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

Brian M. Bowen [1], Pratap Prabhu [1], Vasileios P. Kemerlis [1], Stelios Sidiroglou [2], Angelos D. Keromytis [1], and Salvatore J. Stolfo [1]

[1] Department of Computer Science, Columbia University
{bb2281, pvp2105, vk2209, ak2052, sjs11}@columbia.edu
[2] Computer Science and Artificial Intelligence Laboratory, MIT
stelios@csail.mit.edu

Abstract. We introduce BotSwindler, a bait injection system designed to delude and detect crimeware by forcing it to reveal during the exploitation of monitored information. The implementation of BotSwindler relies upon an out-of-host software agent that drives 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 propose a low overhead approach, called virtual machine verification, for verifying whether the guest OS is in one of a predefined set of states. We present results from experiments with real credential-collecting malware that demonstrate the injection of monitored financial bait for detecting compromises. Additionally, using a computational analysis and a user study, we illustrate the believability of the simulations and we demonstrate that they are sufficiently human-like. Finally, we provide results from performance measurements to show our approach does not impose a performance burden.

1 Introduction

The creation and rapid growth of an underground economy that trades in stolen digital credentials has spurred the growth of crime-driven bots that harvest sensitive data from unsuspecting users. This form of malevolent software employs a variety of techniques ranging from web-based form grabbing and key stroke logging, to screenshots and video capture for the 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 that 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

* This work was partly supported by the National Science Foundation through grants CNS-07-14647 and CNS-09-14312. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

of possible variants, learning and searching all possible signatures to identify unknown binaries is intractable [5]. On the other hand, 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[3] (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].

Another drawback of conventional host-based antivirus software is that it typically monitors from within the host it is trying to protect, making it vulnerable to evasion or subversion by malware; 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 Turing Test [10]. We assert that if attackers are forced to spend their time looking at the actions on each infected host one by one to determine if they are real or not in order to steal information, BotSwindler would be a success; the attackers' 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. Simulations can be tuned to mimic particular users by using various models for keystroke speed, mouse speed and the frequency of errors made during typing.

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 (VMV), 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, as well as those from a large financial institution [4]. Our system automatically monitors

[3] 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].

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

the decoy accounts for misuse to signal exploitation and thus detect the host infection by credential stealing malware.

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 that 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. Although we demonstrate our system with three types of credentials, the system can be extended to support any type of credential that can be monitored for misuse. 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.

We have implemented a prototype version of BotSwindler using a modified version of QEMU [11] running on a Linux host. User simulation is implemented using X11 libraries and interaction with the graphical frame buffer. We demonstrate our prototype through experiments with crimeware on a Windows guest, but BotSwindler can operate on any guest operating system supported by the underlying hypervisor or virtual machine monitor (VMM).

1.1 Overview of Results

To demonstrate the effectiveness of BotSwindler, we tested our prototype against real crimeware samples obtained from the wild. Our results from two separate experiments with different types of decoy credentials show that BotSwindler succeeds in detecting malware through attackers' exploitation of the monitored bait. In our first experiment with 116 Zeus samples, we received 14 distinct alerts using PayPal and Gmail decoys. In a second experiment with 59 different Zeus samples, we received 3 alerts from our banking decoys.

The long-term viability of BotSwindler defense largely depends on the believability of the bait-injecting simulations by the attackers. We performed a computational analysis to see if attackers could employ machine learning algorithms on keystrokes to distinguish simulations. We present results from experiments running Naive Bayes and Support Vector Machine (SVM) classifiers on real and generated timing data to show that they produce nearly identical classification results making this kind of analysis ineffectual for an adversary. To show that adversaries resorting to manual inspection of the user activities would be sufficiently challenged, we evaluated the believability of user simulations via a decoy Turing Test in which human judges were tasked with trying to distinguish BotSwindler's actions from those of a real human. The failure of the judges to distinguish suggests BotSwindler's simulations are convincingly human-like. In our study with 25 human judges evaluating 10 videos of BotSwindler actions and of a human, the judges' average success rate was 46%, indicating the simulations provide a good approximation of human actions.

Finally, recognizing that attackers may try to distinguish simulated behavior via performance metrics, we evaluated the overhead of our approach by measuring the cost

imposed by the virtual machine verification (VMV) technique. Our results indicate that VMV imposes no measurable overhead, making the technique difficult to detect by malware using performance analysis [12].

1.2 Summary of Contributions

This paper makes the following contributions:

- **BotSwindler architecture:** It introduces BotSwindler, a novel, accurate, efficient, and tamper-resistant zero-day crimeware detection system. BotSwindler relies on the use of decoy injection whereby bogus information is used to bait and delude crimeware, causing it to reveal itself during the exploitation of the monitored information.
- **VMSim language:** It introduces VMSim, a new language for expressing simulated user behavior. VMSim facilitates the construction and reproduction of complex user activity, including specifying aggregate statistical behavior.
- **Virtual Machine Verification (VMV):** It introduces virtual machine verification, a low overhead approach for verifying simulation state. VMV enables robust out-of-host user action simulation through graphical state verification.
- **Real malware detection results:** It presents results to show the effectiveness of BotSwindler in detecting real malware when decoy PayPal, Gmail, and banking credentials are injected, stolen, and exploited by the attackers.
- **Statistical and information theoretic analysis:** It presents the results of a computational analysis on generated keystroke timing data to show it would be difficult to detect simulations through analysis with machine learning algorithms or entropy measurements.
- **Believability user study results:** It presents user study results that show the believability of simulations created with BotSwindler's VMSim language.
- **Performance overhead results:** It shows that BotSwindler imposes no measurable overhead, hence making itself undetectable via timing measurement methods.

2 Related Work

Deception-based information resources that have no production value other than to attract and detect adversaries are commonly known as honeypots. 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 [13], 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.* [14] 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. However,

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.* [15], developed a system to randomize generated human input to foil potential analysis techniques that may be employed by malware. The work by Holz *et al.* [1] to investigate keyloggers and dropzones, relied on executing maleware in CWSandbox [16] and automating user input with AutoIt [5]. However, it was limited to ad hoc scenarios designed for the sole purpose of detecting harvesting channels. Their approach depends on miss-configured 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, it is limited by law and skill of the attackers (*i.e.,* they can just secure their dropzone servers). In addition, relying on simulator software that resides within the host, such as AutoIt, provides attackers with a simple means to detect and avoid it. In contrast to these systems, BotSwindler is difficult to detect, automatically injects input that is 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. Egele *et al.* [17] 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 sent in offline analysis. Similarly, Yin *et al.* [18] 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 [18]; both these 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 and Rosenblum [19] 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 [20] between the two levels of abstraction represented by the VM and the underlying service or OS. Garfinkel and Rosenblum 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 [21,22,23], 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.

[5] `http://www.autoitscript.com`

3 BotSwindler Components

The BotSwindler architecture, as shown in Fig. 1, consists of two primary components including a simulator engine, VMSim, and a virtual machine verification component. 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 three sections.

3.1 VMSim

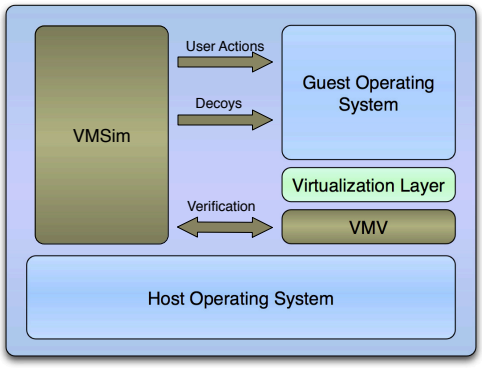

Fig. 1. BotSwindler architecture.

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:

1. the simulator process remains undetected by the malware
2. 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. Representative details of the formal language are provided in Fig. 2 (many details are omitted due to space limitations). The language provides a flexible way to

generate variable simulation behaviors and workflows, but more importantly it supports the use of *cover* and *carry* actions; carry actions result in the injection of decoys (described in Sect. 3.3), 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 Sect. 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.

```
<ActionType> ::= <WinLogin> <ActionType>
   | <CoverAction> <ActionType> | <CarryAction> <ActionType>
   | <WinLogout> | <VerifyAction> <ActionType> | e
<CoverAction> ::= <BrowserAction> <CoverAction>
   | <WordAction> <CoverAction> | <SysAction> <CoverAction>
<BrowserAction> ::= <URLRequest> <BrowserAction>
   | <OpenLink> <BrowserAction> | <Close>
<WordAction> ::= <NewDoc> <WordAction>
   | <EditDoc> <WordAction> | <Close>
<SysAction> ::= <OpenWindow> | <MaxWindow>
   | <MinWindow> | <CloseWindow>
<VerifyAction> ::= Img1 | Img2 | ... | ImgN | Unknown
<CarryAction> ::= <PayPalInject> | <GmailInject>
   | <CCInject> | <UnivInject> | <BankInject>
```

Fig. 2. VMSim language.

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 biometric models for keystroke speed, mouse speed, mouse distance, and the frequency of errors made during typing. These parameters function as controls over the language shown in Fig. 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 outputted as X protocol level data that can be replayed via the XTest extensions.

To construct biometric models for individuals, we have extended QEMU's VMM to support the recording of several features including keycodes (the ASCII code representing a key), the duration for which they are pressed, keystroke error rates, mouse movement speed, and mouse movement distance. Generative models for keystroke timing are created by first dividing the recorded data for each keycode pair into separate classes where each class is determined by the distance in standard deviations from the mean. We then calculate the distribution for each keycode sequence as the number of instances of each class. We adapt simulation keystroke timing to profiles of individual users by generating random times that are bounded by the class distribution. Similarly, for mouse movements we calculate user specific profiles for speed and distance. Recorded mouse movements are broken down into variable length vectors that represent

periods of mouse activity. We then calculate distributions for each user using these vectors. The mouse movement distributions are used as parameters for tuning the simulator actions. We note that identifying the complete set of features to model an individual is an open problem. Our selection of these features is to illustrate a feasible approach to generating statistically similar actions. In addition, these features have been useful for verifying the identify of individuals in keystroke and mouse dynamics studies [24,25]. In Sect. 4.1 we provide a statistical and information theoretic analysis of the simulated times.

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. 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.

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) [26]. 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 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, they are not sufficient 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[6].

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

[6] 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.

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 [20], 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 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 Sect. 4.3 we provide a performance analysis to aid in this consideration.

3.3 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* including online banking logins provided by a collaborating financial institution, login accounts for online servers, and web based email accounts. For the experiments in this paper, we focused on the use of decoy Gmail, PayPal credentials, and banking credentials. These were chosen because they are widely used and known to have underground economy value [1,27], making them alluring targets for crimeware, yet inexpensive for us to create. The banking logins are provided to us by a collaborating financial institution. As part of the collaboration, we receive daily reports showing the IP addresses and timestamps for all accesses to the accounts at any time.

The decoy PayPal and bank accounts have an added bonus that allows us to expose the credentials without having to be concerned about an attacker changing their

password. PayPal requires multi-factor authentication to change the passwords on an account. Yet, we do not reveal all of the attributes of an account making it difficult for an attacker to change the authentication credentials. For the banking logins, we have the ability to manage the usernames and passwords.

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, the 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. The choice as to what time interval to use and how frequently to poll presents significant tradeoffs that we analyze in Sect. 4.4.

In the case of the Gmail accounts, custom scripts access `mail.google.com` to parse the bait account pages, gathering 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 Statistical and Information Theoretic Analysis

In this section we present results from the statistical analysis of generated keystroke timing information. The goal of these experiments was to see if a machine learning algorithm (one that would be available to a malware sample to determine whether keystrokes are real or not) might be able to classify keystrokes accurately into user generated or machine generated. For these experiments, we relied on Killourhy and Maxion's benchmark data set [28]. The data set was created by having 51 subjects repeatedly type the same 10 character password, 50 times in 8 separate sessions, to create 400 samples for each user. Accurate timestamps were recorded by using an external clock. Using this publicly available real user data ensures that experiments can be repeated.

To evaluate VMSim's generated timing information, we used Weka [29] for our classification experiments. We divided the benchmark data set in half and used 200 password timing vectors from each user to train Naive Bayes and Support Vector Machine (SVM) classifiers. The remaining 200 timing vectors from each user were used as input to VMSim's generation process to generate 200 new timing vectors for each user. The same 200 samples were used for testing against the generated samples in the classification experiments. Note that we only used fields corresponding to hold times and inter-key latencies because the rest were not applicable to this work (they can also contain negative values). The normalized results of running the SVM and Naive Bayes classifiers on the generated data and real data are presented in Figs. 3 and 4, respec-

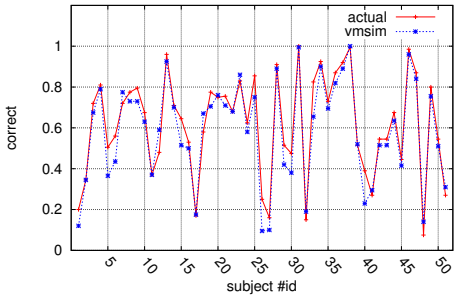

Fig. 3. SVM classification.

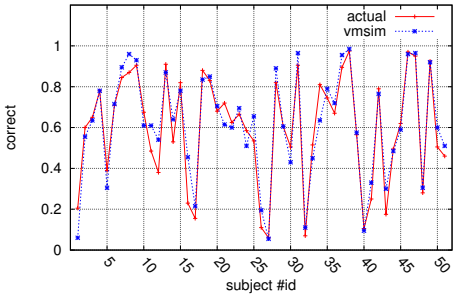

Fig. 4. Naive Bayes classification.

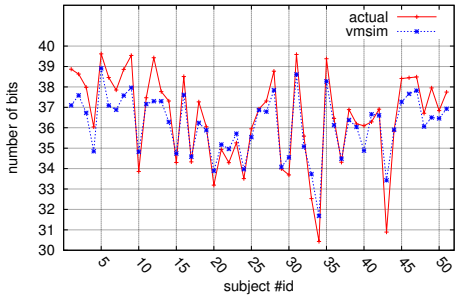

Fig. 5. Entropy of *generated* and *actual* timing data.

tively. The results are nearly identical for these two classifiers suggesting that this particular type of analysis would not be useful for an attacker attempting to distinguish the real from generated actions. In Fig. 5, we present a comparison of entropy values (the amount of information or bits required to represent the data) [30] for the actual and generated data for each of the 200 timing vectors of the 51 test subjects. The results indicate that there is no loss of information in our generation process that would be useful by an adversary that is attempting distinguish real from generated actions.

4.2 Decoy Turing Test

We now discuss the results of a Turing Test [10] to demonstrate BotSwindler's performance regarding the *humanness*, or believability, of the generated simulations. The point of this experiments is to show that adversaries resorting to manual inspection of the user activities would be sufficiently challenged. Though 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 students, 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 the 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.

Figure 6 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 7 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 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 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 answer 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.

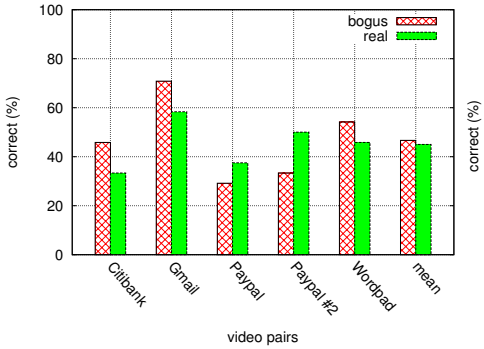

Fig. 6. Decoy Turing Test results: *real* vs. *simulated*.

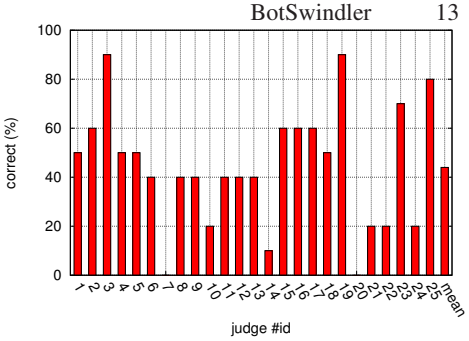

Fig. 7. Judges' overall performance.

4.3 Virtual Machine Verification Overhead

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

Table 1. Overhead of VMV with idle user.

	Min.	Max.	Avg.	STD
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.	Avg.	STD
Native OS	.50	.72	.56	.06
QEMU	.57	.96	.71	.07
QEMU w/VMV	.53	.89	.71	.06

In Table 1, we present the analysis of the overhead of QEMU[7] 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 with 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 defined by thousands of pixels).

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

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.

4.4 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 is 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. PayPal decoy false negative likelihoods.

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

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 this threshold because we experimentally determined that it exhibits no false positives. For the experiments described in Sect. 4.5, we use the 1 hour polling frequency because we believe it provides an adequate balance (the false negative rate is relatively low and the alerts are generated quickly enough).

4.5 Detecting Real Malware with Bait Exploitation

To demonstrate the efficacy of our approach, we conducted two experiments using BotSwindler against crimeware found in the wild. For the first experiment, we injected Gmail and PayPal decoys, and for second, we used decoy banking logins. The experiments relied on Zeus because it is the largest botnet in operation. Zeus is sold as a crimeware kit allowing malicious individuals to create and configure their own unique botnets. Hence, it functions as a payload dissemination framework with a large number of variants. 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 is 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 [8].

In our first experiment, we used 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 Sect. 4.4, 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. The banking logins used in the second experiment 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

[8] `http://www.offensivecomputing.net`

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 116 binaries that were tested with the goal of determining whether attackers would take and exploit the bait credentials. Over the course of five days of monitoring, we received thirteen alerts from the PayPal monitor and one Gmail alert. We ended the study after five days because the results obtained during this period were enough to convince us the system worked[9]. 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 black market. Although Gmail accounts may have value for activities such as spamming, they can be purchased by the thousands for very little cost[10] and there are inexpensive tools that can be used to create them automatically. 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 [27] lists bank accounts 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
- 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.

[9] We ended the study after 5 days, but a recent examination of the monitoring logs revealed alerts still being generated months after.

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

Hence, the number of attacks we detected is a lower bound. In addition, despite our efforts to get active binaries, many were found to be inactive, some cause the system to fail, and some have objectives other than stealing credentials.

In the second experiment, we relied on several bank accounts containing balances over $1,000 USD. In contrast with the PayPal experiments, this experiment relied on an actual bank website with authentic SSL certificates. The bank account balances were frozen so that money could not actually be withdrawn. We ran the simulator for approximately 10 minutes on 59 new binaries. Over the course of five days of monitoring, we received 3 alerts from the collaborating financial institution. The point of these experiments is to show that decoy injection can be useful tool for detecting crimeware that can be difficult to detect through traditional means. These results 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.

5 Applications of BotSwindler in an Enterprise

Beyond the detection of malware using general decoys, BotSwindler is well suited for use 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 run simulations on a user's system when it is idle (*e.g.,* during meetings, at night). 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. One possible application of BotSwindler is in deployment as an enterprise service that runs simulations 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 [31]. In either case, BotSwindler can 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,* internal URLs), but use of specialized decoys does not preclude

the use of general decoys like those detailed in Sect. 3.3. 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 [32].

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. 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. We performed a computational analysis to show

the statistical similarities of simulations to real actions conducted. To demonstrate the believability of the simulations by humans, we conducted a Turing Test that showed we could succeed in convincing humans about 46% of the time. Finally, Botwindler has been shown to be an effective and efficient automated tool to deceive and detect crimeware.

References

1. Holz, T., Engelberth, M., Freiling, F.: Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones. In: European Symposium on Research in Computer Security (ESORICS). LNCS, vol. 5789, pp. 1–18. Springer, Heidelberg, Germany (September 2009)
2. Stahlberg, M.: THE TROJAN MONEY SPINNER. In: 17^{th} Virus Bulletin International Conference (VB) (September 2007), http://www.f-secure.com/weblog/archives/VB2007_TheTrojanMoneySpinner.pdf
3. Researcher Uncovers Massive, Sophisticated Trojan Targeting Top Businesses. Darkreading (July 2009), http://www.darkreading.com/database_security/security/privacy/showArticle.jhtml?articleID=218800077
4. Higgins, K.J.: Up To 9 Percent Of Machines In An Enterprise Are Bot-Infected. Darkreading (September 2009), http://www.darkreading.com/insiderthreat/security/client/showArticle.jhtml?articleID=220200118
5. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the Infeasibility of Modeling Polymorphic Shellcode. In: 14^{th} ACM Conference on Computer and Communications Security (CCS). pp. 541–551. ACM, New York, NY, USA (2007)
6. Blog, T.S.S.: ZeuS Tracker, https://zeustracker.abuse.ch/index.php
7. Messmer, E.: America's 10 most wanted botnets. Network World (July 2009), http://www.networkworld.com/news/2009/072209-botnets.html
8. Measuring the in-the-wild effectiveness of Antivirus against Zeus. Technical report, Trusteer (September 2009), http://www.trusteer.com/files/Zeus_and_Antivirus.pdf
9. Ilett, D.: Trojan attacks Microsoft's anti-spyware (February 2005), http://news.cnet.com/Trojan-attacks-Microsofts-anti-spyware/2100-7349_3-5569429.html
10. Turing, A.M.: Computing Machinery and Intelligence. Mind, New Series 59(236), 433–460 (October 1950)
11. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: USENIX Annual Technical Conference. pp. 41–46. USENIX Association, Berkeley, CA, USA (April 2005)
12. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is Not Transparency: VMM Detection Myths and Realities. In: 11^{th} Workshop on Hot Topics in Operating System (HotOS). USENIX Association, Berkeley, CA, USA (May 2007)
13. Spitzner, L.: Honeytokens: The Other Honeypot (July 2003), http://www.securityfocus.com/infocus/1713
14. Borders, K., Zhao, X., Prakash, A.: Siren: Catching Evasive Malware. In: IEEE Symposium on Security and Privacy (S&P). pp. 78–85. IEEE Computer Society, Washington, DC, USA (May 2006)
15. Chandrasekaran, M., Vidyaraman, S., Upadhyaya, S.: SpyCon: Emulating User Activities to Detect Evasive Spyware. In: Performance, Computing, and Communications Conference (IPCCC). pp. 502–509. IEEE Computer Society, Los Alamitos, CA, USA (May 2007)

16. Willems, C., Holz, T., Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox. In: IEEE Symposium on Security and Privacy (S&P). pp. 32–39. IEEE Computer Society, Washington, DC, USA (March 2007)
17. Egele, M., Kruegel, C., Kirda, E., Yin, H., Song, D.: Dynamic Spyware Analysis. In: USENIX Annual Technical Conference. pp. 233–246. USENIX Association, Berkeley, CA, USA (June 2007)
18. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In: 14^{th} ACM Conference on Computer and Communications Security (CCS). pp. 116–127. ACM, New York, NY, USA (2007)
19. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: 10^{th} Annual Network and Distributed System Security Symposium (NDSS). Internet Society, Reston, VA, USA (February 2003)
20. Chen, P.M., Noble, B.D.: When Virtual Is Better Than Real. In: 8^{th} Workshop on Hot Topics in Operating System (HotOS). pp. 133–138. IEEE Computer Society, Washington, DC, USA (May 2001)
21. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: Tracking Processes in a Virtual Machine Environment. In: USENIX Annual Technical Conference. pp. 1–14. USENIX Association, Berkeley, MA, USA (March 2006)
22. Jiang, X., Wang, X.: "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. In: 10^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). LNCS, vol. 4637, pp. 198–218. Springer, Heidelberg, Germany (September 2007)
23. Srivastava, A., Giffin, J.: Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In: 11^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). LNCS, vol. 5230, pp. 39–58. Springer, Heidelberg, Germany (2008)
24. Monrose, F., Rubin, A.: Authentication via Keystroke Dynamics. In: 4^{th} ACM Conference on Computer and Communications Security (CCS). ACM (April 1997)
25. Ahmed, A.A.E., Traore, I.: A New Biometric Technology Based on Mouse Dynamics. IEEE Transactions on Dependable and Secure Computing (TDSC) 4(3), 165–179 (2007)
26. The XFree86 Project: XVFB(1), `http://www.xfree86.org/4.0.1/Xvfb.1.html`
27. Symantec: Trends for July - December '07. White paper (April 2008)
28. Killourhy, K.S., Maxion, R.A.: Comparing Anomaly Detectors for Keystroke Dynamics. In: 39^{th} Annual International Conference on Dependable Systems and Networks (DSN). IEEE Computer Society Press, Los Alamitos, CA, USA (June-July 2009)
29. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA Data Mining Software: An Update. ACM SIGKDD Explorations Newsletter 11(1), 10–18 (2009)
30. Lee, W., Xiang, D.: Information-Theoretic Measures for Anomaly Detection. In: IEEE Symposium on Security and Privacy (S&P). pp. 130–143. IEEE Computer Society, Washington, DC, USA (2001)
31. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: High Availability via Asynchronous Virtual Machine Replication. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 161–174. USENIX Association, Berkeley, CA, USA (April 2008)
32. Bond, M., Danezis, G.: A Pact with the Devil. In: New Security Paradigms Workshop (NSPW). pp. 77–82. ACM, New York, NY, USA (September 2006)