK07],[MART09].

This explosion of malicious software, when coupled with the inefficiencies of signature-based detection schemes, is the primary motivation behind much of the work done in this field. It has been shown that performing the fine-grained analysis required to obtain usable results and monitor malicious software during execution has a non-negligible impact on the resources of the executing system [PALE10]. Due to this overhead, research into moving the analysis environments off of end-user host machines and into the cloud has led to the creation of existing cloud-based malware analysis frameworks [PALE10]. Existing frameworks however, such as the work proposed in [PALE10], [MART09], attempt to solve the problem by shifting the majority of the computational load into the cloud but do not account for the aggregation of results from multiple analysis engines [MART09].

Our work attempts to answer the following question: how can we overcome the shortcomings of current cloud-based analysis frameworks in order to provide increased performance, accuracy, and flexibility? We attempt to provide a solution that combines the best aspects of existing frameworks while maintaining a modular design that is capable of growing to match additional requirements at a later time.

We distinguish our work from other efforts in this field by producing a new system based upon existing work in an integrated fashion. Our work sets out to provide a holistic system that integrates previous efforts with dynamic analysis tools with modules for merging the results from the different engines into one resource and modules for using that resource with various data classification techniques to produce a final answer (malicious or benign). We begin with developing a new framework that allows modules to be added to fulfill each of these goals and then produce a proof-of-concept system based on that framework. We make the assumption that there is effectively an unlimited resource pool in cloud computing environments. To this end, we propose the following contributions to the field of malware analysis:

1. We propose a system loosely based on previous work in this field that can be extended with additional modules for added functionality, such as adding a new tool that is not included in the example implementation.
2. We propose modules to apply data mining classification techniques to produce some final answer to the original question of “Is this software malicious?”
3. Finally, there seems to be a scarcity of open-source solutions to address this problem; this work will be released as open source (the exact license has not been determined at this point) upon completion of the alpha testing phase.

This report is organized in the following fashion. Section 2 introduces background information, including an explanation of malware, current non-cloud based analysis tools, and the limitations inherent in these tools and techniques. We discuss related work in section 3. Section 4 explains our approach behind designing, developing, and releasing our framework, as well as the implementation created for evaluation purposes. Section 5 provides an evaluation of our framework and implemented system to address performance, flexibility, and reliability concerns. Finally, we conclude with possible directions for future research in section 6.

## **2 Background**

Malware, or malicious code, includes viruses, worms, trojan horses, some rootkits, spyware, adware, bots, and pretty much any software that “deliberately fulfills the harmful intent of an attacker” [EGEL08]. Malware analysis can be broadly broken down into static and dynamic analysis [EGEL08] and efforts have been made to advance both techniques [PALE10]. In this section, we provide background information on malware analysis and classification, including types of analysis, limitations of analysis, and classification.

### *2.1 Types of Malware Detection*

In this section, we discuss malware detection techniques that rely on the finished output of prior analysis, or the results of concurrent analysis for faster detection. In [IDIK07], the authors discuss detection as either signature-based or anomaly-based, with specification-based being a subcategory of anomaly-based (see figure 1 for additional details). All of the detection techniques rely on the accuracy of analysis techniques. Analysis techniques and methods are the focus of this paper; we present this section only for completeness.

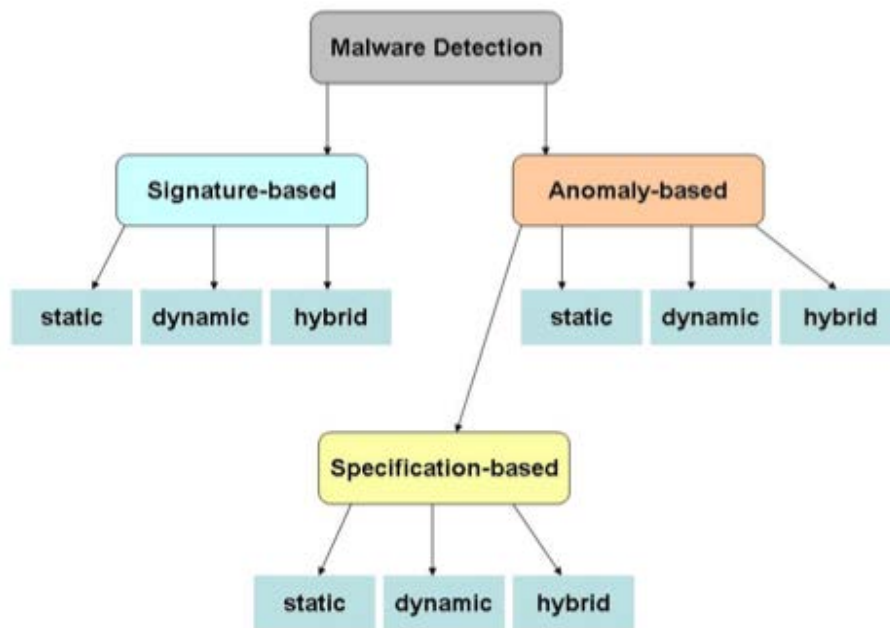


Figure 1: A classification of malware detection techniques [IDIK07]

Signature-based detection is the traditional method that has been employed for decades by anti-virus products. “An anomaly-based detection technique uses its knowledge of what constitutes normal behavior to decide the maliciousness of a program under inspection...Specification-based techniques leverage some specification or rule set of what is valid behavior in order to decide the maliciousness of a program under inspection” [IDIK07].

## 2.2 *Types of Malware Analysis*

Antivirus signatures must be manually created by human analysts that rely on the output from malware analysis efforts to facilitate the signature creation process [EGEL08]. We use the term “analysis” to mean monitoring or examination. This is distinguished from malware detection, which uses the results of prior or concurrent analysis (the output or signature) to make a decision as to the malicious content of a program.

Malware analysis can be conducted in two broad categories: static and dynamic analysis. The output of an analysis, as we use the term, is a report of the examined behaviors (for dynamic analysis) or the code structure (for static analysis).

### 2.2.1 Static Analysis

Static analysis does not execute the sample, but rather inspects the static code, hence the name [PALE10], [EGEL08]. “Static solutions reason on the binary code of a suspicious application without actually

running it, so they can consider all of the behaviors the program may exhibit at run-time” [PALE10]. Static solutions are still employed, but dynamic solutions are quickly becoming the method of choice for analysts due to theoretical limitations in static analysis [PALE10],[EGEL08],[IDIK07], [VAN08].

### 2.2.2 Dynamic Analysis

Dynamic analysis is complementary to static analysis; while static analysis examines the code structure without executing it, dynamic analysis executes the sample in an isolated sandbox environment while monitoring the behavior exhibited [PALE10], [EGEL08]. “Analyzing the actions performed by a program while it is being executed is called dynamic analysis” [EGEL08]. This is typically achieved through the monitoring of function calls, including API and system calls and is usually done through the use of function hooking; function call hooking causes the execution flow to reroute through specialized functions for the purpose of logging (or other activities) before possibly calling the original function [EGEL08]. The information logged will depend on the configuration of the tool used.

## 2.3 *Limitations of Analysis*

If there were no limitations with the existing techniques, methods, and tools used for malware analysis, research into the topic would have plateaued already and progress would be stifled. Unfortunately, there are considerable limitations with each method. In general, both methods and multiple tools are used in conjunction for more reliable results. In this section, we briefly present some of the main issues with static and dynamic analysis techniques.

### 2.3.1 Issues with Static Analysis

“The application of static analysis techniques to malicious software poses serious theoretical problems that limit the overall precision of the final results” [PALE10]. These limitations include code obfuscation, variable generation at run-time, code packing, and polymorphic code with opaque constants [PALE10], [YOU10], [MO07-2]. In many cases, it can be impractical or impossible to know what a piece of code does by just examining the static code. In addition to the limitations of the analysis, performing static analysis requires a fair amount of knowledge and experience with reverse engineering due to the fact that the source code is generally not available for analysis [EGEL08], [NSAC11].

Static analysis tools become much less effective when the author of a piece of malware takes certain steps to prevent static analysis, such as code obfuscation. In [MO07-2], the authors discuss two techniques to obfuscate the code from static analysis by dynamically generating, with 100% accuracy, an address or data constant at runtime. The researchers referred to the output of either of these two algorithms as an “opaque constant”, which cannot be statically determined without extensive computational overhead [MO07-2]. The logic behind why these methods work is complex, and the effect is substantial. Short of solving the 3SAT problem<sup>1</sup> (which is considered NP-hard) [MO07-2] or running the process through a recursive code simulator to actually simulate the process in order to determine what the behavior is (not

---

<sup>1</sup> We will not go into detail about the 3SAT problem, except to note that it is considered NP-hard and is related to opaque constants. For additional information about the problem, please refer to [MAAN12].

quite the same as dynamic analysis and can be quite resource intensive), these opaque constants effectively prevent any form of effective static analysis.

### 2.3.2 Issues with Dynamic Analysis

While the problems that are present with static analysis usually deal with the code being difficult or impossible to gather much useful information from, the primary concerns with dynamic analysis revolve around malware behavior modification due to detection of the analysis environment, safely containing the sample during analysis, getting complete results through complete code coverage during execution, and the computational overhead associated with dynamic analysis [PALE10][EGEL08],[IDIK07], [VAN08], [SUN11],[HUAN11],[SCHM11].

A major concern with dynamic malware analysis is that of hiding the analysis engine from the sample under review; certain malware will modify the exhibited behavior if an analysis engine is detected [SUN11]. Malware that runs at the kernel level, for instance, can detect any analysis environment that may be running in the kernel space (or at a lesser privilege) as well [PALE10]. This can lead to highly inaccurate results that may prevent a sample from being labelled as malicious, thus preventing the creation of a signature and prolonging the malware's effective lifetime before discovery. If no additional analysis is conducted to challenge the initial “non-malicious” result, the threat could go unchecked. Some researchers [PALE10] have labelled malware that fulfills this behavior as “next-generation”.

Another issue is dealing with the containment of the sample under analysis; many malware samples are quite powerful and steps need to be taken to ensure the safety of the analysis system, the network, and any other systems connected to the network (note that a connection to the Internet counts as the network) [PALE10], [EGEL08]. If virtualization technologies are used for analysis, as is usually the case, the hypervisor that sits between the virtual machine and the host OS can be an attack target if it is detected by the sample [TREN12]. If the hypervisor gets breached, the host system (which generally is much less restricted on the network) can become infected and allow the sample escape the sandbox. One recommendation when using dynamic analysis tools is to isolate, all the way at the physical hardware and network level, the testing environment from all other environments; don't use a production machine as an analysis machine and don't let it connect to the production network.

The final major concern is that of code execution coverage. This concern can be caused by the malware detecting the analysis engine and modifying behavior, or it could simply be caused by the proper trigger conditions to exhibit the behavior not being present or occurring [PALE10]. Take the Michelangelo virus as an example of the trigger condition not being present; it only delivers its payload on March 6 (Michelangelo's birthday) and is dormant every other day of the year [MO07-1]. Efforts have been made to correct this through the recursive execution of the sample. In [MO07-1], the authors present a technique of monitoring branch points so that the program can be reproduced for each branch. If each path in every branch statement is followed, it should produce 100% code coverage. There is additional work to be done on this topic.

## 2.4 *Classification*

Classification refers to using known attributes of data to predict attributes about unknown/unseen data. This is also called supervised learning and is typically done using a training set of data that has the attributes of interest labelled to generate models, such as support vector machines or decision trees, and then using those models to predict the value of the attribute on future data; often, an overlap of the training data is used as test data to check the accuracy of the model prior to use [HAN12].

We can apply classification techniques to the output from the dynamic and static analysis of malware from various tools. These tools produce some output, usually logs of text that describe, using some notation, the behaviors present and exhibited by an examined piece of code. These behaviors can be subsequently checked against a model that has been constructed using previous data. In order to produce a model, we need to gather a large quantity of known malicious software of various malware families that are designed for different purposes, analyze each of them in turn, and use the log data as the train/test data to build the model or models. The more data that goes into the model, the more likely it is to be more accurate; care must be taken, however, to not overfit the data [HAN12].

### **3 Related Work**

To the best of our knowledge, no modular, open-source cloud-based framework for malware analysis currently exists. We have taken general inspiration from some other works however; we will briefly discuss some of these inspiring works here.

Moving the analysis environment into the cloud is not a new concept. Martignoni, et al. produced a framework for performing malware analysis in the cloud while mitigating some of the flaws involved with synthetic analysis environments [PALE10], [MART09]. Zheng, et al. produced a system for shifting the majority of antivirus processing into the cloud by using a lightweight host agent to proxy files into the cloud for analysis by a cluster of traditional detection engines and an artificial immune system to detect malicious activity [ZHEN10]. Cha, et al. developed SplitScreen, a distributed malware detection system that allowed for signature based detection methods to be greatly increased in performance through distributed computing [CHA11]. Finally, Schmidt, et al. produced a framework for preventing rootkits during malware detection in cloud computing environments [SCHM11].

The systems from Zheng, et al. and Cha, et al. are interesting and will be modeled in the form of a pre-existing online virus service known as VirusTotal [VIRU13]. Our work will focus primarily on the efforts from Martignoni, et al. and Schmidt, et al. due to the relatively high level of similarity between their respective work and our proposed work.

The works by Martignoni, et al. (henceforth referred to as System M for convenience) and Schmidt, et al. (henceforth referred to as System S for convenience) have a few weaknesses that our work attempts to correct. System M is not designed to be modular, does not deal with the outputs from multiple engines, does not deal with malware that gains root privileges, and is not open-source. System S does handle multiple analysis engines and moves the monitoring component outside the virtual machine to deal with malware that gains root privilege, but is still not modular in design nor open-source. Our work is open-source and modular to help deal with these problems. We attempt to address the lack of dealing with multiple engines by creating a custom module designed to combine multiple outputs from engines into

one format for analysis and a classification module to be able to use data mining techniques to predict if a file is malicious or not; we will still only have one analysis engine, but the processing module allows more engines to be added later (since it's modular) and it should either deal with them or be easily modifiable to deal with them.

### 3.1 Martignoni, et al. (System M)

One of the primary issues with synthetic analysis environments is the homogeneity of the systems; the likelihood that a particular trigger mechanism present in the code will be triggered is considerably lower than if the software is run on the users machines (in the wild) [PALE10]. The solution to this problem, as produced by Martignoni, et al. is to involve end-user interaction to create a larger range of activity present during analysis which would cause the program under examination to behave as if it is running directly on the end-users machine [PALE10], [MART09]. Software normally communicates with the operating system (OS) via system and API calls to obtain or set information such as active windows, file handles, system time, networking information, and so on. System M makes the assumption that software must utilize these calls to obtain the information and services needed for operation, so it attempts to mitigate the issue of homogenous environments by capturing a subset of these calls that is deemed safe and forwarding them to the users machines for execution; the software gets the returned result and can't detect that it was not a local execution [PALE10], [MART09].

The example malware used in the study from Martignoni was the Bancos trojan (others were used for evaluation, but not used as an example in the report); Figure 2 shows the psuedo-code of the trojan and the system calls being forwarded to the end-user for processing. Figure 3 shows the system call proxy setup at a closer level.

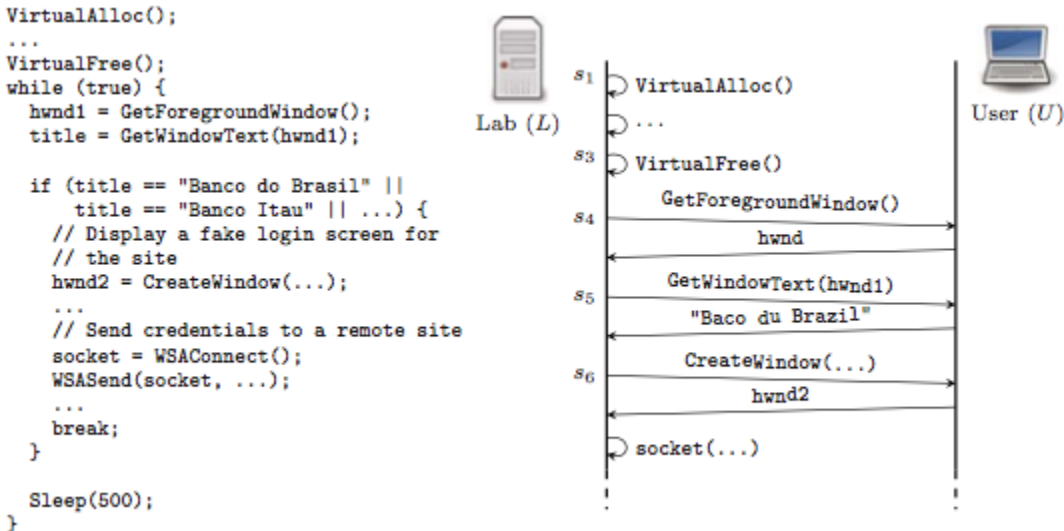


Figure 2: Pseudo-code of the Bancos trojan (left) and sequence diagram of system call forwarding between the lab and the user (right) [MART09].



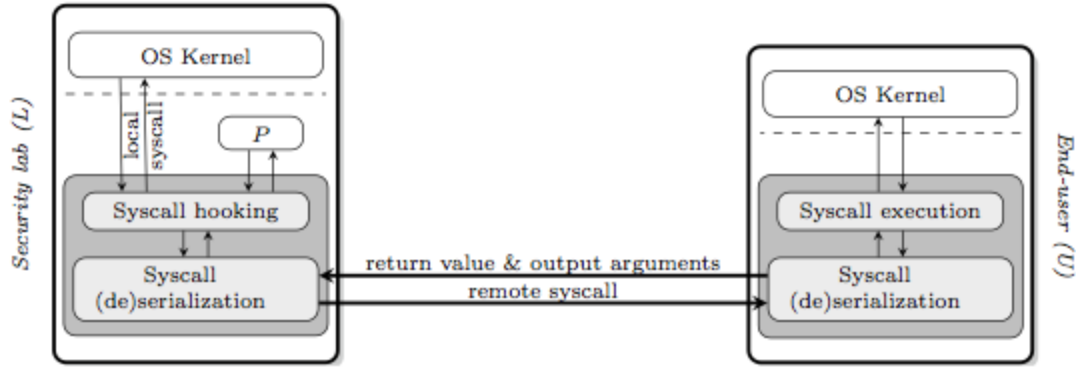


Figure 3: Capturing system calls and forwarding them to the user (P is the program under analysis) [MART09].

This setup effectively tricks the malware into believing that it is running within the environment of the end-user. Only those system calls deemed safe are forwarded. “Remote system calls are selected using a whitelist...Examples of the system calls we consider remote are: NtOpenKey, NtCreateKey (if the arguments indicate that the key is being opened for reading), NtOpenFile, NtCreateFile (if the arguments indicate that the file is being opened for reading), NtQuerySystemInformation, and NtQueryPerformanceCounter” [MART09]. Steps are taken to ensure the safety of the end-user.

While this framework is a starting place for our work, we see a few problems that exist as it stands. First, the assumption that a program can only communicate with the environment through API and system calls is only correct if the program stays in user space, rather than kernel space. If the code runs as root/admin and obtains kernel privileges, it can access memory data structures directly, thus bypassing the hooking mechanism used to proxy the calls. Our framework allows for building existing analysis tools into it, while System M did not address “the problem of correlating the results of multiple analyses” [PALE10], [MART09]. Our modular framework will allow for insertion of modules designed to correlate results if desired. It will also provide a logical placeholder if a module were designed to provide analysis between the VM hypervisor and the VM guest, which would fix the issue of root level malware accessing memory structure directly.

### 3.2 Schmidt, et al. (System S)

The work conducted by Schmidt, et al. targets two problems: malware detection in cloud environments and kernel rootkit prevention, the latter of which will be ignored [SCHM11]. The architecture for the malware detection system is similar to System M in the sense that it hooks system calls from the virtual machines via a small module in the kernel [SCHM11]. These hooked calls are relayed to a kernel agent, which then sends them to a scan proxy for further analysis; the scan proxy potentially resides outside the physical machine running the virtual machines involved in the analysis (it could be on the same physical machine but at the hypervisor level) and acts as a cloud-wide accumulator of data to send to the backend tools (see figure 4 for additional details) [SCHM11].

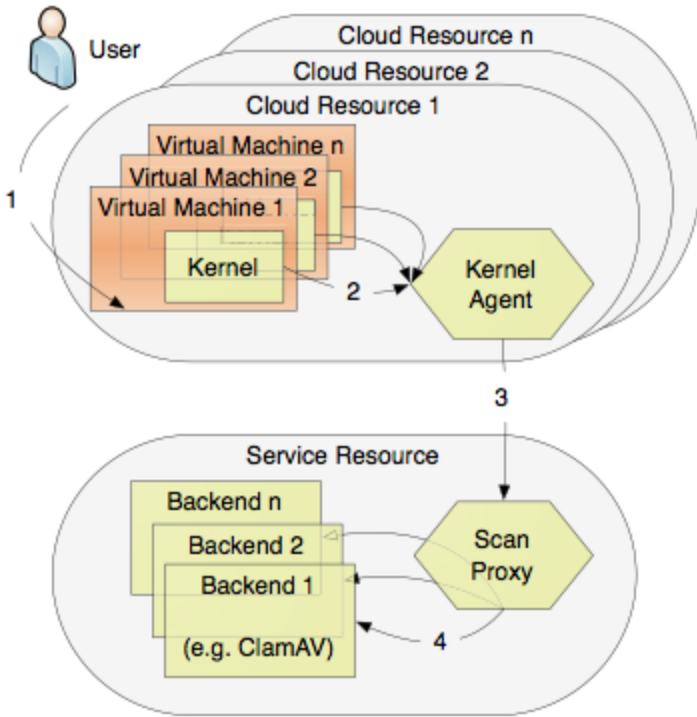


Figure 4: Overview of the framework by Schmidt, et al. [SCHM11].

This system seems very similar to System M but differs in one main way; the actual examination and analysis of the monitored actions are done outside the virtual machine, thus greatly lowering the chance of detection by the malware [SCHM11].

The second part of this work is the kernel rootkit prevention. Since we are essentially ignoring this part for our purposes, we will only briefly describe it here. The kernel rootkit prevention is done through the use of security levels concept in BSD which allows for read-only access to kernel memory while in a secure mode, as well as the idea of marking kernel modules as authorized (or not authorized) and storing that list in kernel memory; the actual marking process must be done in insecure mode so that when secure mode is enabled, those modules may be loaded into memory [SCHM11]. Figure 5 provides additional details.

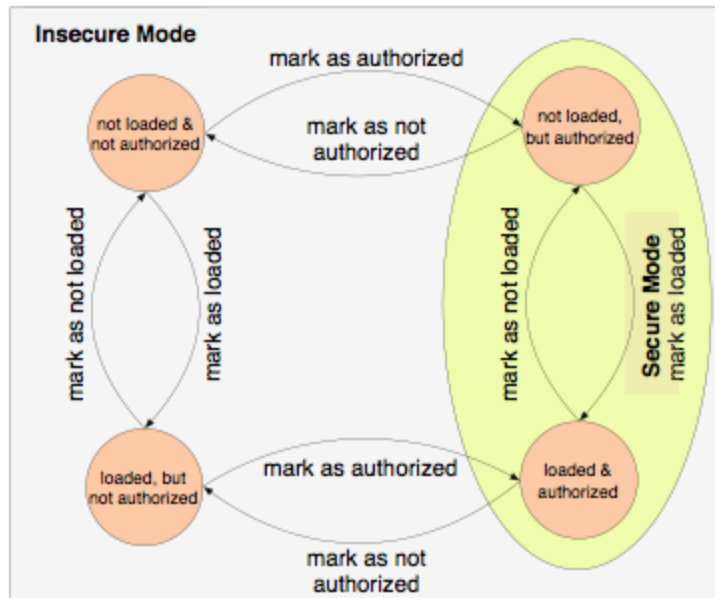


Figure 5: Insecure and secure mode with module loading and unloading in System S' design [SCHM11].

The actual loading process follows the flowchart as shown in figure 6.

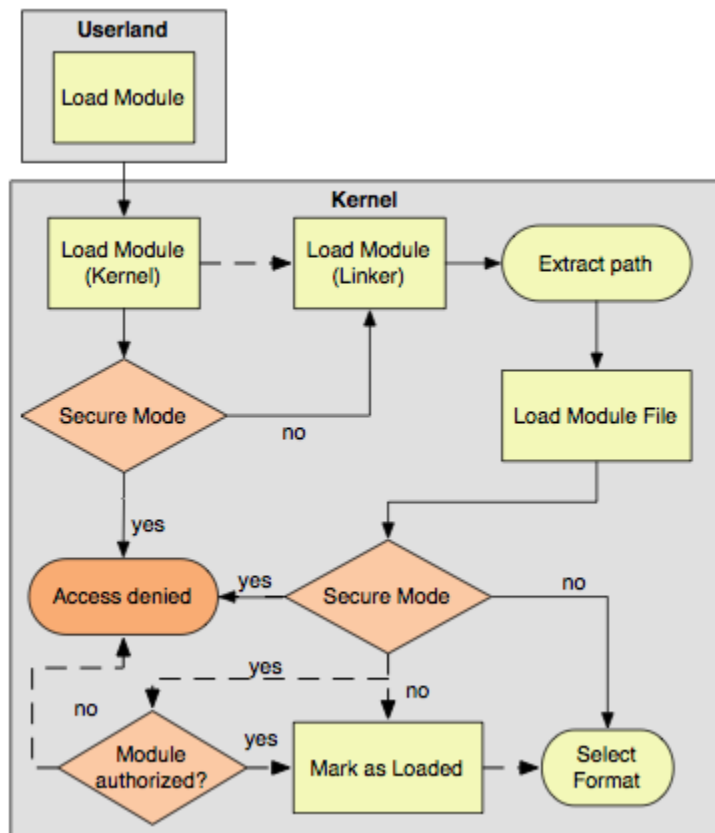


Figure 6: The module loading process in Schmidt, et al's design [13].

Our framework does not explicitly address the rootkit prevention, but that is something that could potentially be added as a separate module, or modification to an existing module, at a later time. System S is not modular nor open-source, although it does provide for a method of analysis that sits outside the virtual machine (in order to mitigate the problem of root privilege malware). Our work attempts to use the general concept of this framework (the design of a proxy, which is analogous to the various controllers in our work, to control multiple engines) while allowing it to easily be expanded upon at a later time due to modularity.

## 4 Design and Implementation

In this section, we will provide explanations of both the framework and the test implementation at various levels of depth. We begin with the framework itself in section 4.1 and continue with the implementation details in 4.2.

### 4.1 The Framework

As we've already stated numerous times, our framework is designed to be modular, cloud-based, and utilize multiple analysis methods. These key attributes are the fundamental underpinnings that make our framework unique. The nature of the cloud and cloud technologies helps mitigate the high computational load of the analysis process, as well as allowing for a greater level of concurrency. The modular design allows for greater flexibility; modules can be added or removed as needed. Finally, the fact that multiple tools are used allows (theoretically) for greater accuracy and reliability of the system.

This is a *private* cloud-based system, meaning that the systems and networks must be locally maintained within the governing organization. This decision was made for two reasons: firstly, the EULA agreements of public cloud services generally do not allow for the (intentional) execution of malicious code, and secondly, any system worth going through the effort of setting up must support Windows as VM guests and therefore must support HVM technologies. This former issue could potentially be worked out with licensing agreements, but the latter is a more technical issue; it could be circumvented, but would require all new custom logic to interface with the API of the particular cloud vendor for creating, deleting, starting, and stopping VMs.

The kernel OS and hypervisor for our framework is the XEN Cloud Platform. XEN must have direct or emulated access to Intel-VT or AMD-V technology in order to run Windows clients. Assuming that virtualization technology is present in the system processor, XEN works on the base hardware (that's how it's meant to be run) or within another virtualization product that will emulate the native virtualization technology to its guest. This becomes an issue with running XEN within a public cloud environment, as most do not appear to emulate this processor feature from our observations.

#### 4.1.1 High-Level Overview

At the most abstract level, this system simply coordinates the execution of predefined tasks in a pipeline or concurrent fashion, depending on configuration, available resources, and category of the task. We've broken down tasks into 5 categories that cover the range of the analysis process. These are, in order:

1. Submission stage
2. Static analysis stage
3. Dynamic analysis stage
4. Post-analysis and pre-classification stage
5. Classification stage

The submission stage includes the method of actually submitting the sample, as well as processing any additional custom runtime arguments if applicable. This stage could be as simple as a command line interface, a basic web page, or a full-blown web services API.

Static analysis tasks include simple things such as checking a database to see if the file has already been analyzed, check the sample using traditional signature-based antivirus tools, using some sort of disassembler (if automatable), or any other analysis task that doesn't involve executing the sample. If the file must be executed, which is generally the case, then dynamic analysis is used. Dynamic analysis tasks usually involve using automated tools to execute the sample in a controlled environment to monitor several different types of behavior. This category is generally considered to be the most dangerous; in fact, all the other categories are basically safe.

The post analysis and pre-classification stage is where any sort of log merging, logical processing, or otherwise administrative type jobs are done before the result of the prior stages is sent to the final classification stage. Classification is the final stage in the process. The end result of this stage, if implemented properly, should yield a "yes or no" answer to the question "is this sample malicious?" The accuracy of this answer will be heavily influenced by the quality of the prior analyses, as well as how well the classification model is constructed.

#### 4.1.2 Low-Level Overview

The following figure reiterates what was explained above, but may provide additional clarity.

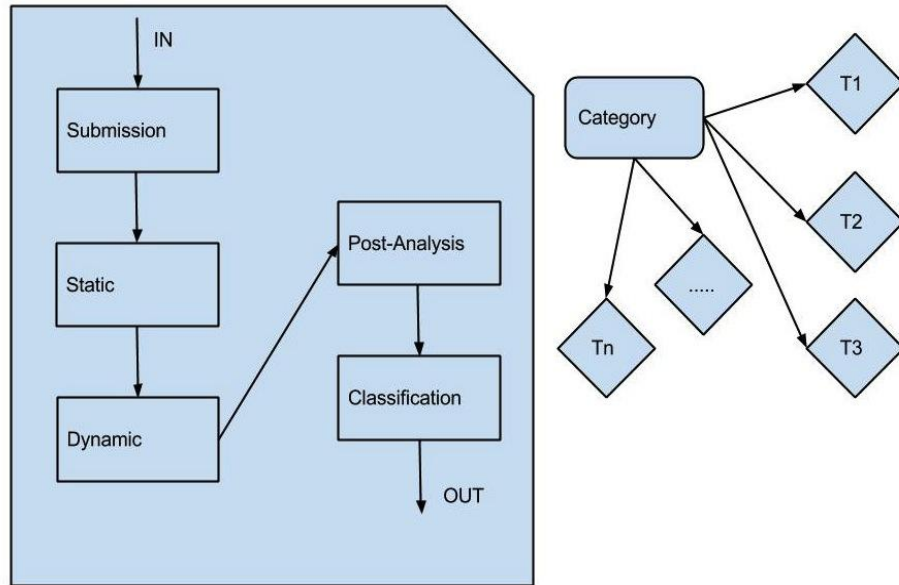


Figure 7: The general analysis flow (left) and the intra-category flow (right).

Some of these steps may be skipped, depending on arguments, if those are implemented. Within any particular category, as long as the input to a tool does not rely on the output of another tool, they may be processed concurrently as well if resources are available.

#### 4.1.3 Framework Global Variables

In order to allow the framework to run, it must know how to reach the machines and certain information about each one; as such, there are a small number of configuration files that contain information such as the IP addresses of each physical machine, each virtual machine, the names/labels of each machine, how much memory is required to run each machine (so it can compute where to launch machines if able), and other information that is necessary.

## 4.2 Implementation Overview

In order to test our framework, we need to have a bare minimum of one module to fit into each category, although modules may cover more than one category. In order to maintain simplicity, we have *only* implemented the following modules:

We break the processing into two general scenarios, training a classification model and general use. During the first stage (training), we have the following steps.

1. Preparation step 1
  - a. Update the database table that holds known decision data about a test sample set. This means to, for each sample in a clean set, update a database table with the sample's hash value and a value of False for maliciousness and for each sample in a malicious set, update the database table with the same information but True for maliciousness.

2. Preparation step 2
  - a. Analyze the entire set (clean and malicious) using Cuckoo (or whatever tools are to be used) and upload the predetermined analysis results to another database table.
3. Model training
  - a. Merge the data from the two preparation steps and format the data to match what the classification engine is expecting. For our implementation, we are using the DecisionTree libraries at [DECI13], so the data is formatted for that library.
  - b. Create the decision tree model and store the data to a storage file to avoid needing to re-compute the model for each use.

The steps are broken apart for the training because changes must be made to the configuration files between each of them. The time required to build the model is not overly important, however, since the model does not need to be regenerated except occasionally. The general operation time is much more important.

During use, the system can be left to continually run unless changes must be made (in which case, it must be restarted to trigger the changed code and settings). This allows one process that analyzes the samples to create the data necessary for classification and feeds that data straight into the classification model built in the training steps. Our current implementation of the general operation stages is as follows (for each sample queued by the system):

1. Analyze the sample using the same Cuckoo modules used during the training stages to gather the necessary data for classification.
2. Upload the data to the same table that was used during the pre-training step 2.
3. Retrieve the data from the database table from the last step and format it into the expected form for the decision tree model.
4. Classify the data using the model.
5. Finally, update the results table and return the results to the user.

Currently, the system must be stopped and configuration changes made to transition from training to non-training mode. While this implementation is simple, it adequately fills in each step in the whole process to allow a sample to go from submission to answer.

#### 4.2.1 High-Level Overview

Due to the small scale of the test lab, there are certain problems that must be worked around; for instance, in a dedicated private cloud, there would usually be some subset of the physical machines whose sole purpose was to hold vast amounts of storage space as a network attached storage (NAS). We need at least two *processing* machines to evaluate the performance metric, therefore we would need *three* physical machines to allow one to become the NAS.

We have a total of four physical machines in our cloud environment. Three of the machines are used for processing, while one of them has been transformed into the NAS supporting the cloud. The primary virtual machine that runs the analysis framework (referred to as the control vm) is a “floating” machine,

meaning that it is stored on the NAS and loaded over the network. This is important because it allows the system to be restored (restarted) easily in the event of the physical host failure.

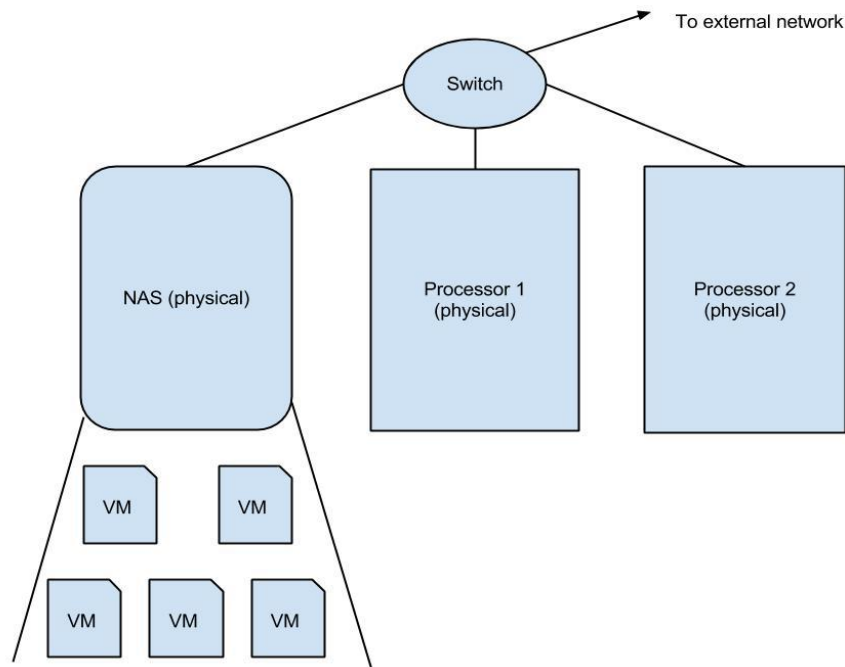


Figure 8: The *ideal* “two” machine cloud.

In the above figure, we show several virtual machines stored on the NAS and loaded over the network. During evaluation, we found that our small network could not handle loading more than a couple vms over the network at a time. In order to overcome this, each physical host holds two instances of a Cuckoo virtual machine. The NAS is still needed to allow for live migration of virtual machines, to store backup templates (so vms can be quickly recreated in the event of hardware failure), and to allow the control vm to be agile.

#### 4.2.2 Mid-Level Overview

At the high level, we have the physical machines hosting a NAS device, which is in turn holding the virtual hard drive files for all the other virtual machine templates and the control vm. At the mid level, we can see that there is one VM that coordinates the entire cloud. This “control VM” is floating, so it can be moved around as needed, but (just like the rest of the machines, virtual or physical) *must* have a static IP address; this is so that it can be reached by the network IP on any host, rather than relying on knowing the host.



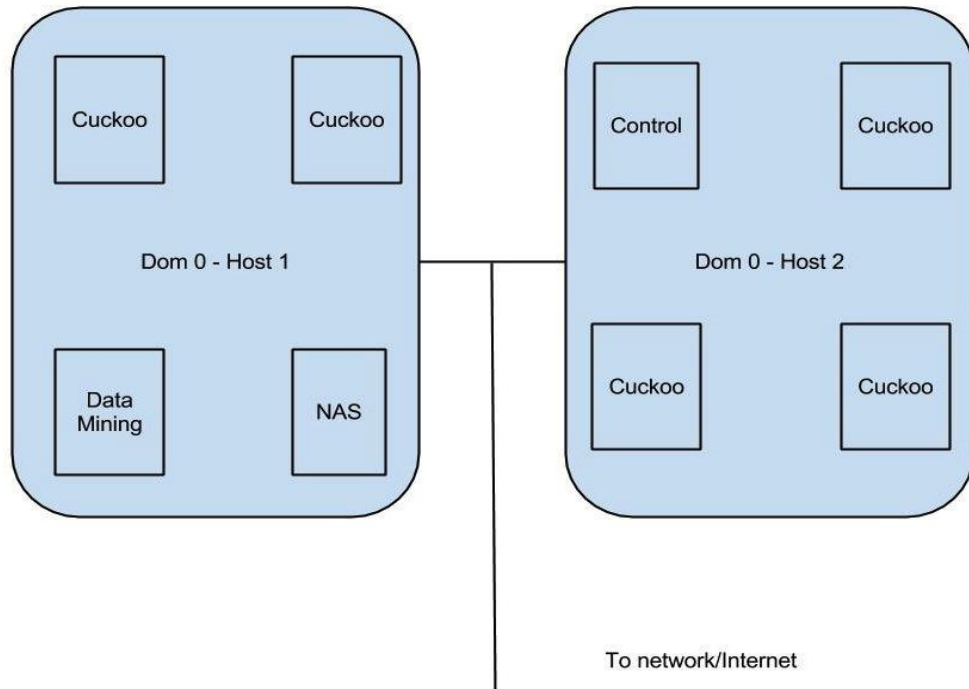


Figure 9: A suggested VM layout based on suggested VM RAM settings and host RAM.

In figure 10 above, we show a suggested VM allocation, although the actual structure will vary as machines are started and stopped by the controller. The NAS and the Control vm will be constantly running, while the Cuckoo vms will only be running during analysis. Each physical host can support up to 2 instances of Cuckoo and the control vm at the same time. More powerful machines (more RAM) could support more instances.

#### 4.2.3 Low-Level Overview

The OS that is actually running on the host machines (known as DOM 0 in XEN terminology) is CentOS and is running the Xen API (or XAPI). The physical machines are organized into a resource pool, and one of them (host 1) is assigned as the pool master. The control VM is an unprivileged guest VM (known as DOM U) so it must “remotely” login to the current pool master and send commands via the XAPI to cause machines to shutdown, startup, migrate machines, or perform other tasks.

The tunnel used for remote communication is secured through SSL, but the control VM still must have access to the username and password. Just for the purposes of simplicity, the security of this information will not be strictly enforced; the information will be stored in one of the global configuration files. In a non-test implementation, additional security measures will need to be taken to protect the login credentials.

In order to store results from prior analyses, the control VM will also be hosting a small MySQL database. The total job duties of the control VM are as follows:

1. Maintain connection to pool host.
2. Attempt to efficiently spread out the analysis tasks across hosts.
3. Maintain internal database for record retention.
4. Queue waiting samples and process them in order.

Figure 10 below shows a more concrete data flow, taking into account the specific tools.

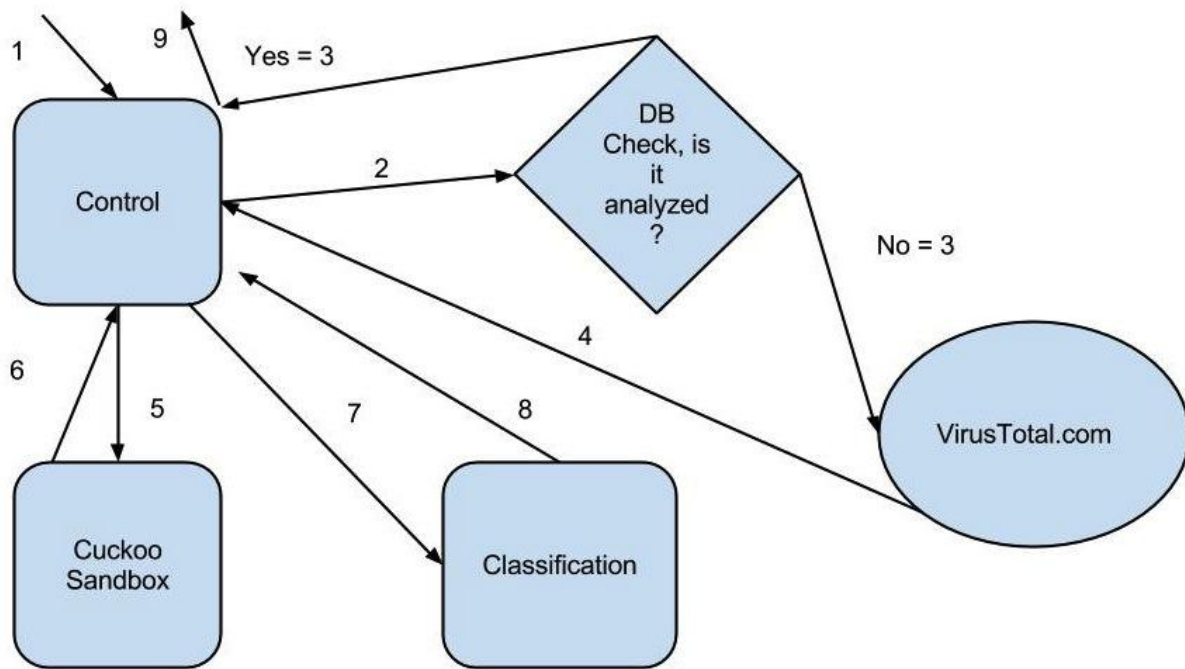


Figure 10: More specific data flow.

The pseudo-code looks like this:

1. Control receives submission
2. Spin up and submit sample to prepped Cuckoo VM.
3. Get response back from Cuckoo.
4. Submit results from 3 to the classification engine
5. Get results back from classification.
6. Store results in database and then return them to the user.

These steps are for a single submission. The number of concurrent submissions that can be handled will vary based on resource configuration, timing, and availability. The largest bottleneck in the process is the analysis done by the Cuckoo Sandbox as that can take anywhere from a couple minutes to much longer (depending on configuration and timeout setting); the control VM, the NAS can all handle several samples within the time of one Cuckoo analysis, therefore the number of concurrent samples that can be handled is essentially the number of concurrent instances of Cuckoo that can be running. On the physical

hardware and configuration used in our lab implementation, that means we will have a maximum of 2 concurrent samples per physical processing host. Due to this restriction, we enforce the limit of two analysis processes per machine during testing and operation; failure to enforce this limit causes too many samples to be waiting for a Cuckoo vm to free and has the negative effect of samples timing out while waiting.

## 5 Evaluation

In order to evaluate our implementation of the framework, we have used a small collection of malicious samples (malware) and a small collection of non-malicious samples. Combined, all the samples will be referred to as the test base. The malicious samples are from the repositories hosted at [CONT13]. The non-malicious samples are several small (less than 10KB) executable files gathered from a clean Windows 7 machine.

Specifically, our test base consists of 76 malicious files and 24 clean files. We only process the malicious and clean separately during the initial preparation step for building the model. During all other testing, the 100 samples are lumped together.

There are three metrics of interest: performance, flexibility, and reliability. Each metric is influenced by a different attribute of the project. We will cover each of these in turn in sections 5.1, 5.2, and 5.3 respectively.

### 5.1 Performance Evaluation

Our definition of higher performance means a shorter time requirement to analyze the same test base. The primary influencing attribute for this metric is the fact that the framework is designed to take advantage of the cloud and utilize parallelism. The major performance boost is from the higher level of concurrency available as the number of machines grow.

Our test cloud environment has 3 processing machines. Since each machine can only support 2 instances of the Cuckoo vms, we will test using our sample test base and using 1 machine (2 processes), 2 machines (4 processes), and 3 machines (6 processes).

During evaluation, we measure the following times:

1. The time required for the training preparation step 1 with both the clean and malicious sample sets.
2. The time required for the training preparation step 2 using the entire test base and 1 machine, 2 machines, and 3 machines.
3. The time required to generate the classification model.
4. Finally, the time required to analyze the entire test base during general operation mode using 1 machine, 2 machines, and 3 machines.

The reason why only one time measure is taken during number 1 and 3 above is because those steps are not distributed and therefore do not gain any advantage (and are not designed to) from using multiple machines. The other two steps both utilize Cuckoo for analysis and the Cuckoo vms are distributed, therefore the number of machines does impact the overall time.

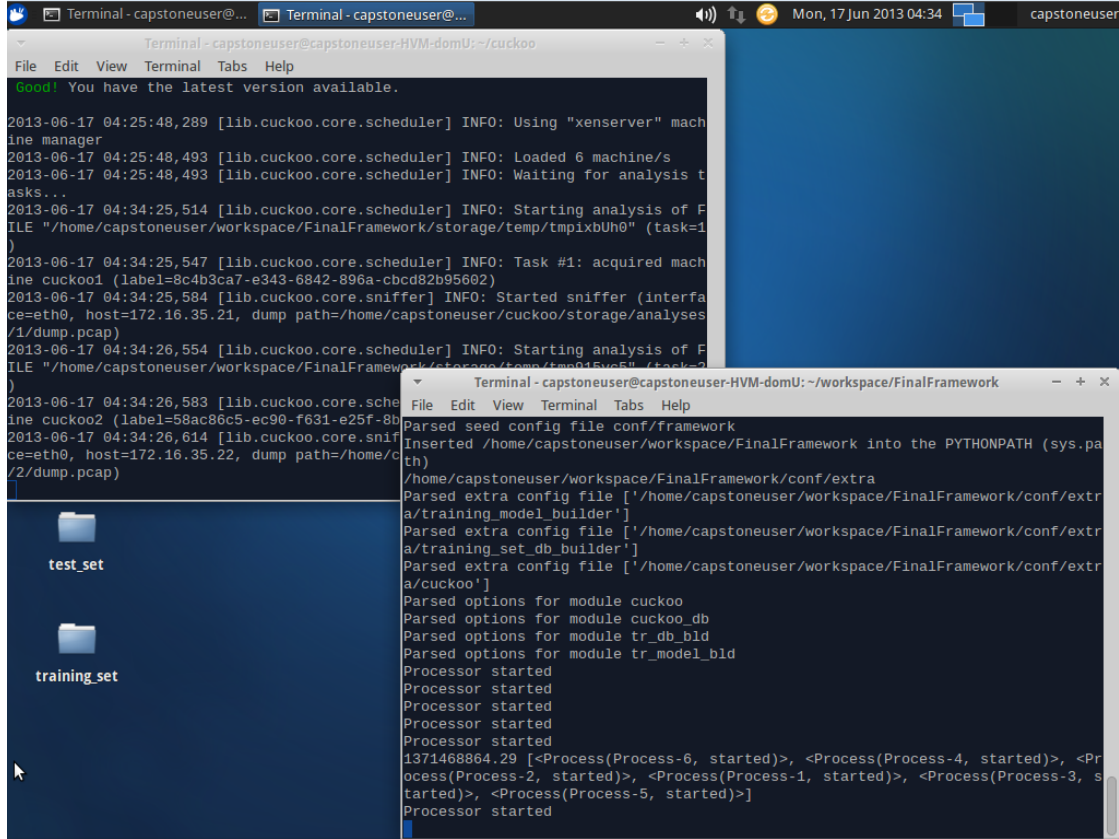


Figure 11: The framework in operation processing the clean samples from the training set with 3 machines.

Figure 11 above shows what the system looks like during operation. Simple information messages are displayed to the user that gives some since of how many analysis processes (one per sample) have been started and are currently running. That is shown in the lower right window, the upper left window if the console output from the Cuckoo tool and shows basic information as well.

Table 1: Total time (in seconds) to process the sample sets for the training model preparation step 1.

Number of cloud machines (down) / size of test base (right)	76 malicious (training)	24 clean (training)	Combined (not counting transition time)
N/A for this test	115.6	43.5	159.1

This shows that the time required for the initial preparation step before building the model is actually quite short. Preparing the table in this step does not use any distributed/cloud technology, although it actually could and would not be that difficult to implement.

Table 2: Total time (in seconds) to process the test base for the training model preparation step 2 using 1, 2, and 3 machines.

Number of cloud machines (down) / size of test base (right)	Full test base (100 samples)
1 machine	7557.0
2 machines	4179.9
3 machines	3090.4

Here is a good example of why the cloud helps. It allows for multiple machines to each be simultaneously processing an individual sample per process. In our environment, our physical hardware is not overly powerful, so it can only support 2 processes per machine. This means that with one machine, 2 samples are processed at a time, with two machines, 4 are processed at a time, and with three machines, 6 are processed at a time.

Table 3: The total time (in seconds) required to build the classification model using the data generated in the training model preparation steps 1 and 2.

Time to build the classification model	25.1
--	------

As can be seen from the data, running our framework within a distributed environment (cloud) yields great improvements. During the training model preparation step 2, the test base processed in about 126 minutes on one machine, while it only took about 70 minutes on two, and about 51 minutes with 3. We did notice a slightly higher amount of timed out Cuckoo analyses during the test with 3 machines, although steps could be taken to help prevent this without much performance hit.

Table 4: The total time required for processing the test base using the general operation mode.

Number of cloud machines (down) / size of test base (right)	Full test base (100 samples)
1 machine	11410
2 machines	6266
3 machines	6480

We obtain similar results in the general operation mode as in the training stages. An area for future research would be overall optimization; for instance, each analysis process loads in its own copy of the classification model currently. The model is loaded in from the file system as I/O activity and is fairly decent in size, causing the loading of the model to take more time than using the model. Since it is used in a read-only fashion, there could be one (thread-safe and multi-process-safe) instantiation that is shared amongst processes that could be loaded only once.

## 5.2 Flexibility Evaluation

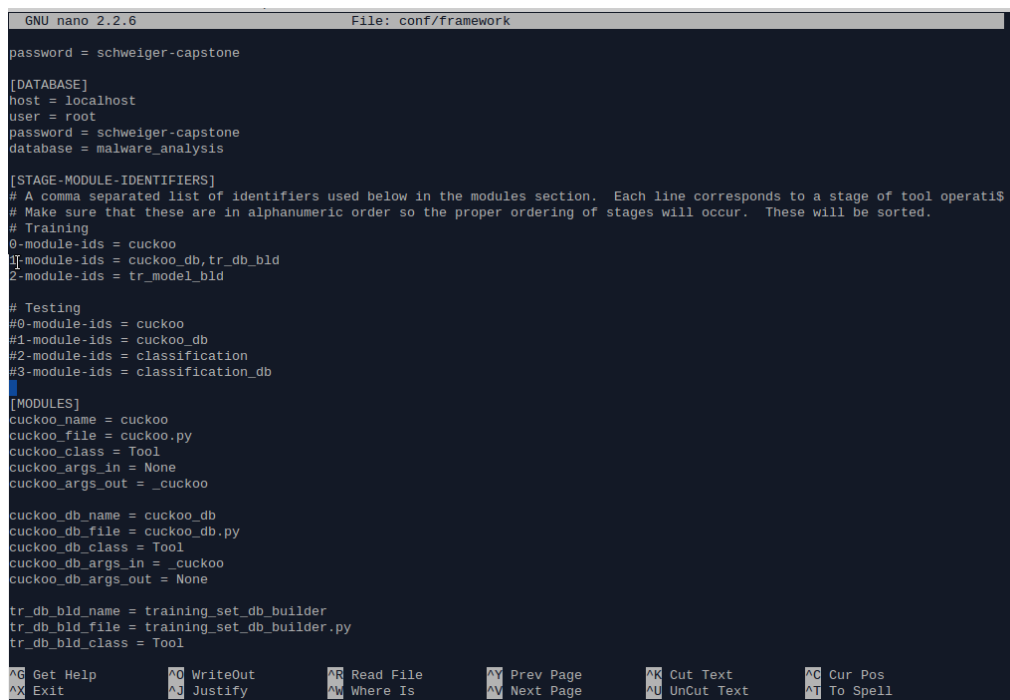
We define greater flexibility as the ease with which additional functionality can be added, or existing functionality can be modified. The flexibility of this system comes from the fact that all of the important virtual machines are floating and settings can be changed easily. Adding and removing tools can take a little work, depending on the tool, although the work involved with adding or removing a tool should not be any more difficult than installing, configuring, and using that tool without the framework. This metric is difficult to measure with hard figures, especially since it will vary greatly depending on what is being added or removed.

The ultimate goal of this metric is to ease the work on the users to integrate tools into the overall process. The general process of putting tools together into one seamless process involves three distinct steps in our framework:

1. Configure the tool to work independently.
2. Add an entry to the framework's main configuration file.
3. Program the module to determine how to use the tool.

Step one is out of the scope of this work, but would be required using any sort of integration system. If the tool isn't installed and configured, it can't be used by the analysis process. The only real requirement that the installed tool must fulfill to be used by our framework is that it must not require direct user interaction during execution (mouse clicks for example) so that it can be automated.

Step two involves adding and/or modifying approximately 6 lines in the framework's main configuration file. The most difficult part of this step is simply deciding at what point in the process the tool should run and what inputs/outputs should be provided to the programmed module from step 3.



```
GNU nano 2.2.6 File: conf/framework

password = schweiger-capstone

[DATABASE]
host = localhost
user = root
password = schweiger-capstone
database = malware_analysis

[STAGE-MODULE-IDENTIFIERS]
# A comma separated list of identifiers used below in the modules section. Each line corresponds to a stage of tool operati$
# Make sure that these are in alphanumeric order so the proper ordering of stages will occur. These will be sorted.
# Training
0-module-ids = cuckoo
1-module-ids = cuckoo_db,tr_db_bld
2-module-ids = tr_model_bld

# Testing
#0-module-ids = cuckoo
#1-module-ids = cuckoo_db
#2-module-ids = classification
#3-module-ids = classification_db

[MODULES]
cuckoo_name = cuckoo
cuckoo_file = cuckoo.py
cuckoo_class = Tool
cuckoo_args_in = None
cuckoo_args_out = _cuckoo

cuckoo_db_name = cuckoo_db
cuckoo_db_file = cuckoo_db.py
cuckoo_db_class = Tool
cuckoo_db_args_in = _cuckoo
cuckoo_db_args_out = None

tr_db_bld_name = training_set_db_builder
tr_db_bld_file = training_set_db_builder.py
tr_db_bld_class = Tool

^G Get Help      ^O WriteOut     ^R Read File    ^V Prev Page    ^K Cut Text     ^C Cur Pos
^X Exit          ^J Justify      ^W Where Is     ^N Next Page    ^U UnCut Text   ^T To Spell
```

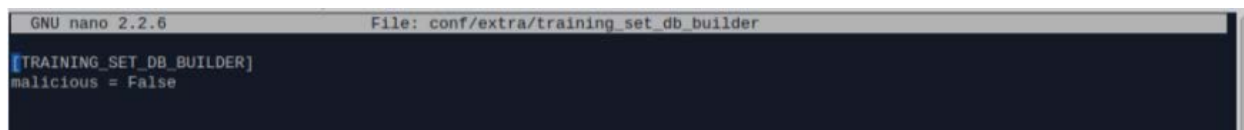
Figure 12: Screenshot of the framework configuration file.

Figure 12 above shows a portion of the current framework configuration file. In order to add a new module, simply decide what to identify the module as, add that to one of the lines under the section titled [STAGE-MODULE-IDENTIFIERS], and add the related entries in the [MODULES] section. All modules that are listed by the same stage number will be run concurrently as threads in the process. The additional lines to add to the [MODULES] section are as follows:

- <id>\_name = name of module
- <id>\_file = filename of the python file.
- <id>\_class = the actual class name of the object in the python file.
- <id>\_args\_in = The arguments to be provided to the module. The path to the sample and a copy of the configuration parser objects are always provided.
- <id>\_args\_out = The arguments to be expected from the module.

Since the file path to the sample and a copy of the configuration parser are always provided to the module, the acceptable options for args\_in and args\_out are None and any variable that begins with an underscore. The variables are converted into temporary files and the paths to those files are provided to the modules that use them. Variables that are named identically are mapped to the same temporary file. For instance, the cuckoo module (see figure 12) has an args\_out variable of “\_cuckoo” and the cuckoo\_db module has an args\_in variable of “\_cuckoo”. This matching causes the path to the *same* temporary file to be provided to both modules (as self.args\_in[2] or self.args\_out[0], respectively. Elements 0 and 1 of self.args\_in are the sample’s file path and the config parser object). The fact that both modules are given the path to the same file allows for writes from one module to be read by another (these operations are not thread safe though, so shared variables much come in different stages).

An optional step that may be taken if needed at this point is to create a new configuration file in the conf/extras directory. At the initial loading of the framework, the framework configuration file is parsed, followed by all files (that can be parsed) in the extras folder. If a new file is created, place all options in a section with a unique title, we recommend the id or name of the tool. This step is only needed if any custom options are needed to be read from the config parser object given to the module.



```
GNU nano 2.2.6 File: conf/extra/training_set_db_builder
[TRAINING_SET_DB_BUILDER]
malicious = False
```

Figure 13: A custom configuration options file used by the training\_set\_db\_builder module.

Finally, step three is to actually program a python module to determine the behavior of the tool and its interactions with the rest of the process. This can be as simple as calling a specific command (command line interface) or as complex as integrating custom logic, parsing input files, and uploading/retrieving data from a database. While this step can be quite a bit of work, the framework does simplify it by providing the python module direct access to the path to the sample under analysis, a copy of the parser object, and any additional input/output arguments configured as already explained.

```

File Edit View Terminal Tabs Help
GNU nano 2.2.6 File: modules/training_set_db_builder.py
...
Created on Jun 6, 2013
@author: capstoneuser
...

from AbstractTool import AbstractTool
from internals.connection import MySQL_Connection as mysql
import hashlib
import internals.utils

class Tool(AbstractTool):
    ...
    Simply uploads the hash from the source file along with whether the file is malicious
    or not (based on config file); this is to be used only when building the database to
    use for training a model.
    ...

    def override(self):
        parser = self.args_in[1]
        sample = self.args_in[0]
        db = mysql()

        #m = hashlib.sha256()
        #with open(sample, 'rb') as f:
        #    m.update(f.read())
        #hash_val = m.hexdigest()

        hash_val = internals.utils.get_sha256(sample)

        malicious = parser.getboolean('TRAINING_SET_DB_BUILDER','malicious')
        sql = "insert into 'malicious_answer' ('sha256','malicious') values ('"
        sql = sql + hash_val + "','" + str(malicious) + "');"
        db.execute(sql)

Get Help WriteOut Read File Read 36 lines Cut Text Cur Pos
Exit Justify Where Is Prev Page Next Page UnCut Text To Spell

```

Figure 14: The code for the module to build part of the training set information in the database.

As can be seen from figure 13, the code does not have to be long and complex. The framework provides an Object-Oriented structure and some utilities to help simplify needed tasks as well. For instance, the internal connection python module has a class to handle access to the configured database and a class to handle connections to the XenServer pool master. In the created tool module, the code must extend AbstractTool (from the AbstractTool python file in the modules folder) and override the “override(self)” function. No other function should be overwritten.

```

GNU nano 2.2.6 File: modules/AbstractTool.py
...
Sets up the internal fields that are to be used by extending classes. These are as follows:
args_in = Tuple of arguments for reading purposes only (this is not enforced, but abuse will cause problems).
Index 0 is always the file path to the sample itself (use open with 'rb' mode).
Index 1 is always the parser object (to be read from if needed).
Indices 2+ are the in-parameters from the configuration and will always be file paths if present.
args_out = Tuple of arguments for writing (or reading) purposes. This tuple could be empty if no output files
are configured. Any present elements will always be file paths (use open with 'wb' mode)
...

try:
    self._module_name = name
    self._semaphore = semaphore
    self.args_in = argument_dict.get('in')
    self.args_out = argument_dict.get('out')
except:
    # Explicitly re-raise exception
    raise

def run(self):
    ...
    Acquires the semaphore, launches override, then releases the semaphore.
    This is why you should not touch the semaphore in extending classes.
    ...
    try:
        self._semaphore.acquire()
        self.override()
        self._semaphore.release()
    except:
        import sys
        print sys.exc_info()

def override(self):
    raise NotImplementedError

```

Figure 15: An excerpt from the AbstractTool class.



Figure 14 shows the general structure of the AbstractTool class. The top code is part of the `__init__(...)` function (the signature was off screen). As can be seen, the framework uses semaphores to handle the threads within a stage and to prevent the analysis from continuing to the next stage until all current stage tools have returned. In order to prevent stalling the process (and the whole system if enough processes are stalled), the programmed module should always ensure that it completes or times out).

Our evaluation of this metric is that it at least compares to, if not improves, the required work that any other approach for integrating completely separate tools into the same process. We feel that ours provides the structure and tools to simplify the process of adding new tools or modifying existing tools (at least the tools' interactions with the rest of the process, not the tool itself).

### 5.3 *Reliability*

Our evaluation of reliability is based solely on our implementation of a decision tree based classification model that is very naïve. The decision tree is build using only the information returned from VirusTotal via Cuckoo. This is meant to serve as a proof-of-concept implementation of the framework. The test base for this metric is the same 100 sample test base used for the performance test, but with additional clean and malicious files that were not used for training. The total size of this test base is 147 files.

The decision tree library for the classification was retrieved from [DECI13]. We are using it along with information pulled from the database to build the model. In order to evaluate this metric, there are three basic steps:

1. Build the model
2. Test the model
3. Check the results

Step one is done by fetching all rows from the ``virustotal`` and the ``malicious_answer`` database tables, removing irrelevant columns (such as the file's hash), formatting it, writing the formatted results out to a temporary file, and passing the file to the decision tree library to build the model. This is all done as part of the training steps (see the performance metric evaluation).

Step two consists of retrieving the VirusTotal results from the database based on the sample's sha256 hash value. In the event that the VirusTotal results are not available, a result of non-malicious is returned. While this may not be accurate, a more sophisticated classification model could correct this (this is one area of future research). Based on the VirusTotal results, the decision tree model is traversed until a leaf is reached. The value of the leaf node is returned. The results are uploaded into a third database table for comparison to the known data.

Finally, step three consists of reviewing the tables and comparing the returned result from step two with the expected value. We are using a known test set, so the correct result is known.

The following table shows the evaluation of the module’s reliability as an accuracy matrix of the true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Just as a brief description, the following definition are used:

- True positive: A malicious sample is classified as malicious.
- True negative: A non-malicious sample is classified as non-malicious.
- False positive: A non-malicious sample is classified as malicious.
- False negative: A malicious sample is classified as non-malicious.

The “good” results are the “true” results, while the “false” results are bad. Note that the sum of all four categories and any samples that could not be classified must add up to the total number of samples (147).

Table 5: The results of the reliability testing using all 3 machines.

True malicious value (down), classified malicious value (right)	Malicious	Non-Malicious
Malicious	80 (TP)	2 (FN)
Non-Malicious	1 (FP)	20 (TN) + 44 unclassified

Our implementation defaults to assuming a sample is clean if it cannot be classified using the model or because the proper input data is not present. While this may not be the best approach, we have used it for reasons of simplicity. A more complex, more sophisticated model could be used to achieve higher accuracy, as this was simply to show a proof of concept.

$$Accuracy\ using\ default\ metric = \frac{TP + TN}{Total\ Sample\ Size} = \frac{80 + 20 + 44}{147} = \frac{144}{147} = 0.98$$

$$Accuracy\ ignoring\ failed\ samples = \frac{80 + 20}{147} = \frac{100}{147} = 0.68$$

The accuracy of the model should grow as larger sample bases are used to regenerate the model over time. In addition, we will be investigating other means of classification (such as using support vector machines) and utilizing additional behavioral features from Cuckoo to generate more sophisticated classification models in future research.

## 6 Conclusion and Future Work

We began this project with one primary goal: to create a modular, extensible framework to help create a malware analysis lab in a cloud environment. We wanted to help answer the question of “how can we overcome the shortcomings of current cloud-based analysis frameworks in order to provide increased performance, accuracy, and flexibility?” We believe we have answered that question with our framework and our implemented system.

Our research showed that the need to effectively and efficiently analyze malware was, and still is, needed. Due to this need, there are many researchers, companies, and public institutions across the planet all working on trying to further the science of malware analysis. Our framework provides a starting place for small research organizations to build a specialized analysis environment in-house to use for testing ideas and theories. It's written in Python, and currently utilizes the Xen Cloud Platform as the underlying operating system on the physical hosts, although it is flexible enough that it could easily be moved to a different virtualization platform.

Each part of the malware analysis process can be identified as a discrete component and our framework relies upon this fact. Functionality of the system and the integration of existing external tools depend on the use of custom modules, although our implementation provides a base system on which to build. We have created a sample implementation that includes basic modules for each step of the analysis process, including traditional anti-virus checks, dynamic analysis, tool output aggregation, and classification. The Cuckoo module provides for both anti-virus checking and dynamic analysis, the decision tree classification model allows for the end-result classification, and the modules the tie together Cuckoo, the classification engine, and the database together serve to aggregate the data.

The beauty of our design, as well, is that each of these modules can be expanded, disabled, or completely replaced. We have shown that utilizing distributed resources through cloud computing provides a boost to the overall performance of our system, and have advocated for the benefits of using the system rather than building another solution from scratch. While it is a fairly rudimentary system, our solution allows for a modular design and is capable of growing to match additional requirements at a later time. There is always room for improvement, but the fundamental goals were met.

Our work is distinguished from other efforts in this field due to the following attributes:

- Our work provides a holistic system that is based on previous efforts using dynamic analysis tools with modules for integrating the results into various data classification models to produce a final answer (malicious or benign).
- Our work can be extended with additional modules for added functionality, such as adding a new tool that is not included in the example implementation.
- Our work is completely open-source and eases the integration of other tools into the whole process.

## **6.1 Future Research**

Due to the reasons provided in the previous section, we have deemed our work as successful, but there is always more work to do. Our classification engine was fairly naïve (it basically just used the results from traditional AV solutions via VirusTotal and Cuckoo), the framework is complete enough to allow modifications and more sophisticated models to be used. Using this project as a starting base, we would like to pursue several avenues of related research.

We would like our future research to be able to answer the following questions:

- How can we optimize the existing framework to allow for greater productivity on fewer resources with better accuracy?

- How can we utilize the output from Cuckoo for a more diverse feature set with which to build a more sophisticated classification model?
- Can we achieve better results using a different virtualization platform or a different programming language?
- Can we make the process of modifying, adding, or removing tools from the processing pipeline even easier for the user?

Each of these questions is related, but serves as a slightly different research approach. We would like to focus on improving the current system and framework before we branch out and start comparing alternate technologies and other tools. Our current classification model most certainly needs work in order to be more effective. The fact that 44 samples out of the 147 sample test base were not able to be classified is something that needs to be addressed

## 6.2 Acknowledgements

We like to acknowledge and thank VirusTotal for graciously providing us with a private API key for use during our project. While we could have used a public key, this could have drastically altered our timing results, as the public key only allowed for 4 requests per minute [VIRU13].

## 7 References

- [ADWA12] (2012) adware. [Online]. Available: <http://dictionary.reference.com/browse/adware>
- [CHA11] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Anderson, "Splitscreen: Enabling efficient, distributed malware detection," *Journal of Communications and Networks*, vol. 13, no. 2, pp. 187-200, April 2011.
- [CONT13] (2013) Malware Dump. [Online]. Contagio. Available: <http://contagiodump.blogspot.com>
- [CUCK12] (2012) Cuckoo. [Online]. Available: <http://www.cuckoosandbox.org/>
- [DECI13] (2013, May) A. Kak, Decision Tree. [Online]. Available: <https://engineering.purdue.edu/kak/distDT/DecisionTree-1.7.1.html>
- [EGEL08] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1-6:42, Mar. 2008, [Online]. Available: <http://doi.acm.org/10.1145/2089125.2089126>
- [HAN12] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed. Elsevier Inc., 2012.
- [HUAN11] H.-D. Huang, C.-S. Lee, H.-Y. Kao, Y.-L. Tsai, and J.-G. Chang, "Malware behavioral analysis system: Twman," in *Intelligent Agent (IA), 2011 IEEE Symposium on*, april 2011, pp. 1–8.
- [IDIK07] N. Idika and A. P. Mathur, "A survey of malware detection techniques," February 2007.
- [MAAN12] P. Maandag, H. Barendregt, and A. Silva, "Solving 3-SAT," Bachelor Thesis, Radboud University Nijmegen, [Online], 2012, Available:

- [http://www.cs.ru.nl/bachelorscripties/2012/Peter\\_Maandag\\_3047121\\_\\_\\_\\_Solving\\_3-Sat.pdf](http://www.cs.ru.nl/bachelorscripties/2012/Peter_Maandag_3047121____Solving_3-Sat.pdf)
- [MALW12] (2012, October) Malware. [Online]. Available: <http://en.wikipedia.org/wiki/Malware>
- [MART09] L. Martignoni, R. Paleari, and D. Bruschi, "A framework for behavior-based malware analysis in the cloud," 2009.
- [MICR07] Microsoft, "Understanding anti malware technologies," Microsoft, Tech. Rep., 2007.
- [MO07-1] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP '07. IEEE Symposium on*, may 2007, pp. 231 –245.
- [MO07-2] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, dec. 2007, pp. 421 –430.
- [NSAC11] N. S. A. C. for Assured Software, "On analyzing static analysis tools," July 2011, [cas@nsa.gov](mailto:cas@nsa.gov).
- [NEW12] (2012) New mac os x backdoor being used for an advanced persistent threat campaign. [Online]. Available: [http://www.kaspersky.com/about/news/virus/2012/New\\_Mac\\_OS\\_X\\_Backdoor\\_Being\\_Used\\_for\\_an\\_Advanced\\_Persistent\\_Threat\\_Campaign](http://www.kaspersky.com/about/news/virus/2012/New_Mac_OS_X_Backdoor_Being_Used_for_an_Advanced_Persistent_Threat_Campaign)
- [ORDE12] (2012, October) Ordering. [Online]. Available: <http://www.hex-rays.com/products/ida/order.shtml>
- [PALE10] R. Paleari, "Dealing with next generation malware," Ph.D. dissertation, Universita degli Studi Di Milano, 2010.
- [PROJ12] (2012) Projects. [Online]. Available: <http://honeynet.org/project>
- [RANS12] (2012, March) Ransom trojans spreading beyond russian heartland. [Online]. Available: <http://news.techworld.com/security/3343528/ransom-trojans-spreading-beyond-russian-heartland/>
- [SCAR12] (2012) Scareware. [Online]. Available: <http://dictionary.reference.com/browse/scareware?s=t>
- [SCHM11] M. Schmidt, L. Baumgartner, P. Graubner, D. Bock, and B. Freisleben, "Malware detection and kernel rootkit prevention in cloud computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, feb. 2011, pp. 603 –610.
- [SUN11] M.-K. Sun, M.-J. Lin, M. Chang, C.-S. Lai, and H.-T. Lin, "Malware virtualization-resistant behavior detection," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, dec. 2011, pp. 912 –917.
- [TREN12] (2012, August) Trend micro security expert: Malware attack against vmware limited in scope. [Online]. Available: <http://www.crn.com/news/security/240006175/trend-micro-security-expert-malware-attack-against-vmware-limited-in-scope.htm>

- [VAN08] J. Van Randwyk, K. Chiang, L. Lloyd, and K. Vanderveen, "Farm: An automated malware analysis environment," in *Security Technology, 2008. ICCST 2008. 42nd Annual IEEE International Carnahan Conference on*, oct. 2008, pp. 321–325.
- [VERI12] Verizon, "2012 data breach investigations report," Tech. Rep., 2012.
- [VIRU13] (2013), Virus Total. [Online]. Available: <https://www.virustotal.com/en/>
- [WHAT08] (2008) What bots do and how they work. [Online]. Available: <http://www.honeynet.org/node/54>
- [YOU10] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, nov. 2010, pp. 297–300.
- [ZHEN10] Z. Zheng and Y. Fang, "An ais-based cloud security model," in *International Conference of Intelligent Control and Information Processing*, Dalian, China, August 2010, pp. 153–158.