A Layered Architecture for Detecting Malicious Behaviors

Lorenzo Martignoni*, Elizabeth Stinson†, Matt Fredrikson‡, Somesh Jha‡, John C. Mitchell†

*Università degli Studi di Milano, †Stanford University, ‡University of Wisconsin

Abstract. We address the *semantic gap* problem in behavioral monitoring by using hierarchical behavior graphs to infer high-level behaviors from myriad low-level events. Our experimental system traces the execution of a process, performing data-flow analysis to identify meaningful actions such as "proxying", "keystroke logging", "data leaking", and "downloading and executing a program" from complex combinations of rudimentary system calls. To preemptively address evasive malware behavior, our specifications are carefully crafted to detect alternative sequences of events that achieve the same high-level goal. We tested eleven benign programs, variants from seven malicious bot families, four trojans, and three mass-mailing worms and found that we were able to thoroughly identify high-level behaviors across this diverse code base. Moreover, we effectively distinguished malicious execution of high-level behaviors from benign by identifying remotely-initiated actions.
Keywords: Dynamic, Semantic Gap, Malware, Behavior, Data-Flow

1 Introduction

In the first half of 2007, Symantec observed more than five million active, distinct bot-infected computers [1]. Botnets are used to perform nefarious tasks, such as: keystroke logging, spyware installation, denial-of-service (DoS) attacks, hosting phishing web sites or command-and-control servers, spamming, click fraud, and license key theft [2,3,4,5,6,7]. Malicious bots are generally installed as applications on an infected (Windows) host and have approximately the same range of control over the compromised host as its rightful owner. A botmaster can flexibly leverage this platform in real-time by issuing commands to his botnet. Several characteristics typical of botnets increase the difficulty of robust network-based detection; in particular, bots may: exhibit high IP diversity, have high-speed, always-on connections, and communicate over encrypted channels. Since a botmaster controls both the bots and the command-and-control infrastructure, these can be arbitrarily designed to evade network-based detection measures.

It is widely recognized that malware defenders operate at a fundamental disadvantage: malware producers can generate malware variants by simple measures such as packing transformations (encryption and/or compression) and may evade existing AV signatures by systematic means [8]. For the signature purveyors, moreover, analyzing a novel malware instance and creating a detection

signature requires substantially greater effort than that required by evasion. The source of this asymmetry is the signature scanners' emphasis on malware's infinitely mutable syntax, rather than on the actions taken by malware. As a result, even the most effective signature-scanners fail to detect more than 30% of malware seen in the wild [9,10]. Therefore, it is essential to develop effective methods that identify the behaviors that make malware useful to their installers.

1.1 Our Approach

We propose, develop, and evaluate a behavior-based approach that targets the high-level actions that financially motivate malware distribution. For bots, these actions include "proxying", "keystroke logging", "data leaking", and "program download and execute." We build representations of these high-level actions hierarchically, taking care to identify only the essential components of each action. The lowest level event in our behavior specifications are system call invocations. Since any specific operating system kernel exports a finite set of operations, we can expect to be able to enumerate all possible ways to interface with that kernel in order to achieve a certain effect (e.g., send data over the network). Since there are a finite number of ways to achieve each high-level action, we can expect to create representations that encode all such ways. Consequently, we can hope to correct the asymmetry present in syntax-based approaches to malware detection.

In this paper we propose and evaluate a behavior-based malware detector that takes as input the behavior specifications introduced above and an event stream provided by our system-wide emulator (Qemu), which monitors process execution. A system-wide emulator provides a rich source of information but infers no higher-level effects or semantics from the observed events. This disconnect between a voluminous stream of low-level events and any understanding of the aggregate effect of those events [13] is referred to as the *semantic gap*. We address the semantic-gap by decomposing the problem of specifying high-level behaviors into layers, making our specifications composable, configurable, less error-prone, and easy to update. Our system compares a monitored process's event stream to behavior specifications and generates an event when there is a match. This generated event may then be used in the specification of a higher-layer behavior.

Fig. 1 provides a subset of the hierarchy of events used to specify our sample target high-level behavior: downloading and executing a program, which is used in malware distribution. Events are represented via rectangles, with directed edges between them indicating dependencies; e.g., the `tcp_client` event depends upon the `sync_tcp_client` and `async_tcp_client` events. At the lowest layer of the hierarchy, $L0$, we identify successful system call invocations. Each $L1$ event aggregates $L0$ events that have a common side effect, as is the case with the $L1$ `net_recv` event which is generated whenever any of the $L0$ events `recv`, `recvfrom`, or `read` occur. Consequently, we can represent "all ways to receive data over the network" using a single event. Events at layers $L2$ and higher identify correlated sequences of lower-layer events that have some aggregate, composite effect; e.g., `sync_tcp_client` identifies when a synchronous TCP socket has been successfully created, bound, and connected upon.

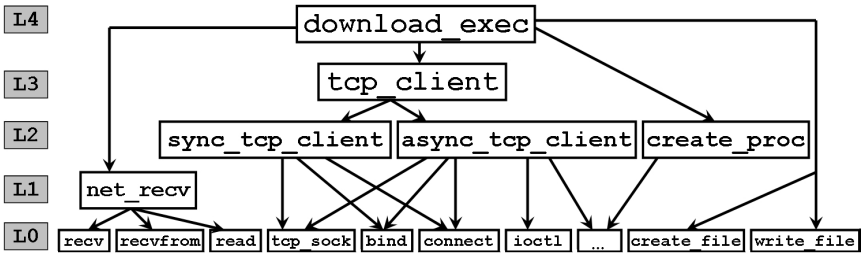

Fig. 1. A subset of the hierarchy of events used to specify `download_exec`

Correlating low-level events generally entails specifying constraints on those events' arguments. In some cases, we need to specify that data used in one event is dependent upon data used in another event. Consequently, Qemu performs instruction-level data-flow analysis (tainting) and exports two related operations: `set_tainted` designates a memory region tainted with a particular label; and `tainted` determines whether a memory region contains data received from a particular source (as identified by its taint label). An important class of tainted data is that which is derived from local user input; this *clean data* is used to differentiate locally-initiated from remotely-initiated actions. Both `tainted` and `set_tainted` can be used in our behavior specifications; consequently, we can designate novel taint sources without changing our system implementation.

Commonly, malware variants are generated by: (I) applying packing transformations (compression and/or encryption) to a binary, (II) applying instruction-level obfuscation such as nop insertion as in [15], (III) applying source-level obfuscations as in [8], (IV) using a bot-development kit, which provides a point-and-click interface for specifying bot configuration details and builds the requested bot, or (V) directly modifying the source of an existing bot to add novel functionality and/or commands. Our behavioral graphs are insensitive to the type of changes entailed in (I) – (III) since the semantics of a malware's behavior are unchanged. The changes in (IV) also do not affect the bot's implementation of a particular command, only whether that command is available or not. Moreover, since we identify the fundamental system-call signatures for high-level behaviors, even changing the implementation as in (V) without changing the overall semantic effect would not suffice to evade detection.

The contributions of this paper include:

- A behavior-specification language (described in Section 2) that can be used to describe novel, semantically meaningful behaviors.
- A detector (described in Section 3) that identifies when a process performs a specified high-level action, regardless of the process's source-code implementation of the action.
- Our evaluation (described in Section 4) demonstrates that our detector can distinguish malicious execution of high-level behaviors from benign.

2 Representing High-Level Behaviors

In this section, we define our behavior graphs, each of which describes a correlated sequence of events that has some particular semantic effect (such as `connect` or `tcp_client`). The graph for a behavior B identifies only the fundamental component events required to achieve B and constrains these events as minimally as possible. Matching a behavior graph generates an event that can be used as a component within another graph; e.g., matching the `tcp_client` graph generates the `tcp_client` event, which can be used in specifying other behaviors, such as `tcp_proxy`. In this way, we compose graphs hierarchically, which enables us to recognize complex process behaviors.

2.1 Behavior Graphs

A *behavior graph* is a directed graph of a form that is adapted from and extends AND/OR graphs [26]. A behavior graph can be thought of as a template; multiple different sequences of events can be bound to this template subject to the edge constraints; binding and matching are described more precisely in sect. 2.1. Fig. 2 contains the behavior graph for our running example, `download_exec`.

Each behavior graph has a *start point*, drawn as a single point at the top of the graph, internal nodes, and an *output event*, which is represented via a shaded rectangle. Each internal node in the graph has a name, such as `create_file`, and formal parameters, such as `fh0`, `fname`, `fname_len`, as in fig. 2. Together, a node's name and formal parameters characterize a set of events, namely those events whose name is the same as the node's name. Whereas internal nodes represent input events needed in order to continue graph traversal, the special output event represents an action taken by our system; hence no additional input is required to traverse an edge from a penultimate node to the output event. For example, any sequence of events that matches the graph in fig. 2 up to the `create_proc` node will also reach the `download_exec` node and generate a `download_exec` event. When we match a graph and generate an output event e, the parameters for e are obtained from e's constituent events; e.g., the socket descriptor, `rem_ip`, and `rem_port` arguments for the `download_exec` output event in fig. 2 are obtained from its constituent `tcp_client` event.

AND-edge sets and OR-edge sets: A behavior graph may have AND-edges and OR-edges. OR-edges are drawn simply as directed edges, while AND-edges are drawn using a horizontal line to form an AND-edge set. In fig. 2, a sequence of events can reach the `net_recv` node by either of the two OR-edges leading into this node. In contrast, the AND-edges into `write_file` indicate that both `net_recv` *and* `create_file` are required to match this portion of the graph. If a node's in-edge set contains AND-edges and OR-edges, this expresses an OR of ANDs. We use AND-edge sets to identify events which can occur in any relative order but must all precede some other event.

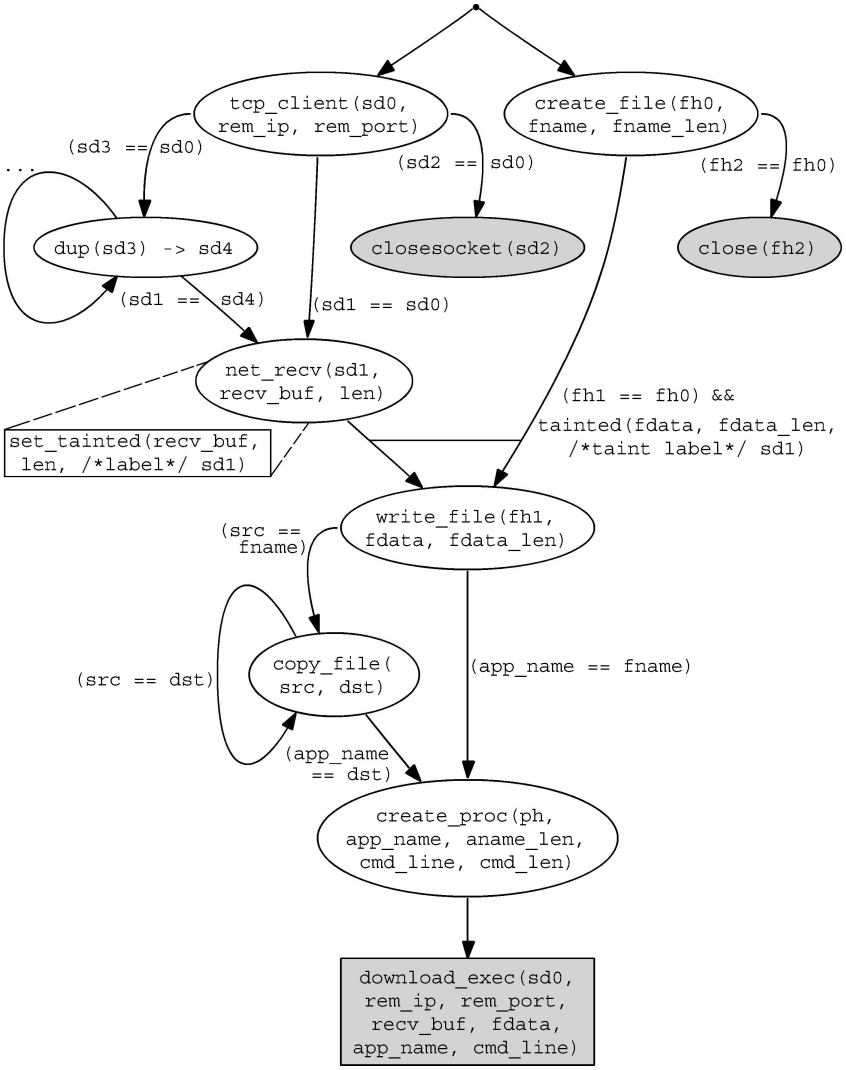

Fig. 2. AND/OR graph for downloading a file and executing it

Annihilator and Replicator Nodes: We correlate events by specifying predicates on their parameters; thus, it's important to know when a parameter has been destroyed or duplicated. *Annihilator nodes* are used to represent that certain events destroy objects; e.g., calling `closesocket` on a socket descriptor `sd` releases `sd`, rendering it unable to be used in subsequent events. Annihilator nodes are represented via shaded ellipses, as with the `close(fh)` node in fig. 2. The edge from `create_file(fh0, ...)` to `close(fh2)` imposes the con-

dition that `close` cannot be called on the newly-created file handle prior to `write_file(...)` being called on that same handle. Certain events, which we refer to as *replicators*, duplicate objects, such as socket descriptors or files. For example, calling `dup` on a socket descriptor or file handle creates a copy of the passed object; any operation that could be called on the original object can equivalently be called on the duplicate. We represent this via replicator nodes as with `dup(...)` and `copy_file(...)` in fig. 2. Since a replicator operation can be called repeatedly on its own output, replicator nodes contain a self-loop. For succinctness, some annihilators and replicators are excluded from the figures.

Edge Predicates: A directed edge can be labeled with predicates that must be satisfied to traverse the edge. Our system provides three predicate types: argument-value comparison, regular expression matching, and the `tainted` predicate. In *argument-value comparison*, we can apply any of the standard relational operators ($=$, $\neq$, $>$, $<$) to compare an argument value to a constant or to another argument. Fig. 2 contains several argument-value predicates, such as `(sd1 == sd0)` between the `tcp_client` and `net_recv` events. We can also specify that a string or buffer argument value must match a constant *regular expression* as used in the `send_email` behavior graph to identify transmission of SMTP protocol messages (e.g., `MAIL FROM`). The `tainted` predicate identifies data-flow relationships that must hold; we can require that an argument be derived from a general taint source (e.g., the network) or a specific taint source (e.g., a particular network connection). Fig. 2 includes a data-flow dependency; namely, the data written to the newly-created file (`fdata`) must be derived from data received over the specified network connection as indicated by its taint label (`sd1`).

`On-reach` **Actions:** Our monitoring system can perform an action in response to reaching a given node. An `on_reach` action is represented in the graph via a rectangle – connected to its corresponding node via dashed lines – containing the action to be performed. Fig. 2 shows that, upon reaching the `net_recv` node, the received buffer will be marked tainted with the taint label `sd1`.

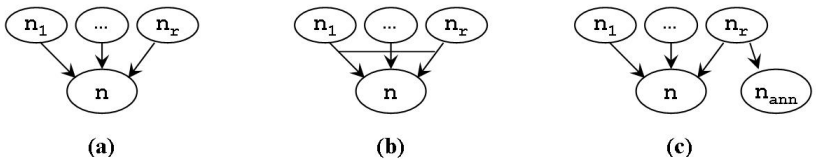

Fig. 3. Graph G with (a) OR-ed edges, (b) AND-ed edges, (c) an annihilator

Summary: A behavior graph defines a set of event sequences that match the graph, and may specify one or more on-reach events that will be generated when events match the graph in certain ways. These properties may be captured precisely in a rigorous definitions that allow us to prove properties of various algorithms. For example, a sequence $E = e_1, e_2, \ldots, e_k$ of events matches a behavior graph G if there is a function f from a subset of the nodes of G to events in E and a substitution S on variables that appear in formal parameters of the graph that satisfy the following conditions:

1. If there is an OR-edge set into a node n with $f(n) \in E$, as illustrated in fig. 3(a), then $\exists i.\, f(n_i) \in E$.
2. If there is an AND-edge set into a matched node n with $f(n) \in E$, as illustrated in fig. 3(b), then $\forall i.\, f(n_i) \in E$.
3. If there are matched nodes $n_r \rightarrow n$ with an annihilator node as illustrated in fig. 3(c), then $\nexists$ event $e \in E$ with $f(n_r) < e < f(n)$ and e matches $ev(n_{ann})$ by any $S' \supseteq S$.
4. If predicate P appears on an edge between nodes n and n' with $f(n) \in E$ and $f(n') \in E$, then the substitution instance $S(P)$ of P is true.

2.2 Behavior-Specification Language

A major contribution of our work is our behavior-specification language and monitoring system. Together, these can be used to specify then identify novel semantically-meaningful behaviors. The substrate consists of the graphs at each layer. Each of the behaviors specified by these graphs is a primitive that can be used in defining additional behaviors. Table 1 contains some primitives from our resulting behavior-specification language. We can describe "log keystrokes then send them in an email" using two of these primitives (`keylogging` and `send_email`) and correlating their arguments in a particular way, which illustrates the powerful, high-level expressiveness of our language.

2.3 Graph Construction

We developed our graphs manually and iteratively through domain knowledge and analysis of tens of gigabytes of execution traces, obtained from multiple runs

Table 1. Some primitives in our resulting behavior-specification language

Event	*Arguments*
tcp_client	`sd`, `loc_ip`, `loc_port`, `rem_ip`, `rem_port`
tcp_server	`sd`, `loc_ip`, `loc_port`, `cli_ip`, `cli_port`
net_send	`sd`, `buf`, `buf_len`
net_recv	`sd`, `buf`, `buf_len`
send_email	`sd`, `targ_ip`, `from_addr`, `to_addr`, `data`
keylogging	`data`, `data_len`

of (I) around fifteen standard applications (including Googletalk, Filezilla, Firefox, putty, mIRC, Internet Explorer, Outlook, Thunderbird, SecureFX, Windows Media Player, SecureCRT, Unreal IRCd, Apple Software Update, Quicktime, etc.), (II) over one hundred specially-crafted programs, and (III) several malicious programs. We present our evaluation of these graphs' coverage in sect. 4.2.

Constructing $L0$ Graphs Recall that $L0$ graphs represent successful system call invocations. The challenges here are as follows, (I) Windows implements the sockets API through a single system call, `NtDeviceIoControlFile`, (II) we do not have source access to the target OS, and (III) we need to be able to differentiate invocations of `listen` from invocations of `accept` and so on. We rely on analysis of process execution traces in order to identify commonalities (in arguments) across all invocations of a sockets function s_1 but which are not present in any invocations of all other sockets functions, s_2, s_3, ..., s_k. These commonalities are the basis of our $L0$ behavior graphs. The coverage of any graph then relies upon the diversity of process traces. Our process traces delineate entry to and return from each sockets function and identify all system calls invoked therein, including each system call's arguments and return value.

For some functions, such as `socket`, we crafted a suite of programs that invoked the function using all possible combinations of valid arguments. The execution of other sockets functions, however, is stateful in that it depends directly upon previous actions performed on the same socket descriptor; e.g. `recv`. Hence, it is not enough to provide different argument combinations to `recv`, we must also precede the invocation of `recv` with different combinations of particular sockets functions, such as `socket`, `bind`, `listen`, `connect`, and so on.

Pending System Call Invocations: A system call *sc* may not immediately return success or failure but rather return `STATUS_PENDING`; `NtWaitForSingleObject` is subsequently invoked on *sc*'s associated event object. We encode this path in our $L0$ graphs so as to identify *eventually successful* system call invocations.

Constructing $L1$ Graphs Recall that $L1$ graphs aggregate $L0$ events that have a similar side effect. Since the system call interface is finite, we can enumerate the "relevant" effects of each system call and construct an $L1$ graph for each such effect, where by "relevant" we mean "of interest". In our case, there were two effects that required $L1$ graphs: `net_send` and `net_recv`. These were immediately identifiable through domain knowledge. Note that aggregation graphs can exist at higher layers as well; e.g., we use an $L3$ graph to aggregate `async_tcp_client` and `sync_tcp_client` so that we may identify generally `any tcp_client`.

Constructing $L2$ Graphs The graphs at $L2$ identify correlated sequences of lower-layer events which have some aggregate, composite effect, e.g. `create_write_file`. For each target $L2$ behavior, we identify the events essential to that behavior and any dependencies between those constituent events. This identification comes through (I) domain knowledge, such as encoding that in order

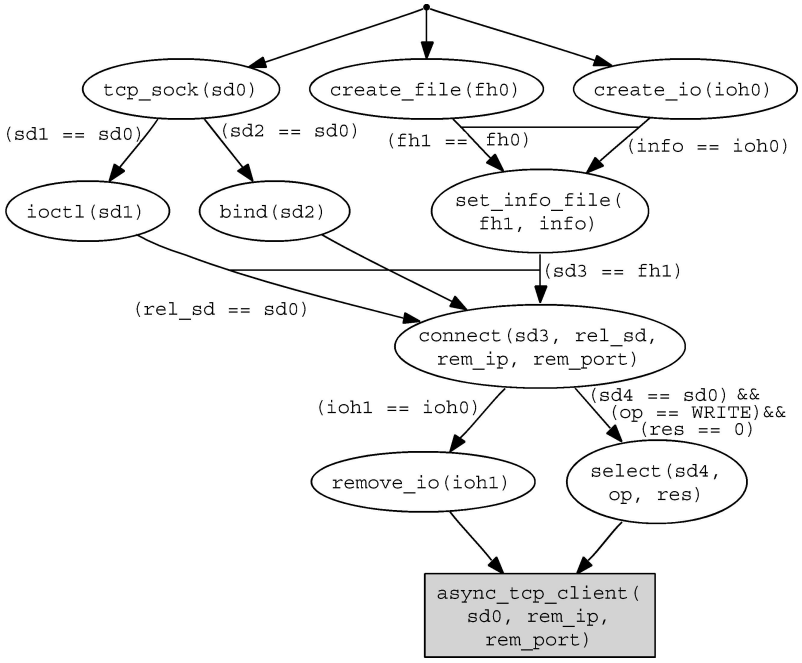

Fig. 4. *L*2 AND/OR graph for an asynchronous TCP client

to `connect` or `listen` on a socket, that socket must first have been (explicitly or implicitly) bound, and (II) analysis of process traces, as used to construct the graphs for asynchronous network interaction. Windows exports a rich API for performing asynchronous network interaction, including the standard polling model using `select` on a socket as well as event-based approaches, such as via `WSAEventSelect` and `WSAAsyncSelect`. We are able to represent all of these through a single asynchronous TCP client graph as in fig. 4. This graph was built by examining process traces of existing applications which use the Windows asynchronous API as well as augmenting this analysis with traces of specially-crafted programs designed to capture more execution diversity.

3 System Implementation

Figure 5 depicts the architecture of our system, which has two main components: a system-wide emulator (Qemu) and a behavior matcher. Qemu emulates and traces the execution of analyzed programs in an isolated virtual environment. We use a hybrid emulated/virtualized approach, where the execution of the process under analysis is emulated while the execution of all other processes in the system is virtualized using KQemu [38]. The behavior matcher obtains information about process events from the emulator and attempts to match

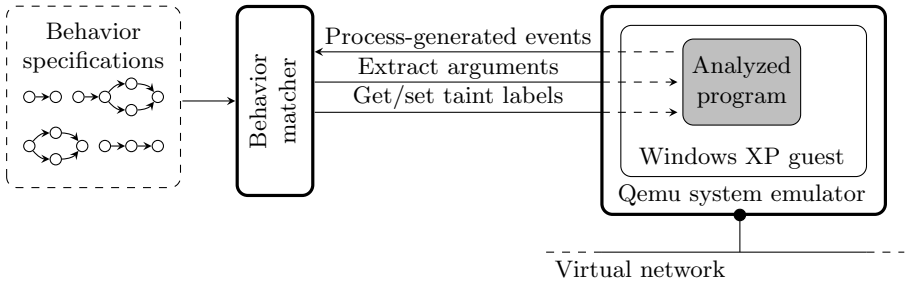

Fig. 5. Architecture of the system.

this input event stream to the behavior graphs. The behavior matcher operates independently of the particular monitoring technique and, as such, could be used in concert with, e.g., a process emulator. We use system-wide rather than process emulation for reasons relating to ease of experiment execution and cleanup. In particular, Qemu offers built-in support for rollback of system state and enables easy isolation of the monitored process from the external world.

3.1 System Emulator

Our system-wide emulator extends Qemu [39], an open-source emulator based on dynamic binary translation, by adding guest-OS-aware virtual machine introspection and taint analysis capabilities [25]. Guest-OS awareness is essential as we must be able to determine: which system call was invoked, which process invoked it, and the format of the system call's argument buffers. Our system currently emulates the IA-32 architecture and supports Microsoft Windows XP.

Process-generated Events: We instrument the code executed in the emulator by hooking the `sysenter` and `sysexit` instructions, which identify, respectively, invocation of and return from system calls. The instrumentation causes the emulator to provide this event stream to the behavior matcher in real-time.

Taint Analysis: The code executed in the emulator is also instrumented to perform taint analysis. In order to propagate taint labels through data dependencies, we extend the semantics of instructions that assign a value to a register or memory location, excluding floating point operations. We set the label of an assignment instruction's destination operand to be the union of the source operands' labels. Instructions instrumented in this manner are referred to as *taint propagation instructions*. To reduce overhead, we perform taint analysis on user-space code only. Our system also includes support for custom taint propagation rules over operations at a higher level than machine code instructions. In particular, we use this support to propagate taint across system calls that participate in hostname resolution; we assign the labels from the input hostname buffer to the location storing the resolved IP address.

Local User Input Tracking: Our local user input tracking module is designed for Win32 GUI applications, which receive messages indicating keyboard or mouse input events. The receiving application invokes its handler for the input event via a call to `DispatchMessage`. Mouse input messages do not provide the data value associated with the event; hence, identifying this data is a challenge. We address this by entering *clean mode* whenever the monitored process is handling receipt of a mouse click or keystroke; we define that period as starting with select invocations of `DispatchMessage` and ending with the corresponding returns. During clean mode, all taint propagation instructions unconditionally set the labels of their destination operands to be the special `clean` label. We present evaluation details related to user input tracking in sect. 4.6.

3.2 Behavior Matcher

At startup, the behavior matcher loads the provided set of behavior graphs. The matcher maintains some state for each graph, including the graph's current set of active nodes. A node n_{act} in graph G is *active* when we have received some event sequence $<ev_1, ev_2, ..., ev_k>$ which causes us to transition from the `start` state of G to n_{act}. There may be multiple event sequences corresponding to any particular active node; these event sequences (including each event's actual parameters) are also part of a graph's state. For brevity, certain details of the matching algorithm are omitted.

The behavior matcher is notified in real-time by the emulator every time the monitored process invokes or returns from a system call. Given a new event e with name $name_e$, for each behavior graph, the matcher: (I) checks whether there is a transition from an active node to a node n_{new} whose name is the same as $name_e$; if not, discard e, (II) extracts e's actual parameters and binds them to n_{new}'s formal parameters; (III) evaluates the predicates on (n_{act},n_{new}); if they do not hold, discard e; (IV) if there is an `on-reach` action associated with n_{new}, then execute it; (V) if there is an edge from n_{new} to this graph's output event, the matcher generates the appropriate synthetic event.

4 Evaluation

This section provides the results of testing our dynamic specification-driven system monitor on seven malicious and eleven benign applications. After describing the experimental setup, we provide results demonstrating our ability to fill the semantic gap. Additionally, in testing the bots and benign applications against seven behavior graphs (referred to as *malspecs*) corresponding to bots' most threatening behaviors, there were no false negatives and seven false positives.

4.1 Experimental Setup

We performed our evaluation of the system in the environment depicted in Fig. 5. The evaluation framework consists of a victim Qemu virtual machine VM_{vict},

Table 2. Actions over which benign programs were exercised

Application	Interaction
ftp.exe, FTP Wanderer	Connect to server, authenticate, get a file, get multiple files
Internet Explorer	Access google.com, perform FTP access, download and execute a program.
Outlook Express	Download and read email containing an external image, reply to email, download and execute an attachment.
PuTTy	Connect and authenticate with server, send commands, use as SSH tunnel.
WinSCP, pSCP	Copy a file from server to client (and vice versa) using wildcards, download and execute a program.
SDK Installer	Download and install debugging tools from Microsoft server.
mIRC	Chat on a typical channel, DCC send, DCC get.
Google Talk	Chat, start a voice call, attempt a file transfer.
EasyProxy	Start proxy, route HTTP traffic.

which is connected to a second virtual machine VM_{gway}, which is acting as a network gateway. On VM_{vict}, the system-wide emulator monitors the target malicious or benign process. The purpose of VM_{gway} is three-fold: it isolates the emulator from the external network to prevent further infection; it provides a realistic network environment for the execution of network-aware malware; and it hosts the command-and-control (C&C) server used to direct bots' activities.

4.2 Graph Validation

To determine whether our behavior graphs adequately cover semantically-equivalent but programmatically-different execution paths, we ran a diverse suite of applications within our monitoring framework and performed matching against a set of behavior graphs corresponding to generally innocuous actions. The column headings in Table 3 identify the tested behavior graphs. We drove each application's execution via performing the actions described in Table 2. Moreover, during process execution, we performed manual analysis of network traffic and OS state in order to obtain "ground truth" about a process's actions. In this way, we were able to determine which behavior specifications any particular process should match at any point in time. Table 3 shows the output of our behavior matcher on each application and for each behavior graph. In all instances, the behavior matcher's output comported with ground truth, demonstrating that our graphs identify the fundamental components of the tested behaviors. Recall that graphs at $L2$ and higher compose lower-layer graphs. Hence, our evaluation was performed over more than forty distinct graphs.

4.3 Specifications of Malicious Behavior

The malicious behavior specifications used in our evaluation (malspecs) reflect the targeted class of malware: bots. We targeted bots because their diverse range

Table 3. Graph validation results. Blank entries indicate that the software did not perform the tested behavior.

	TCP Client	TCP Server	Net Send	Net Recv	Create Proc	Dwnld File	Dwnld & Exec	Send Email	TCP Proxy
ftp.exe	✓	✓	✓	✓		✓			
Internet Explorer	✓		✓	✓	✓	✓	✓		
Outlook Express	✓		✓	✓	✓	✓	✓	✓	
PuTTy	✓	✓	✓	✓		✓			
pSCP	✓		✓	✓		✓			
WinSCP	✓		✓	✓	✓	✓	✓		
FTP Wanderer	✓	✓	✓	✓	✓	✓	✓		
SDK Installer	✓		✓	✓		✓			
mIRC	✓	✓	✓	✓		✓			
Google Talk	✓		✓	✓		✓			
Easy Proxy	✓	✓	✓	✓					✓

of behaviors encompasses the full range of behaviors performed by some other types of malware. Our malspecs (described briefly in Table 4) correspond to the most alarming threats posed by bots [2,3,4,5,6,7], including: malware install (M1, M2), spamming (M3), DoS attacks (M4), proxying (M5), and identity theft (M6, M7). Since bots act at the behest of a remote entity (the botmaster), we describe their actions as *remotely-initiated* (RI), which occurs when the values used to perform an action depend on data received over the network [16].

4.4 Malware Results

We evaluated our system against seven malicious bots: rbot, Agobot, DSNX, Spybot, gSys, rxbot, and SDBot. When run in VM_{vict}, the bot connected to its C&C server (hosted in VM_{gway}), received a series of commands, and executed each. Table 5 shows the malspecs matched by each bot. From this, two conclusions can be drawn: first, we can detect when a process performs a high-level, semantically meaningful action, such as **Remotely-Initiated Net Download** (M2); and secondly, a single malspec can be used to identify a malicious behavior in a variety of bots. In one case, a command fed to a bot caused the bot to crash; consequently, we don't have results of executing the `email` command on rBot, which we expected would match **RI Send Email** malspec (M3).

4.5 Benign Application Results

To determine whether our malspecs sufficiently encode the difference between malicious behavior and benign, we tested eleven benign applications against these malspecs. We chose benign applications and actions over which to drive

Table 4. Malspecs used for evaluation. Recall that "RI" stands for remotely-initiated. Use of "tainted" in the below refers to data received over the network.

	Name	Description
M1	RI Create and Execute File	A file with a tainted name is created, tainted data is written to the file, and a process is created from the file.
M2	RI Net Download	A connection to a tainted address or port is created, a file with a tainted name is created, and tainted data is written to the file.
M3	RI Send Email	A sequence of messages is matched using regular expressions, and found to correspond to an SMTP message sent to a tainted email address.
M4	RI Sendto	A UDP packet is sent to a tainted port or address.
M5	RI TCP Proxy	An application binds to a tainted port number, connects to a tainted address, and relays information from the tainted port to the tainted address.
M6	Keylogging	An application captures keystrokes destined for another process.
M7	Data Leak	An application sends data from either the filesystem or the registry over a network connection.

Table 5. Results on malicious bots. Blank entries denote behaviors not matched because the bot did not implement them; † entries denote behaviors which, when exercised, caused the bot to crash.

	M1	M2	M3	M4	M5	M6	M7
rBot	✓	✓	†	✓	✓		✓
Agobot	✓	✓	✓	✓	✓		✓
DSNX	✓	✓			✓	✓	✓
SpyBot	✓				✓	✓	✓
gSys	✓	✓		✓	✓		✓
rxBot	✓	✓	✓	✓	✓		✓
SDBot	✓	✓		✓	✓		✓

each application by favoring those with the greatest perceived likelihood of triggering a match on at least one malspec. Due to the black-box nature of many Win32 applications, this selection process is imperfect. Since we favored network-intensive applications and since our malspecs define remotely-initiated actions as those which use network-supplied parameters, we expect some false positives.

Table 6 provides the results of evaluating each of our benign programs against the set of malspecs. We ran each program under two scenarios: first, with user-input tracking disabled, which corresponds to the $\overline{\text{UI}}$ column; and second, with user-input tracking enabled, which correspond to the UI column. What this means is that, e.g., GoogleTalk matched M2, M4, and M7 when we performed no user input tracking and only matched M7 when this tracking was enabled. We note that, in general, we are better able to distinguish malicious from benign when we take local user input into consideration in the manner described in 3.1.

Table 6. Results on benign applications. "$\overline{\text{UI}}$" refers to an experiment in which user input tracking was not used, and "UI" to one with it enabled.

	M1		M2		M3		M4		M5		M6		M7	
	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI
ftp.exe													✓	✓
FTP Wanderer			✓										✓	
Internet Explorer	✓		✓								✓	✓		
Outlook Express	✓		✓										✓	✓
PuTTy													✓	✓
pSCP													✓	✓
WinSCP	✓												✓	✓
SDK Installer			✓											
mIRC			✓										✓	
Google Talk			✓				✓						✓	✓
EasyProxy														

The malspec matched by most benign programs (regardless of whether user input was taken into consideration) is **Leak** (M7). **Leak** identifies when data read from a file or the registry is subsequently sent on the network. This manifests in malicious applications when sensitive user data or product keys are transmitted to the botmaster. The deficiency of this malspec is its coarse granularity; i.e., reading data from *any* file on the system and sending any portion of that causes a match. In actuality, we would prefer to encode that, when an application reads data that *does not belong* to that application, this is considered a breach. So, in a sense, a more finely-tuned **Leak** malspec would retrofit fine-grained access control for applications on Windows systems, enabling application X to read from files and registry keys belonging to X. As proof of concept, we tuned the **Leak** malspec to exclude cookie files and certain registry keys belonging Internet Explorer (IE), which explains why IE does not match M7.

4.6 Tracking Local User Input

Since our benign results make clear the importance of identifying and tracking data which is dependent upon local user input, it is important to understand how often the system is cleaning data in response to local user input (as described in 3.1). If it is the case that our system is in "clean mode" the vast majority of the time, one might question the validity of our distinction between malicious and benign. We identified the number of instructions executed by a benign process over its lifetime as well as the number of instructions executed by that process while it was in *clean* mode. The percent of instructions executed in clean mode for three representative applications was: mIRC, 1%; Outlook, 3%; and IE, 9%. Thus, user-input tracking is performed for a very small portion of a process's lifetime and, hence, our designation of data as clean is conservative.

Table 7. Performance overhead of the system. The **Tainting** column identifies the factor slowdown of running Qemu with tainting over vanilla Qemu. Each MX column identifies the factor slowdown (over vanilla Qemu) of performing both tainting and behavior matching for the given malspec. Startup time is not included and is on the order of ten seconds.

	Tainting	**M1**	**M3**	**M6**
Internet Explorer	5.25	11.53	7.19	5.64
pSCP	7.32	8.08	19.62	7.42
Agobot	3.01	16.40	23.73	16.84
rBot	9.50	11.20	11.08	9.62

4.7 Additional Malware

Though our sample malspecs target malicious bots, high-level specifications can be generated to identify other classes of malware. To demonstrate this, we evaluated four Trojans (Bancos, two variants of Banker, and Delf) and three mass-mailing worms (all variants of Bagle) using our previous malspecs plus a new malspec designed to detect self-propagation through email. With no modifications to the **Leak** malspec (M7), each Trojan matched it. To identify self-propagation through email, we modified the **Remotely-Initiated Send Email** malspec (M3). Rather than requiring that the data-flow be from the network to an SMTP message, we specified that the data-flow must be from the code of the executable itself to an SMTP message, which corresponds to a process sending its own code in an email. This demonstrates that specifying signatures for entirely new classes of malware can be straightforward and intuitive.

4.8 Performance Overhead

We evaluated the performance overhead of our system on a subset of the malicious and benign applications used in the evaluation, including Agobot, rBot, pSCP, and Internet Explorer. We ran each application under three different scenarios: (I) Qemu with no tainting; (II) Qemu with tainting; (III) Qemu with tainting and behavior matching for each of three different malspecs. For each application under each scenario, we measured the amount of wall clock time elapsed between a set of events captured in system logs. We selected events that did not depend on user input, so as to preserve as much determinism as possible.

The **Tainting** column in Table 7 identifies the factor slowdown of using Qemu with tainting over Qemu without tainting, which we refer to as vanilla Qemu. We rely on previous work to determine the overhead of vanilla Qemu relative to native execution, which is substantial: on the order of a 7X to 23X [12]. Each MX column identifies the factor slowdown of performing both tainting and behavior matching for the given malspec. To obtain the total slowdown over native execution, we add the MX value to the numbers in [12]; e.g., running behavior matching using the M3 malspec on rBot exacts an 18X to 34X performance penalty over native execution. Our system yields rich information and would ease the analysis performed in applications which may be less performance sensitive.

5 Limitations and Future Work

Limitations: There are several approaches to evasion that we can imagine attackers would adapt against a system such as ours. In particular, since we identify correlated sequences of system calls, efforts to disrupt our ability to correlate are an obvious choice. This disruption could take the form of splitting the work required to achieve some high-level action across multiple processes or across different instantiations of the same process. Another high-level approach at evasion relates to our assumption that the malicious process interacts with the kernel. Malware that expropriate kernel functionality would disrupt our ability to see and thus correlate their events. For example, an application could use raw sockets and write its own IP and transport-layer headers rather than calling the standard `sockets` functions such as `connect`, `accept`, and so on. Malicious software could also attempt to subvert our user-input tracking. Another approach to evasion relates to breaking our assumption about data-flow; in particular, malware could convert data-flow dependencies into control-flow dependencies thus defeating our mechanism for determining when an action is remotely-initiated. Finally, because we are interposing on a process, we are vulnerable to Time-Of-Check-Time-Of-Use (TOCTOU) bugs as in [24].

Future Work: We are very interested in exploring automated ways of generating the behavior graphs at various layers of the hierarchy. At $L0$, perhaps given source code access, we could ascertain precisely the set of low-level events (and the constraints on those events) that corresponds to each `sockets` operation. Moving up the hierarchy, such access would also presumably enable us to determine all possible sequences of events which achieve some semantic effect, such as `tcp_client`. An alternative approach may be to use symbolic execution to infer these behavior graphs. In this way, we would still achieve our semantic understanding of the aggregate effect of a process's actions but would have more confidence in our coverage than can be obtained through even rigorous testing.

6 Related Work

Behavior-Based Malware Detection: Host-based behavior-based research has been done to identify rootkits, spyware, and bots [22,23,20,19,16]. In [19], Cui *et al* identify *extrusions*: stealthy outgoing network connections made by malicious processes. In the commercial sector, Sana Security's ActiveMDT [21] correlates a process's exhibition of various mostly stateless behaviors to determine whether the process is likely to be malicious. The simple behaviors include: whether a process spawns or terminates other processes, the directory from which a process executes, whether the process attempts to hide, and so on.

Egele *et al* present a method and system for detecting spyware implemented as a Browser Helper Object (BHO) in [30]. The method identifies *malicious information access and processing* when sensitive information flows (such as the list of URLs visited) are written by a BHO to the network, file system, or shared

memory. Moreover, they perform static analysis to identify instructions that are control-dependent on sensitive information. Since spyware-writers could prevent the static analysis in [30] from identifying the post-dominator node, they consider failure of their static analysis to be indicative of malicious intent. This control-flow tracking is only performed for BHO code so it's unclear whether such tracking, if applied to general-purpose programs, would blur the ability to distinguish between malicious and benign. Yin *et al* developed a related malware detector, *Panorama* [18], which performs full-system, instruction level tainting and can express more diverse leakage policies than [30]. We can express the behavior identified by these systems using our specification language. As with [18], we do not currently track implicit information flows.

The behavioral specifications developed by Christodorescu *et al* [11] are similar to ours. Our specifications differ in three important ways. First, we use AND-edges which enables expressing concurrent behaviors. Second, we introduce synthetic event nodes, in order to identify complex behaviors hierarchically. Additionally, the specifications used in Christodorescu's work were generated automatically using data mining techniques, as opposed to the manual techniques we used. This has a few significant implications. Most importantly, their specifications identify sequences of actions which happen to occur in some malicious software; the aggregate effect of such sequences is unknown as is the value to the malware of performing those actions. That is, their specifications may identify *incidental*, rather than fundamental or mission-critical, behaviors as are targeted by our work. Additionally, no effort is made to cover semantically equivalent sequences. Consequently, there may be alternative sequences of system calls which have the same effect as a mined sequence but are not identified in their graphs.

Dynamic Code Analysis: Some systems use emulation to monitor the execution of suspicious executables [27,33,34]; however, rather than attempting to infer high-level behaviors, these systems merely report the numerous low-level events, such as system calls and API invocations, generated during execution. Other research has focused on addressing the shortcomings of dynamic analysis, including using symbolic execution to explore multiple execution paths [31,32].

Semantic Gap Problem: The semantic gap problem was explored by Garfinkel *et al*, as part of an attempt to embed an intrusion detector into a virtual machine monitor [25]. Related systems include honeypots [29,28], where introspective capabilities are used to examine the state of the filesystem in order to detect hidden files. Rather than encoding semantic information about the system, Jones [35] applied implicit techniques to infer relevant state. One notable result was the use of these techniques to detect processes hidden by rootkits [36].

7 Conclusion

Bots are an extremely widespread and serious problem, allowing remote bot masters to direct the activities of millions of compromised hosts. We develop new

behavioral monitoring techniques that are effective for identifying meaningful high-level actions, based on hierarchical behavior graphs. Behavior graphs provide a high-level specification language that can be used to describe semantically meaningful behaviors such as "proxying", "keystroke logging", "data leaking", and "downloading and executing a program." To address evasive malware behavior, our specifications are carefully crafted to detect alternate sequences of events that achieve the same goal.

Our experimental emulation-based detector identifies when a process performs a specified high-level actions, regardless of the process's source-code implementation of the action. We tested multiple malicious bots and benign programs and found that we were able to thoroughly identify high-level behaviors across a diverse code base. In addition, we are able to distinguish malicious execution of high-level behaviors from benign ones by distinguishing remotely-initiated from locally-initiated actions.

References

1. Symantec Internet Security Threat Report, Trends for January-June 07, Volume XII, September 2007.
2. Keizer, G.: Bot Networks Behind Big Boost In Phishing Attacks. TechWeb, Nov. 2004.
3. Parizo, E.: New bots, worm threaten AIM network. SearchSecurity, Dec. 2005.
4. Naraine, R. Money Bots: Hackers Cash In on Hijacked PCs. eWeek, Sept. 2006.
5. Overton, M.: Bots and Botnets: Risks, Issues, and Prevention. In Virus Bulletin Conference, Oct. 2005.
6. Ianelli, N., Hackworth, A.: Botnets as a Vehicle for Online Crime. CERT Coordination Center, Dec. 2005.
7. Ilett, D.: Most spam generated by botnets, says expert. ZDNet UK, Sept. 22, 2004.
8. Christodorescu, M., Jha, S.: Testing Malware Detectors. In Proc. of the International Symposium on Software Testing and Analysis, July 2004.
9. SRI Honeynet and BotHunter Malware Analysis Automatic Summary Analysis
10. Jevans, D.: The Latest Trends in Phishing, Crimeware and Cash-Out Schemes. Private correspondence.
11. Christodorescu, M., Jha, S., and Kruegel, C.: Mining specifications of malicious behavior. In Proc. of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, August 2007.
12. NoAH Foundation: Containment Environment Design
13. Chen, P., and Noble, B.: When Virtual is Better than Real. Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems.
14. Petritsch, H.: Understanding and Replaying Network Traffic in Windows XP for Dynamic Malware Analysis. Master's Thesis, February 2007.
15. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-Aware Malware Detection. In IEEE Symposium on Security and Privacy, May 2005.
16. Stinson, E., Mitchell, J.: Characterizing Bots' Remote Control Behavior. In Proc. of the 4th DIMVA Conference, July 2007.
17. Newsome, J., Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In Network and Distributed Systems Symposium, February 2005.

18. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In Proc. of the 14th ACM conference on Computer and communications security, October 2007.
19. Cui, W., Katz, R., Tan, W.: BINDER: An Extrusion-based Break-in Detector for Personal Computers. In Proc. of the 21st Annual Computer Security Applications Conference, December 2005.
20. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based Spyware Detection. In Proc. of the 15th USENIX Security Symposium, August 2006.
21. United States Patent Application 20070067843 Method and apparatus for removing harmful software. Williamson, Matthew; Gorelik, Vladimir. March 22, 2007.
22. Strider GhostBuster Rootkit Detection
23. Wang, Y., Beck, D., Vo, B., Roussev, R., Verbowski, C.: Detecting Stealth Software with Strider GhostBuster. Microsoft Technical Report MSR-TR-2005-25.
24. Garfinkel, T.: Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In Network and Distributed System Security, Feb. 2003.
25. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In Network and Distributed Systems Symp., Feb. 2003.
26. Nilsson, N.: Problem-Solving Methods in Artificial Intelligence. McGraw-Hill, 1971.
27. Bayer, U., Moser, A., Kruegel, C., and Kirda, E.: Dynamic Analysis of Malicious Code. Journal in Computer Virology, Volume 2, Number 1, Springer Computer Science Journal. August 2006.
28. Jiang, X., Xu, D., and Wang, X.: Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007), Alexandria, VA, November 2007.
29. Jiang, X., Wang, X.: 'Out-of-the-box' Monitoring of VM-based High-Interaction Honeypots. Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID 2007), Queensland, Australia, September 2007.
30. Egele, M., Kruegel, C., Kirda, E., Yin, H., and Son, D.: Dynamic Spyware Analysis. Proceedings of Usenix Annual Technical Conference. USA, June 2007.
31. Moser, A., Kruegel, C., and Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. Proceedings of IEEE Symposium on Security and Privacy, IEEE Computer Society Press. USA, May 2007.
32. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Poosankam, P., Song, D., and Yin, H.: Book chapter in "Botnet Analysis", Editors Lee W., et. al., 2007.
33. Norman Sandbox
34. Willems, C.: Automatic Behaviour Analysis of Malware. Master Thesis. University of Mannheim.
35. Jones, S.: Implicit Operating System Awareness in a Virtual Machine Monitor. Ph.D. Thesis, University of Wisconsin - Madison, April 2007.
36. Jones, S., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: VMM-based Hidden Process Detection and Identification using Lycosid. In ACM International Conference on Virtual Execution Environments, March 2008.
37. Vasudevan, A., and Yerraballi, R.: Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. Proceedings of IEEE Symposium on Security and Privacy, IEEE Computer Society Press. USA, May 2006.
38. Bellard, F.: QEMU Accelerator (KQEMU).
39. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator.