BotSwindler: Tamper Resistant Injection of Believable Decoys in VM-Based Hosts for Crimeware Detection

Brian M. Bowen Pratap Prabhu Vasileios P. Kemerlis Stelios Sidiroglou-Douskos
Angelos D. Keromytis Salvatore J. Stolfo
Department of Computer Science Columbia University

January 28, 2010

Abstract

We introduce BotSwindler, a bait injection system designed to delude and detect crimeware by forcing it to reveal itself during the exploitation of monitored information. Our implementation of BotSwindler relies upon an out-of-host software agent to drive user-like interactions in a virtual machine, seeking to convince malware residing within the guest OS that it has captured legitimate credentials. To aid in the accuracy and realism of the simulations, we introduce a low overhead approach, called virtual machine verification, for verifying whether the guest OS is in one of a predefined set of states. We provide empirical evidence to show that BotSwindler can be used to induce malware into performing observable actions and demonstrate how this approach is superior to that used in other tools. We present results from a user study to illustrate the believability of the simulations and show that financial bait information can be used to effectively detect compromises through experimentation with real credential-collecting malware.

1 Introduction

The creation and rapid growth of an underground economy that trades in stolen digital credentials has spurred the growth of crimeware-driven bots that harvest sensitive data from unsuspecting users. This form of malevolent software employs a variety of techniques from web-based form grabbing and key stroke logging, to screenshots and video capture for purposes of pilfering data on remote hosts to automate financial crime [1, 2]. The targets of such malware range from individual users and small companies to the most wealthiest organizations [3]. Recent studies indicate bot infections are on the rise and up to 9% of the machines in an enterprise are now bot-infected [4].

Traditional crimeware detection techniques rely on comparing signatures of known malicious instances to identify unknown samples, or on anomaly-based detection techniques in which host behaviors are monitored for large deviations from a baseline. Unfortunately, these approaches suffer a large number of known weaknesses. Signature-based methods can be useful when a signature is known, but due to the large number of possible variants, learning and searching all possible signatures to identify unknown binaries is intractable [5]. Anomaly-based methods are susceptible to false positives and negatives, limiting their potential utility. Consequently, a large amount of existing crimeware now operate undetected by antivirus software. A recent study focused of Zeus[1] (the largest botnet with over 3.6 million PC infections in the US alone [7]), revealed that the malware bypassed up-to-date antivirus software 55% of the time [8].

[1]Zeus uses key-logging techniques to steal sensitive data such as user names, passwords, account numbers. It can be purchased on the black market for $600, complete with support and maintenance [6].

Another drawback to conventional host-based antivirus software is that it typically monitors from within the host, making it vulnerable to evasion or subversion by malware. In fact, we see an increasing number of malware attacks that disable defenses such as antivirus software prior to undertaking some malicious activity [9].

In this work, we introduce BotSwindler, a novel system designed for the proactive detection of credential stealing malware on VM-based hosts. BotSwindler relies upon an out-of-host software agent to drive user simulations that are meant to convince malware residing within the guest OS that it has captured legitimate credentials. By the nature of its out-of-host operating position, the simulator is tamper resistant and difficult to detect by malware residing within the host environment. We posit that malware that detects BotSwindler would need to analyze the behavior of its host and decide whether it is observing a human or not. In other words, the crimeware would need to solve a reverse Turing Test [10]. We assert that if attackers are forced to spend their time looking at the actions on each host it infects one by one to determine if they are real or not in order to steal information, BotSwindler would be a success; the crimeware's task does not scale. To generate simulations, BotSwindler relies on a formal language that is used to specify a simulation of human user's sequence of actions. The language provides a flexible way to generate variable simulation behaviors that appear realistic.

One of the challenges in designing an out-of-host simulator lies in the ability to detect the underlying state of the OS. That is, to verify the success or failure of mouse and keyboard events that are passed to the guest OS. For example, if the command is given to open a browser and navigate to a particular URL, the simulator must validate that the URL was successfully opened before proceeding with the next command. To aid in the accuracy and realism of the simulations, we developed a low overhead approach, called virtual machine verification, for verifying whether the state of the guest OS is in one of a predefined set of states.

BotSwindler aims to detect crimeware by deceptively inducing it into an observable action during the exploitation of monitored information injected into the guest OS. To entice attackers with information of value, the system supports a variety of different types of bait credentials including decoy Gmail and PayPal authentication credentials. Our system automatically monitors the decoy accounts for misuse to signal exploitation and thus detect the host infection by credential stealing malware. In other cases, we inject behaviors into the host designed simply to get malware to reveal itself through network activity. As part of this work we demonstrate BotSwindler's utility in detecting malware by means of monitored decoys that are stolen by real crimeware found in the wild and exploited by the adversaries controlling that crimeware.

BotSwindler presents an instance of a system and approach that can be used to deal with information-level attacks, regardless of their origin. In our prototype, we rely on credentials for financial institutions because they are good examples for which we can easily evaluate, but the approach is aimed at any kind of large-scale automated harvesting of "interesting" data — where "interesting" depends on both the environment and the malware. As one of the contributions of this work, we consider different applications of BotSwindler including how it could be applied practically in an enterprise environment with simulations and decoys adapted to the specific deployment setting. In part of doing so, we discuss how BotSwindler can be deployed to service hosts that include those which are not VM-based, making this approach broadly applicable.

The design of BotSwindler's out-of-host simulator relies on the use of a VM that is fully-virtualized (employs binary translation) and the native host on which the VM runs. We demonstrate the prototype version of BotSwindler using Qemu [11] running on Linux, but the approach does not rely on anything implementation specific to these systems. More specifically, the main components of BotSwindler depend on X11 libraries and interaction with the graphical frame buffer. Hence, our approach is generally applicable to other systems that are fully-virtualized (*e.g.,* VMware [12]) and the operating systems on which they are supported.

In summary, the primary contributions of this work include:

- A tamper resistant simulator that is driven by a software agent external to a VM-based host making it difficult to subvert by malware residing within the host.
- A low overhead approach for verifying whether the state of the guest OS is in one of a predefined set of states called virtual machine verification (VMV).
- A novel approach for malware detection that relies on the use of decoy injection whereby bogus information is used to bait and delude crimeware, forcing it to reveal itself during the exfiltration or exploitation of the monitored information.
- An evaluation of the believability of the simulations using a Turing Test whereby human judges are challenged to distinguish between authentic user actions and those generated by BotSwindler.
- A demonstration of BotSwindler's ability to detect active crimeware using financial bait information and bogus email accounts.
- A demonstration of how BotSwindler can be useful tool for dynamic crimeware analysis with a comparison that demonstrates its superiority to other tools in detecting malicious network activity.
- An architecture that describes how BotSwindler could be deployed practically, on a large scale, to serve an enterprise environment with non-VM based hosts.

2 Related Work

Deception-based information resources that have no production value other than to attract and detect adversaries are commonly known as Honeypots [13]. Honeypots serve as effective tools for profiling attacker behavior and to gather intelligence to understand how attackers operate. They are considered to have low false positive rates since they are designed to capture only malicious attackers, except for perhaps an occasional mistake by innocent users. Spitzner discusses the use of honeytokens [14], which he defines as "a honeypot that is not a computer," citing examples that include bogus medical records, credit card numbers, and credentials. Our work harnesses the honeytoken concept to detect crimeware that may otherwise go undetected.

Injecting human input to detect malware has been shown to be useful by Borders *et al.* [15] with their Siren system. The aim of Siren is to thwart malware that attempts to blend in with normal user activity to avoid anomaly detection systems. Detection is performed by manually injecting human input to generate a sequence of network requests and observing the resulting network traffic to identify differences from the known sequences of requests; deviations are flagged as malicious. Expanding upon Siren, Chandrasekaran *et al.* [16] developed a system to randomize generated human input to foil potential analysis techniques that may be employed by malware. Work by Holz *et al.* [1] to investigate keyloggers and dropzones relied on automating user input with AutoIt [17], but was limited to ad hoc scenarios designed for the sole purpose of detecting harvesting channels. Their approach relies on misconfigured and insecure dropzone servers to learn about what sort of information is being stolen. While this effort did reveal lots of interesting details about stolen information, the approach is limited by law and skill of the attackers (they can just secure their dropzone servers). In contrast to these systems, BotSwindler automatically injects input designed to be believable, relies on monitored decoy credentials for detection, and provides a platform to convince malware that it has captured legitimate credentials.

Taint analysis is another technique that has been used to detect credential stealing malware. Eagle *et al.* [18] used taint analysis to track information as it is processed by the web browser and loaded in to browser helper objects (BHOs). Their approach allows for a human analyst to observe where information is being

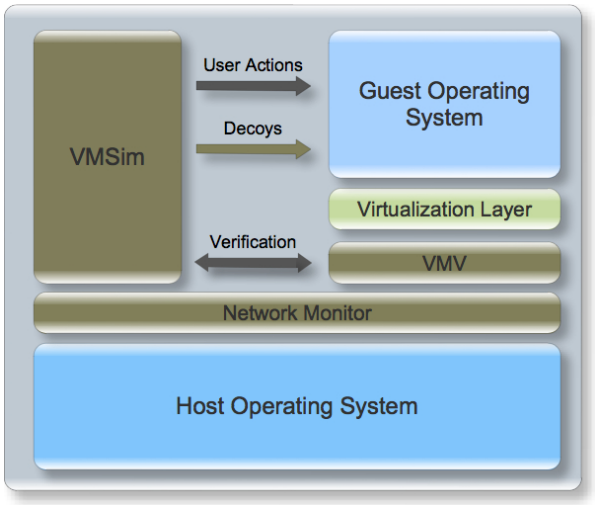

Figure 1: BotSwindler Architecture

sent in offline analysis. Similarly, Yin *et al.* [19] built Panorama, a taint tracking system that extends beyond BHOs to handle tracking throughout multiple processes, memory swapping, and disks. These systems may work well to track information in a system, but they do so with large overhead (factor of 10-20 slowdown in the systems described) or contain components that reside on the guest [19], both features that can be detected by malware and used for evasion purposes.

BotSwindler injects monitored bait into VM-based hosts by simulating user activity that is of interest to crimeware. The simulation is performed on the native OS outside of the VM to minimize artifacts that could be used to tip-off resident malicious software. To keep track of the simulation state within the virtual environment, our approach relies on a form of virtual machine introspection (VMI), a concept proposed by Garfinkel *et al.*[20] to describe the act of inspecting a virtual machine's software from outside the virtual environment. The challenge of VMI lies in overcoming the semantic gap [21] between the two levels of abstraction represented by the VM and the underlying service or OS. Garfinkel *et al.* focused on inspecting memory, registers, device state, and other process related information to implement an attack resistant host-based IDS for VMs whereby the IDS is located outside of the guest in the virtual machine monitor (VMM). Other VMI implementations include [22, 23, 24], but unlike most of these approaches, we circumvent the semantic gap and rely on artifacts found in the VMM graphical framebuffer. To the best of our knowledge, we are the first to focus on the verification of state for user simulations, a challenge with unique requirements.

3 BotSwindler Components

The BotSwindler architecture, as shown in Figure 1, consists of three primary components including a simulator engine, VMSim, a virtual machine verification component, and a network monitor. Another aspect of BotSwindler (although not shown in the figure) are the monitored decoys that we employ for detecting malware. These components are described in the next four sections.

$$
\begin{aligned}
&< ActionType > ::= < WinLogin >< ActionType > \\
&\quad | < CoverType >< ActionType > | < CarryType >< ActionType > \\
&\quad | < WinLogout > | < VerifyAction >< ActionType > | \epsilon \\
&< CoverAction > ::= < BrowserAction >< CoverAction > \\
&\quad | < WordAction >< CoverAction > \\
&\quad | < SysAction >< CoverAction > \\
&< BrowserAction > ::= < URLRequest >< BrowserAction > \\
&\quad | < OpenLink >< BrowserAction > | < Close > \\
&< WordAction > ::= < NewDoc >< WordAction > \\
&\quad | < EditDoc >< WordAction > | < Close > \\
&< SysAction > ::= < OpenWindow > | < MaxWindow > \\
&\quad | < MinWindow > | < CloseWindow > \\
&< VerifyAction > ::= Img1 \mid Img2 \mid ... \mid ImgN \mid Unknown \\
&< CarryAction > ::= < PayPalInject > | < GmailInject > \\
&\quad | < CCInject > | < UnivInject > | < BankInject >
\end{aligned}
$$

Figure 2: VMSim Language

3.1 VMSim

BotSwindler's user simulator component, VMSim, performs simulations that are designed to convince malware residing inside the VM that command sequences are genuine. We posit that successfully creating a sequence of actions that tricks the malware into stealing and uploading a decoy credential can be achieved only if two essential requirements are met:

- the simulator process remains undetected by the malware;
- the actions of the simulator appear to be generated by a human.

We approach the first requirement by decoupling the location of where the simulation process is executed and where its actions are received. To do this, we run the simulator outside of a virtual machine and pass its actions to the guest host by utilizing the X-Window subsystem on the native host. The second requirement is addressed through a simulation creation process that entails recording, modifying, and replaying mouse and keyboard events captured from real users. To support this process, we leverage the Xorg Record and XTest extension libraries for recording and replaying X-Window events. The product is a simulator that runs on the native host producing human-like events without introducing technical artifacts that could be used to alert malware of the BotSwindler facade.

VMSim relies on formal language to specify the sequence of actions in the simulations. Details of the formal language are provided in Figure 2 (many details are omitted due to space limitations). The language provides a flexible way to generate variable simulation behaviors and workflows. It supports the use of *cover* and *carry* actions; carry actions result in the injection of decoys (described in 3.4), whereas cover actions include everything else to support the believability of carry traffic. For example, cover actions may include the opening and editing of a text document (`WordActions`) or the opening and closing of particular windows (`SysActions`). The `VerifyAction` allows VMSim to interact with VMV (described in Section 3.2) and provides support for conditional operations, synchronization, and error checking. Interaction with the VMV is crucial for the accuracy of simulations because a particular action may cause random delays for which the simulation must block on before proceeding to the next action.

The simulation creation process involves the capturing of mouse and keyboard events of a real user as distinct actions. The actions that are recorded map to the constructs of the VMSim language. Once the actions are implemented, the simulator is tuned to mimic a particular user by using various models for keystroke speed, mouse speed and the frequency of errors made during typing. These parameters function as controls over the language shown in Figure 2 and aid in creating variability in the simulations. Depending

on the particular simulation, other parameters such as URLs or other text that must be typed are then entered to adapt each action. VMSim translates the language's actions into lower level constructs consisting of keyboard and mouse functions, which are then output as X protocol level data that can be replayed via the XTest extensions.

One of the advantages of using a language for the generation of simulation workflows is that it produces a specification that can be ported across different platforms. This allows the cost of producing various simulation workflows to be amortized over time. In the prototype version of BotSwindler, the task of mapping mouse and keyboard events to language actions is performed manually a single action at a time. The mappings of actions to lower level mouse and keyboard events are tied to particular host configurations. Although we have not implemented this for the prototype version of BotSwindler, the process of porting these mappings across hosts can be automated using techniques that rely on graphical artifacts like those used in the VMV implementation and applying geometric transformations to them.

As part of the simulation creation process, trial runs of the generated workflows are executed to automatically formulate a whitelist of known and allowed traffic. The whitelist is used by the network monitoring component described in Section 3.3.

Once the simulations are created, playing them back requires VMSim to have access to the display of the guest OS. During playback, VMSim automatically detects the position of the virtual machine window and adjusts the coordinates to reflect the changes. Although the prototype version of BotSwindler relies on the display to be open, it is possible to mitigate this requirement by using the X virtual frame buffer (Xvfb) [25]. By doing so, there would be no requirement to have a screen or input device.

3.2 Virtual Machine Verification

The primary challenge in creating an of out-of-host user simulator is to generate human-like events in the face of variable host responses. This task is essential for being able to tolerate and recover from unpredictable events caused by things like the fluctuations in network latency, OS performance issues, and changes to web content. Conventional in-host simulators have access to OS APIs that allow them to easily to determine such things. For example, simulations created with the popular tool AutoIt [17] can call its `WinWait` function, which can use the Win32 API to obtain information on whether a window was successfully opened. In contrast, an out-of-host simulator has no such API readily available. Although the Xorg Record extensions do support synchronization to solve this sort of problem, it does not work for this particular case. The Record extensions require synchronization on an X11 window as opposed to a window of the guest OS inside of an X11 window, which is the case for guest OS windows of a VM[2].

We address this requirement by casting it as a verification problem to decide whether the current VM state is in one of a predefined set of states. In this case, the states are defined from select regions of the VM graphical output, allowing states to consist of any visual artifact present in a simulation workflow. To support non-deterministic simulations, we note that each transition may end in one of several possible next states. We formalize the VMV process over the set of transitions T, and set of states S, where each $t_0, t_1, ..., t_n \in T$ can result in the the set of states $s_{t1}, s_{t2}, ..., s_{tn} \subseteq S$. The VMV decides a state verified for a current state c, when $c \in s_{ti}$.

The choice for relying on the graphical output allows the simulator to depend on the same graphical features a user would see and respond to, enabling more accurate simulations. In addition, information specific to a VM's graphical output can be obtained from outside of the guest without having to solve the semantic gap problem [21], which requires detailed knowledge of the underlying architecture. A benefit of our approach is that it can be ported across multiple VM platforms and guest OS's. In addition, we do not have to be concerned with side effects of hostile code exploiting a system and interfering with the Win32

[2]This was also a challenge when we tested under VMware Unity, which exports guest OS windows as what appear to be ordinary windows on the native host.

API like traditional in-host simulators do because we do not rely on it. In experiments with AutoIt scripts and in-host simulations, we encountered cases where scripts would fail as a result of the host being infected with malware.

The VMV was implemented by extending the Simple DirectMedia Layer (SDL) component of Qemu's [11] VMM. Specifically, we added a hook to the `sdl_update` function to call a VMV `monitor` function. This results in the VMV being invoked every time the VM's screen is refreshed. The choice of invoking the VMV only during `sdl_update` was both to reduce the performance costs and because it is precisely when there are updates to the screen that we seek to verify states (it is a good indicator of user activity).

States are defined during a simulation creation process using a pixel selection tool (activated by hotkeys) that we built into the VMM. The pixel selection tool allows the simulation creator to select any portion of a guest OS's screen for use as a state. In practice, the states should be defined for any event that may cause a simulation to delay (*e.g.,* network login, opening an application, navigating to a web page). The size of the screen selection is left up to the discretion of the simulation creator, but typically should be minimized as it may impact performance. In Section 4.2 we provide a performance analysis to aid in this consideration.

3.3 Network Monitoring

BotSwindler provides *detection in depth* with an additional layer of network monitoring to detect when malware attempts to exfiltrate its quarry. The network monitor functions by recording and alerting when there is malicious traffic originating from the VM host. To identify the malicious traffic, we leverage the fact that the simulator has control of most of the traffic emanating from the host. This enables the use of a whitelist of known and allowed traffic to distinguish from the malicious. The whitelist is constructed as part of the simulation generation process as described in Section 3.1.

3.4 Trap-based Decoys

Our trap-based decoys are detectable outside of a host by external monitors, so they do not require host monitoring nor do they suffer the performance burden characteristic of decoys that require constant internal monitoring (such as those used for taint analysis). They are made up of *bait information* such as online banking logins provided by a collaborating financial institution [3], login accounts for online servers, and web based email accounts. In this study, we focus on the use of decoy Gmail and PayPal credentials. These were chosen because they are widely used and known to have underground economy value [26, 1], making them alluring targets for crimeware, yet inexpensive for us to create. The decoy PayPal accounts have an added bonus that allows us to expose the credentials without having to be concerned about an attacker changing the password. PayPal requires multi-factor authentication to change passwords on an account. Since we do not reveal all of the attributes of an account, it is difficult for an attacker to change the password.

Custom monitors for PayPal and Gmail accounts were developed to leverage internal features of the services that provide the time of last login, and in the case of Gmail accounts, IP address of the last login. In the case of PayPal, the monitor logs into the decoy accounts every hour to check the PayPal recorded last login. If the delta between the times is greater than 75 seconds, the monitor triggers an alert for the account and notifies us by email. The 75 second threshold was chosen because PayPal reports the time to a resolution of minutes rather than seconds. In addition, our experiments revealed fluctuations between the time differences of PayPal's recorded last login and the BotSwindler monitor's time making it necessary to add additional time to the threshold. These differences are due to time synchronization issues and latency in the PayPal login process. The choice as to what time interval to use and how frequently to poll presents significant tradeoffs that we analyze in Section 4.3.

[3] By agreement, the institution requested that its name be withheld.

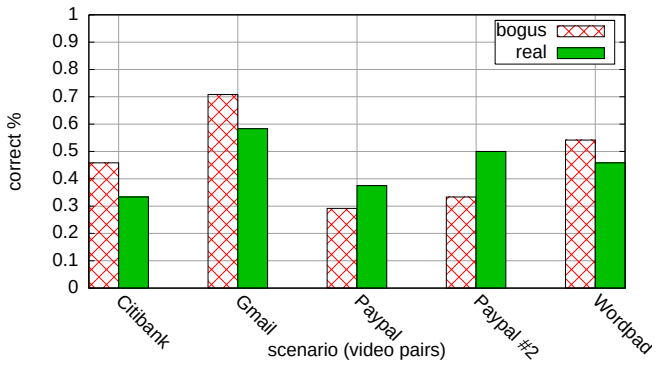

Figure 3: Decoy Turing Test results: Real vs. Simulated.

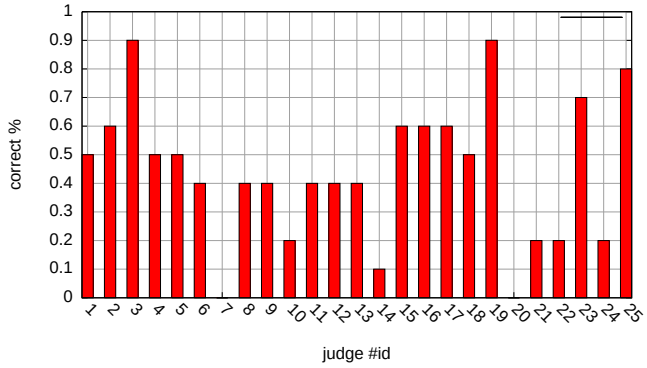

Figure 4: Judges' overall performance.

In the case of the Gmail accounts, custom scripts access `mail.google.com` to parse the bait account pages, gather- ing account activity information. The information includes the IP addresses for the previous 5 account accesses and the time. If there is any activity from IP addresses other than the BotSwindler monitor's host IP, an alert is triggered with the time and IP of the offending host. Alerts are also triggered when the monitor cannot login to the bait account. In this case, we conclude that the account password was stolen (unless monitoring resumes) and maliciously changed unless other corroborating information (like a network outage) can be used to convince otherwise.

4 Experimental Results

4.1 Decoy Turing Test

In this section we present the experimental results of a Turing Test [10] to demonstrate BotSwindler's performance regarding the *humanness* or believability of the generated simulations. Although the simulations are designed to delude crimware, here we focus on convincing humans, a task we posit to be a more difficult feat making the adversaries task of designing malware that discerns decoys far more difficult. To conduct this study, we formed a pool of 25 human judges, consisting of security-minded PhDs, graduate-level stu-

dents, and security professionals. Their task was to observe a set of 10 videos that capture typical user actions performed on a host and make a binary decision about each video: *real* or *simulated* (*i.e.,* whether the video shows the actions of a real user or those of a simulator). Our goal was to demonstrate believability of the simulated actions by showing failure of human judges to reliably distinguish between authentic human actions and those generated with BotSwindler. Our videos contained typical user actions performed on a host such as composing and sending an email message through Gmail, logging into a website of a financial institution such as Citibank or PayPal, and editing text document using Wordpad. For each scenario we generated two videos: one that captured the task performed by a human and another one that had the same task performed by BotSwindler. Each video was designed to be less than a minute long since we assumed that our judges would have limited patience and would not tolerate long-running simulations.

The human generated video samples were created by an independent user who was asked to perform sets of actions which were recorded with a desktop recording tool to obtain the video. Similar actions by another user were used to generate keystroke timing and error models, which could then be used by VMSim to generate keystroke sequences. To generate mouse movements, we rely on movements recorded from a real user. Using these, we experimentally determine upper and lower bounds for mouse movement speed and replay the movements from the real user, but with a new speed randomized within the determined limits. The keyboard and mouse sequences were merged with appropriate simulator parameters such as credentials and URLs to form the simulated sequence which was used to create the decoy videos. To make the environments as similar as possible, the actions of the real user and those of the simulator were recorded on the same Windows XP OS running as a Qemu VM.

Figure 3 summarizes the results for each of the 10 videos. The videos are grouped in per-scenario pairs in which the left bars correspond to simulated tasks, while the right bars correspond to the tasks of authentic users on which the simulations are based. The height of the bars reflects the number of judges that correctly identified the given task as real or simulated. The overall success rate was $\sim$46%, which indicates that VMSim achieves a good approximation of human behavior. The ideal success rate is 50%, which suggests that judges cannot differentiate whether a task is simulated or real.

Figure 4 illustrates the overall performance of each judge separately. The judges' correctness varies greatly from 0% up to 90%. This variability can be attributed to the fact that each judge interprets the same observed feature completely differently. For example, since VMSim uses real user actions as templates to drive the simulation, it is able to include advanced "humanized" actions inside simulations, such as errors in typing (*e.g.,* invalid typing of a URL that is subsequently corrected), `TAB` usage for navigating among form fields, auto-complete utilization and so forth. However, the same action (*e.g.,* `TAB` usage for navigating inside the fields of a web form) is assumed by some judges as a real human indicator, while some others take it as a simulation artifact. This observation is clearly a "toss up" as a distinguishing feature. An important observation is that even highly successful judges could not achieve a 100% accuracy rate. This indicates that given a diverse and plentiful supply of decoys, our system will be believable at some time. In other words, given enough decoys, BotSwindler will eventually force the malware to reveal itself. We note that there is a "bias" towards the successful identification of bogus videos compared to real videos. This is might be due to the fact that most of the judges guess "simulated" when unsure due to the nature of the experiment. Despite this bias, results indicate that simulations are highly believable by humans. In cases where they may not be, it is important to remember that the task of fooling humans is far harder than tricking malware, unless the adversary has solved the AI problem and designed malware to solve the Turing Test. Furthermore, if attackers have to spend their time looking at the actions one by one to determine if they are real or not, we consider BotSwindler a success because that approach does not scale for the adversary.

4.2 Overhead of Virtual Machine Verification

The overhead of the VMV in BotSwindler is controlled by several parameters including the number of pixels

Table 1: Overhead of VMV with idle user

	Min.	Max.	Average	Std. Deviation
Native OS	.48	.70	.56	.06
Qemu	.55	.95	.62	.07
Qemu w/VMV	.52	.77	.64	.07

Table 2: Overhead of VMV with active user

	Min.	Max.	Average	Std. Deviation
Native OS	.50	.72	.56	.06
Qemu	.57	.96	.71	.07
Qemu w/VMV	.53	.89	.71	.06

in the screen selections, the size of the search area for a selection, the number of possible states to verify at each point of time, and the number of pixels required to match for positive verification. A key observation responsible for maintaining low overhead is that the majority of the time, the VMV process results in a negative verification, which is typically obtained by inspecting a single pixel for each of the possible states to verify. The performance cost of this result is simply that of a few instructions to perform pixel comparisons. The worst case occurs when there is a complete match in which all pixels are compared (*i.e.,* all pixels up to some predefined threshold). This may result in thousands of instructions being executed (depending on the particular screen selection chosen by the simulation creator), but it only happens once during the verification of a particular state. It is possible to construct a scenario in which worse performance is obtained by choosing screen selections that are common (*e.g.,* found on the desktop) and almost completely matches but results in a negative VMV outcome. In this case, obtaining a negative VMV result may cost hundreds of thousands of CPU cycles. In practice, we have not found this scenario to occur; moreover, it can be avoided by the simulation creator.

In Table 1, we present the analysis of the overhead of Qemu[4] with the BotSwindler extensions. The table presents the amount of time, in seconds, to load web pages on our test machine (2.33GHz Intel Core 2 Duo and 2GB 667MHz DDR2 SDRAM) with idle user activity. The results include the time for a native OS, an unmodified version of Qemu (version 0.10.5) running Windows XP, and Qemu running Windows XP with the VMV processing a verification task (a particular state thousands of pixels in size). We included tests for the native OS as a reference point, but in determining the overhead of the VMV they are not relevant because we are only concerned with the time differences between web page loads on Qemu without the VMV versus the web page load times with the VMV. In Table 2, we present the results from a second set of tests where we introduce rapid window movements forcing the screen to constantly be refreshed. By doing this, we ensure that the BotSwindler VMV functions are repeatedly called. The results indicate that the rapid movements do not impact the performance on the native OS, whereas in the case of Qemu they result in a $\sim$15% slowdown. This is likely because Qemu does not support graphics acceleration, so all processing is performed by the CPU. The time to load the web pages on Qemu with the VMV is essentially the same as without it. This is true whether the tests are done with or without user activity. Hence, we conclude that the performance overhead of the VMV is negligible.

[4]Qemu does not support graphics acceleration, so all processing is performed by the CPU.

Table 3: PayPal decoy false negative likelihoods.

Polling Frequency	False Negative Rate
.5 hour	.0417
1 hour	.0208
24 hour	.0009

4.3 PayPal Decoy Analysis

The PayPal monitor relies on the time differences recorded by the BotSwindler monitoring server and the PayPal service for a user's last login. The last login time displayed by the PayPal service is presented with a granularity of minutes. This imposes the constraint that we must allow for at least one minute of time between the PayPal monitor, which operates with a granularity of seconds, and the PayPal service times. In addition, we have observed that there are slight deviations between the times that can likely be attributed to time synchronization issues and latency in the PayPal login process. Hence, it is useful to add additional time to the threshold used for triggering alerts (we make it longer than the minimum resolution of one minute).

Another parameter that influences the detection rate is the frequency at which the monitor polls the PayPal service. Unfortunately, it is only possible to obtain the last login time from the PayPal service, so we are limited to detecting a single attack between polling intervals. Hence, the more frequent the polling, the greater the number of attacks on a single account that we can detect and the quicker an alert can be generated after an account has been exploited. However, the fact that we must allow for a minimum of one minute between the PayPal last login time and the BotSwindler monitor's, implies we must consider a significant tradeoff. The more frequent the polling, the greater the likelihood for false negatives due to the one minute window. In particular, the likelihood of a false negative is:

$$P_{FN} = \frac{\text{Length of window}}{\text{Polling interval}}$$

Table 3 provides examples of false negative likelihoods for different polling frequencies using a 75 second threshold. These rates assume only a single attack per polling interval. We rely on a 75 second threshold because it was experimentally determined that it exhibits no false positives. For the experiments described in Section 4.4, we use the 1 hour polling frequency because we believe it provides an adequate balance (the false negative rate is low enough and the alerts are generated quickly enough).

4.4 Detecting Real Malware with Bait Exploitation

To demonstrate the efficacy of our approach, we designed an experiment to test BotSwindler on crimeware found in the wild. In particular, we focused on Zeus because of the large number of variants that exist and their potential impact. Despite the abundant supply of Zeus variants, many are no longer functional because they require active command and control servers to effectively operate. This requirement gives Zeus a relatively short life span because these services become inactive (*e.g.,* they are on a compromised host that discovered and sanitized). To obtain active Zeus variants, we subscribed to an active feed of binaries at the Swiss Security blog, which has a Zeus Tracker [6] and Offensive Computing [27], where we have an email feed of the latest submissions.

To conduct our experiments, we created 5 PayPal decoys and 5 Gmail decoys. We deliberately limited the number of accounts to avoid upsetting the providers and having our access removed. After all, the use of these accounts as decoys requires us to continuously poll the servers for unauthorized logins as described

in Section 4.3, which could become problematic with a large number of accounts. To further limit the load on the services, we limited the BotSwindler monitoring to once every hour.

We constructed a BotSwindler sandbox environment so that any access to `www.paypal.com` would be routed to a decoy website that replicates the look-and-feel of the true PayPal site. This was done for two reasons. First, if BotSwindler accessed the real PayPal site, it would be more difficult for the monitor to differentiate access by the simulator from an attacker, which could lead to false positives. More importantly, hosting a phony PayPal site enabled us to control attributes of the account (*e.g.,* balance and verified status) to make them more enticing to crimeware. We leveraged this ability to give each of our decoy accounts unique balances in the range of 4,000 - 20,000 USD, whereas in the true PayPal site, they have no balance. In the case of Gmail, the simulator logs directly into the real Gmail site, since it does not interfere with monitoring of the accounts (we can filter on IP) and there is no need to modify account attributes.

The decoy PayPal environment was setup by copying and slightly modifying the content from `www.paypal.com` to a restricted lab machine with internal access only. The BotSwindler host machine was configured with NAT rules to redirect any access directed to the real PayPal website to our test machine. The downside of using this setup is that we lack a certificate to the `www.paypal.com` domain signed by a trusted Certificate Authority. To mitigate the issue, we used a self-signed certificate that is installed as a trusted certificate on the guest. Although this is a potential distinguishing feature that can be used by malware to detect the environment, existing malware is unlikely to check for this. Hence, it remains a valid approach for demonstrating the use of decoys to detect malware in this proof of concept experiment. We have other decoys that do not have this limitation, but they may not have the same broad appeal to attackers that make PayPal accounts so useful.

The experiments worked by automating the download and installation of individual malware samples using a remote network transfer. For each sample, BotSwindler conducted various simulations designed from the VMSim language to contain inject actions, as well as other cover actions. The simulator was run for approximately 20 minutes on each of the binaries that were tested. Over the course of five days of monitoring, we received nine alerts from the PayPal monitor and one Gmail alert. The Gmail alert was for a Gmail decoy ID that was also associated with a decoy PayPal account; the Gmail username was also a PayPal username and both credentials were used in the same workflow (we associate multiple accounts to make a decoy identity more convincing). Given that we received an alert for the PayPal ID as well, it is likely both sets of credentials were stolen at the same time. Although the Gmail monitor does provide IP address information, we could not obtain it in this case. This particular alert was generated because Gmail detected suspicious activity on the account and locked it, so the intruder never got in.

We attribute the fewer Gmail alerts to the economics of the blackmarket. Although Gmail accounts may have value for activities such as spamming, they can be purchased by the thousands for very little cost[5] and there are inexpensive tools that can be used to create them automatically [28]. Hence, attackers have little incentive to build or purchase a malware mechanism, and to find a way to distribute it to many victims, only to net a bunch of relatively valueless Gmail accounts. On the other hand, high-balance verified PayPal accounts represent something of significant value to attackers. The 2008 Symantec Global Internet Security Threat Report [26] lists bank accounts (which PayPal falls in the category of) as being worth $10-$1000 on the underground market depending on balance.

For the PayPal alerts that were generated, we found that some alerts were triggered within an hour after the corresponding decoy was injected, where other alerts occurred days after. We believe this variability to be a consequence of attackers manually testing the decoys rather than testing through some automatic means. In regards to the quantity of alerts generated, there are several possible explanations that include:

- As a result of the one-to-many mapping between decoys and binaries, the decoys are exfiltrated to many different dropzones where they are then tested.

[5] We have found Gmail accounts being sold at $20 per 1000.

Table 4: Exfiltration detection results for different crimeware samples

Name	User Action Required	CWSandbox	Anubis	ThreatExpert	BotSwindler
Zeus	No	-	-	X[6]	X
Agent	Yes	-	-	-	X
Banker	Yes	-	-	-	X
Bzub	Yes[7]	-	-	X	X
Sinowal	Yes	-	-	-	X
HaxSpy	Yes	-	-	-	X
Goldun	Yes	-	-	-	-
Small	Yes	-	-	-	X
NetSky	No	X	X	X	X

- The decoy accounts are being sold and resold in the underground market where first the dropzone owner checks them, then resell them to others, who then resell them to others who check them.

While the second case is conceivable for credentials of true value, our decoys lack any balance. Hence, we believe that once this fact is revealed to the attacker during the initial check, the attackers have no reason to keep the credentials or recheck them (lending support for the first case). We used only five PayPal accounts with a one-to-many mapping to binaries making it impossible to know exactly which binary triggered the alert and which scenario actually occurred. We also note that the number of actual attacks may be greater than what was actually detected. The PayPal monitor polls only once per hour, so we do not know when there are multiple attacks in a single hour. Hence, the number of attacks we detected is a lower bound on the number of actual attacks. Nevertheless, these results do validate the use of financial decoys for detecting crimeware. A BotSwindler system fully developed as a deployable product would naturally include many more decoys and a management system that would store information about which decoy was used and when it was exposed to the specific tested host.

4.5 Detecting Malware through Exfiltration

BotSwindler is designed to induce crimeware into observable action by convincing it that simulated user actions are human. Observable action may either be the exploitation of bait information or network activity as described in this section. Detecting malicious network activity provides utility as a means of malware detection and as a tool in the analysis of malware to aid in profiling attacker goals. To demonstrate the efficacy of BotSwindler at eliciting the network activity of malware, we provide a comparison of BotSwindler against other tools used for dynamic malware analysis.

The study included a comparison of four tools including CWSandbox [29], ThreatExpert [30], Anubis [31], and BotSwindler in their network analysis of information stealing malware. The malware samples we used were obtained from VX Heavens [32], Offensive Computing [27], and The Swiss Blog [6]. Table 4 provides a summary of the results of our comparisons. Since most of the samples have numerous aliases, we have used the most common name in this table and included a mapping to the MD5 hashes in Table 5 for a positive identification.

The selection of malware test samples varied by the type of information the malware attempts to steal. The test samples range from some recently discovered in the wild, such as the Zeus sample, to older samples like Banker. Within the table we distinguish between samples that require user actions to draw network activity versus those that do not. Detecting network activity on malware that is triggered by user actions or user information is what differentiates BotSwindler from the rest. As a control, we included a sample of the

Table 5: Identification of Malware Samples

Name	MD5
Zeus	650F84C94C85155C8F29F7CDE1F47F6B
Agent	259D99514E860BBB9979F4B423EDE204
Banker	EBA9B013AD15AAAFBB2488A0EB8BF8E7
Bzub	3960376D373FDB2C2D96E95B498B3C8F
Sinowal	1A1FA2C628D633039AF9BABD158D448B
HaxSpy	07D75865135DDD45416F7349235238A8
Goldun	F364EC032B9BCF7E8CBC64CF07FC4347
Small	5A435FF3BA3FF620D60A08A4C3C92A6C
NetSky	3D23EC8B55840B95EA75197CE9446B6D

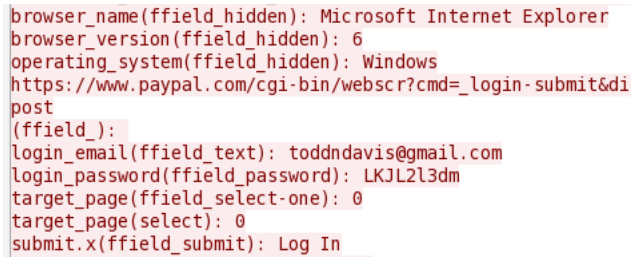

Figure 5: Example of a Sinowal variant exfiltrating a set of BotSwindler's decoy PayPal credentials.

NetSky email worm, which automatically attempts to email itself upon infecting a host. The goal of using this as a control was to confirm that the analysis tools do indeed monitor for network activity. We note that all of tools detected the network activity of NetSky.

To conduct the study, we configured BotSwindler to run workflows with actions that access a well-known bank, PayPal, and Gmail servers. In addition, we included a set of actions for document creation and editing, which resulted in a large number of keystrokes being entered. As with any dynamic analysis tool, one of the limitations lies in providing sufficient test coverage to provoke malware into observable action. For example, some malware can have very specific targets (like a single URL) that it will look for before acting. While this is a challenge, a vast majority of information stealing software we experimented with seemed to be non-target specific or to have a pre-determined list of targets that include some of the most popular services. We leveraged this point in designing our simulations.

Table 4 provides a summary of our results against nine unique malware samples. Note that most of these

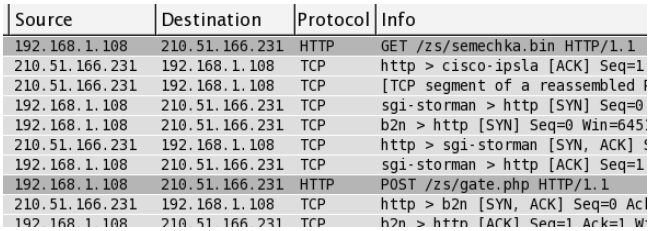

Figure 6: Sample network traffic displayed in Wireshark for a typical Zeus bot.

malware samples represent a class for which there are numerous variants on which these results also apply. As shown in the table, in most cases, we were able to detect network activity where the other tools did not. Figure 5 shows an example of network activity elicited from a Sinowal trojan. In this case, we were able to see the actual decoy credentials being exfiltrated in unencrypted network traffic. We note that in some of the other cases, we detected network activity, but it was encrypted, as in the case of Zeus (external misuse monitoring of stolen credentials covers the cases where stealthy malware exfiltrates credentials over covert channels).

In two cases, ThreatExpert was also able to detect some activity. In the case of Zeus, it detected and reported an `HTTP GET` request for configuration from a foreign server. A partial network trace from a Zeus bot is shown in Figure 6. The `HTTP GET` is shown by the first of the two shaded packets and represents the first of out-going requests for the Zeus bot. ThreatExpert did not detect any communication beyond the initial one, such as `HTTP POST`s (shown by the second shaded packet) that are used as part a typical exfiltration process, where BotSwindler did. In the case of Bzub, ThreatExpert reported on network activity that is triggered by the opening of a browser process, which can possibly be done without user interaction.

One particular case where BotSwindler did not succeed was for an old password stealer program called Goldun. As with the other tools, we failed to see any network activity in our tests because our simulator did not access `http://www.e-gold.com`, for which this malware specifically targets. This example highlights the limitation of this approach in detecting and automatically analyzing malware in that insufficient test case coverage can lead to false negatives. While this is a concern for general malware analysis, we note that there are other cases where it might be appropriate to have a narrow range of test coverage, as described in Section 5.

5 Applications of BotSwindler in an Enterprise

Beyond the detection of malware using general decoys, BotSwindler is well suited to be deployed in an Enterprise environment where the primary goal is to monitor for site-specific credential misuse and to profile attackers targeting that specific environment. Since the types of credentials that are used within an enterprise are typically limited to business applications for specific job functions, rather than general purpose uses, it is feasible for BotSwindler to provide complete test coverage in this case. For example, typical corporate users have a single set of credentials for navigating their company intranet. Corporate decoy credentials could be used by BotSwindler in conducting simulations modeled after individuals within the corporation. These simulations may emulate system administrative account usage (i.e. logging in as root), access to internal databases, editing of confidential documents, navigating the internal web, and other workflows that apply internally. Furthermore, software monocultures with similar configurations, such as those found in an enterprise, may simplify the task of making a single instance of BotSwindler operable across multiple hosts.

Within the enterprise environment, BotSwindler can be deployed to run simulations on a user's system when it is idle (during meetings, at night, etc.). Although virtual machines are common in enterprise environments, in cases where they are not used, they can be created on demand from a user's native environment [12]. Figure 7 provides an illustration of one possible application of BotSwindler where it is deployed as an enterprise service that runs simulation over exported copies of multiple users' disk images. In another approach, a user's machine state could be synchronized with the state of a BotSwindler enabled virtual machine [33]. In either case, these applications of BotSwindler allow it to tackle the problem of malware performing long-term corporate reconnaissance. For example, malware might attempt to steal credentials only after they have been repeatably used in the past. This elevates the utility of BotSwindler from a general malware detector to one capable of detecting targeted espionage software.

The application of BotSwindler to an enterprise would require adaptation for site-specific things (*e.g,*

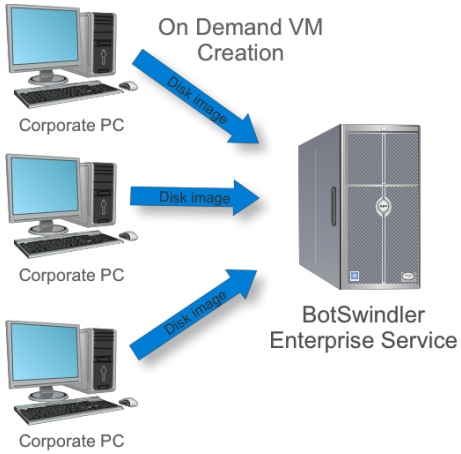

Figure 7: Diagram of BotSwindler Enterprise Service

internal URLs), but use of specialized decoys does not preclude the use of general decoys like those detailed in Section 3.4. General decoys can help the organization identify compromised internal users that could be, in turn, the target of blackmail, either with traditional means or through advanced malware [34].

6 Limitations and Future Work

Our approach of detecting malware relies on the use of deception to trick malware to capture decoy credentials. As part of this work, we evaluated the believability of the simulations, but we did so in a limited way. In particular, our study measured the believability of short video clips containing different user workflows. These types of workflows are adequate for the detection of existing threats using short-term deception, but for certain use cases (such as the enterprise service) it is necessary to consider long-term deception, and the believability of simulation command sequences over extended periods of time. For example, adversaries conducting long-term reconnaissance on a system may be able to discover some invariant behavior of BotSwindler that can be used to distinguish real actions from simulated actions, and thus avoid detection. To counter this threat, more advanced modeling is needed to be able to emulate users over extended periods of time, as well as a study that considers the variability of actions over time. For long-term deception, the types of decoys used must also be considered. For example, some malware may only accept as legitimate those credentials that it has seen several times in the past. We can have "sticky" decoy credentials of course, but that negates one of their benefits (determining when a leak happened).

Malware may also be able to distinguish BotSwindler from ordinary users by attempting to generate bogus system events that cause erratic system behavior. These can potentially negatively impact a simulation and cause the simulator to respond in ways a real user would not. In this case, the malware may be able to distinguish between authentic credentials and our monitored decoys. Fortunately, erratic events that result in workflow deviations or simulation failure are also detectable by BotSwindler because they result in a state that cannot be verified by the VMV. When BotSwindler detects such events, it signals the host is possibly infected. The downside of this strategy is that it may result in false positives. As part our future work we will investigate how to measure and manage this threat using other approaches that ameliorate this weakness.

7 Conclusion

BotSwindler is a bait injection system designed to delude and detect crimeware by forcing it to reveal itself during the exploitation of monitored decoy information. It relies on an out-of-host software agent to drive user-like interactions in a virtual machine aimed at convincing malware residing within the guest OS that it has captured legitimate credentials. As part of this work we have demonstrated BotSwindler's utility in detecting malware by means of monitored financial bait that is stolen by real crimeware found in the wild and exploited by the adversaries that control that crimeware. We have presented the results of experiments that show how BotSwindler's simulated workflows can be used to induce malware into observable network action, which can then be detected. These experiments also demonstrate that BotSwindler is capable of eliciting and detecting network activity in malware samples where other systems fail. In anticipation of malware seeking the ability to distinguish simulated actions from human actions, we designed our system to be difficult to detect by the underlying architecture and the believable actions it generates. To demonstrate the believability of the simulations, we conducted a Turing Test that showed we could succeed in convincing humans about 46% of the time. We assert that if attackers are forced to spend their time looking at the actions on each host it infects one by one to determine if they are real or not in order to steal information, BotSwindler would be a success; the crimeware's task does not scale.

References

[1] T. Holz, M. Engelberth, and F. Freiling, *Learning More about the Underground Economy: A Case-Study of Keyloggers and Dropzones*, ser. Lecture Notes in Computer Science (LNCS). Springer Berlin / Heidelberg, September 2009, vol. 5789, pp. 1–18.

[2] M. Sthlberg, "The trojan money spinner," F-Secure Corporation, Technical report, September 2007. [Online]. Available: http://www.f-secure.com/weblog/archives/VB2007_TheTrojanMoneySpinner.pdf

[3] (2009, July) Researcher uncovers massive, sophisticated trojan targeting top businesses. Darkreading. [Online]. Available: http://www.darkreading.com/database_security/security/privacy/showArticle.jhtml?articleID=218800077

[4] K. J. Higgins. (2009, September) Up to 9 percent of machines in an enterprise are bot-infected. Darkreading. [Online]. Available: http://www.darkreading.com/insiderthreat/security/client/showArticle.jhtml?articleID=220200118

[5] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo, "On the infeasibility of modeling polymorphic shellcode," in *Proc. of the 14th ACM conference on Computer and Communications Security (CCS)*, Alexandria, VA, USA, 2007, pp. 541–551.

[6] (2009, November) abuse.ch zeus tracker. The Swiss Security Blog. [Online]. Available: https://zeustracker.abuse.ch/index.php

[7] E. Messmer. (2009, July) America's 10 most wanted botnets. NETWORKWORLD. [Online]. Available: http://www.networkworld.com/news/2009/072209-botnets.html

[8] "Measuring the in-the-wild effectiveness of antivirus against zeus," Trusteer, Technical report, September 2009. [Online]. Available: www.trusteer.com/files/Zeus_and_Antivirus.pdf

[9] D. Ilett. (2005, February) Trojan attacks microsoft's anti-spyware. CNET News. [Online]. Available: http://news.cnet.com/Trojan-attacks-Microsofts-anti-spyware/2100-7349_3-5569429.html

[10] A. M. Turing, "Computing machinery and intelligence," *Mind, New Series*, vol. 59, no. 236, pp. 433–460, October 1950.

[11] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proc. of USENIX Annual Technical Conference*, Anaheim, CA, USA, April 2005, pp. 41–46.

[12] (2009, November) VMware. [Online]. Available: http://www.vmware.com

[13] (2009, November) Honeypots, intrusion detection, incident response. [Online]. Available: http://www.honeypots.net

[14] L. Spitzner. (2003, July) Honeytokens: The other honeypot. SecurityFocus. [Online]. Available: http://www.securityfocus.com/infocus/1713

[15] K. Borders, X. Zhao, and A. Prakash, "Siren: Catching evasive malware," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, USA, May 2006, pp. 78–85.

[16] M. Chandrasekaran, S. Vidyaraman, and S. Upadhyaya, "Spycon: Emulating user activities to detect evasive spyware," in *Proc. of the Performance, Computing, and Communications Conference (IPCCC)*, New Orleans, LA, USA, May 2007, pp. 502–509.

[17] (2009, November) AutoIt Script. [Online]. Available: http://www.autoitscript.com

[18] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic spyware analysis," in *Proc. of the USENIX Annual Technical Conference*, Santa Clara, CA, USA, June 2007, pp. 233–246.

[19] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panaroma: Capturing system-wide information flow for malware detection and analysis," in *Proc. of the 14^{th} ACM conference on Computer and Communications Security (CCS)*, Alexandria, VA, USA, 2007, pp. 116–127.

[20] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. of Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, USA, February 2003.

[21] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proc. of the 8^{th} Workshop on Hot Topics in Operating System (HotOS)*, Washington, DC, USA, May 2001, pp. 133–138.

[22] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *Proc. of the USENIX Annual Technical Conference*, Boston, MA, USA, March 2006, pp. 1–14.

[23] X. Jiang and X. Wang, ""Out-of-the-Box" monitoring of vm-based high-interaction honeypots," in *Proc. of the 10^{th} International Symposium on Recent Advances in Intrusion Detection (RAID)*, Cambridge, MA, USA, September 2007, pp. 198–218.

[24] A. Srivastava and J. Giffin, "Tamper-resistant, application-aware blocking of malicious network connections," in *Proc. of the 11^{th} International Symposium on Recent Advances in Intrusion Detection (RAID)*, Cambridge, MA, USA, September 2008, pp. 35–58.

[25] (2009, November) Xvfb(1). The XFree86 Project, Inc. [Online]. Available: http://www.xfree86.org/4.0.1/Xvfb.1.html

[26] Symantec, "Trends for july - december '07," White paper, April 2008. [Online]. Available: eval.symantec.com/.../b-whitepaper_exec_summary_internet_security_threat_report_xiii_04-2008.en-us.pdf

[27] (2009, November) Offensive computing. [Online]. Available: http://www.offensivecomputing.net

[28] (2009, November) Gmail account creator. [Online]. Available: http://www.gmailaccountcreator.com

[29] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, USA, March 2007, pp. 32–39.

[30] (2009, November) Threatexpert. [Online]. Available: http://www.threatexpert.com

[31] (2009, November) Anubis: Analyzing unknown binaries. [Online]. Available: http://anubis.iseclab.org

[32] (2009, November) Vx heavens. [Online]. Available: http://vx.netlux.org

[33] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA, April 2008, pp. 161–174.

[34] M. Bond and G. Danezis, "A pact with the devil," in *Proc. of the New Security Paradigms Workshop (NSPW)*, Dagstuhl, Germany, September 2006, pp. 77–82.