Storytelling Security: User-Intention Based Traffic Sanitization

Huijun Xiong Danfeng (Daphne) Yao Zhibin Zhang

***Abstract*—Malicious software (malware) with decentralized communication infrastructure, such as peer-to-peer botnets, is difficult to detect. In this paper, we describe a traffic-sanitization method for identifying malware-triggered outbound connections from a personal computer. Our solution correlates user activities with the content of outbound traffic. Our key observation is that user-initiated outbound traffic typically has corresponding human inputs, i.e., keystroke or mouse clicks. Our analysis on the causal relations between user inputs and packet payload enables the efficient enforcement of the inter-packet dependency at the application level.**

We formalize our approach within the framework of protocol-state machine. We define new application-level traffic-sanitization policies that enforce the inter-packet dependencies. The dependency is derived from the transitions among protocol states that involve both user actions and network events. We refer to our methodology as *storytelling security*.

We demonstrate a concrete realization of our methodology in the context of peer-to-peer file-sharing application, describe its use in blocking traffic of P2P bots on a host. We implement and evaluate our prototype in Windows operating system in both online and offline deployment settings. Our experimental evaluation along with case studies of real-world P2P applications demonstrates the feasibility of verifying the inter-packet dependencies. Our deep packet inspection incurs overhead on the outbound network flow. Our solution can also be used as an offline collect-and-analyze forensic tool.

I. Introduction

Personal computers have been and continue to be targets of many clandestine cyber crimes. Modern attackers aim to remotely control infected machines and conceal their tracks. Because they are able to infect a large number of distributed computers (e.g., the recently taken-down Mariposa botnet has estimated 12 million victims [12]), the malicious activities such as denial-of-service attacks launched from individual hosts may not be obvious to the conventional statistics based detection. Botnet victims are usually distributed across the globe, e.g., Mariposa botnet reached into 190 countries [12]). For botnets with a centralized command-and-control (C&C) architecture, Internet-wide (as opposed to local area network) detection of suspicious network-traffic patterns can be effective in identifying an unusually large traffic volume to specific domains or IP addresses.

In comparison, botnets (and malware in general) based on peer-to-peer (P2P) communication architecture is much harder to detect than those run with centralized servers. For example, `Peacomm` uses P2P network to search for new commands and executables. Each `Peacomm` bot's binary comes with a hardcoded list of peers. The malware is delivered as a trojan, as it is disguised as video attachment to email. Once it gets executed on the victim machine, it publishes itself to the network by contacting its peers. The bot searches for some keys that are hardcoded. The search returns a value that is decoded into a URL, which hosts the new executables that bots can download. This search-and-download process is repeated, each time with a new search key. Keys are generated based on the current date and a random number from 0 to 31 [7]. A Peacomm peer is identified with a hash value as well as IP address and port number. The size of peer list varies but is usually around one hundred. Upon execution, the bot actively sends heartbeat messages at high port numbers. Other P2P botnets include Storm and Nugache [5], [20].

Naive statistical approaches of Internet-wide detection can be easily circumvented by attackers through the use of techniques such as fast IP flux [9] or domain flux, where IPs or domain names of command servers change frequently. The scale and the distributed and dynamic nature of data also increase the difficulty of accurate detection. Recently, researchers proposed a method for localizing botnet members based on specific communication characteristics underlying types of P2P botnets [17]. Despite the progress of network-based detection techniques, there is an urgent need for providing *host-based* security mechanisms that can provide a robust line of defense. The host-based protection requires sophisticated techniques beyond the existing signature-based solutions found in most commercial security products for PCs.

In this paper, we describe a host-based network security approach for monitoring outbound P2P traffic and detecting suspicious malware activities originated from the host. Specifically, our solution is capable of examining the payload of outbound packets and perform content-based correlation with user inputs (namely keyboard and mouse inputs), in order to identify the network activities not initiated by the user.

Our ultimate goal is to devise an intelligent agent that is capable of performing complex inference to distinguish suspicious traffic from legitimate ones. Such an agent comprehensively observes activities of a host across the operating system and applications and makes real-time decisions. Our key observation is that user-initiated outbound traffic typically has corresponding human inputs. Our goal is to block traffic that is *not* associated with legitimate user inputs, which is referred to by us as *user-intention based traffic sanitization*. For P2P traffic, however, one of the main technical challenges for correlating user inputs with outbound network connections

This work has been supported in part by NSF grant CAREER CNS-0953638 and CNS-0831186.

Xiong and Yao are with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24060. Zhang is with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China.

is that some connections are automatically generated by P2P clients. These traffic should be properly examined without creating false alerts.

We identify the *inter-packet dependency* in the application-layer traffic. Inter-packet dependency is defined by us as the causal relations among a sequence of packets observed, and how their payloads are related to their order of arrival. For example, in typical P2P applications search keywords (specifically their hash values) appear in the payload of outbound search requests; subsequently, file hashes returned from the peers in response to the user's query appear in the subsequent outgoing request at downloading. The correct inter-packet dependency can be obtained according to the protocol specifications of the (legitimate) application specifications. In this paper, we do not consider unknown protocols. Traffic of unknown protocol type can be used to infer protocol specification, as recently demonstrated by Wang et al [29].

We hypothesize that the inter-packet dependency can be identified and enforced at the application-layer traffic for security purposes. Traffic that does not follow the pre-defined inter-packet dependency can be identified and flagged. We perform a case study on eMule traffic demonstrating the feasibility of our hypothesis. We formally describe our models and present our prototype implementation and experimental evaluation.

Intuitively, this causal relation in network flows of a host build a logical story around the observed network events and user actions. Thus, we refer to our approach as *storytelling security*. The storytelling approach is powerful, and is useful beyond the specific P2P traffic studied. For example, similar analysis can be applied to file-system behaviors such as file-system access, as well as other application-level traffic such as HTTP flows of browser. In general, our approach is a specific form of the anomaly detection, which is field pioneered by Denning [4]. What distinguishes our work from existing ones is that *i)* we uniquely integrate user behaviors in our inference, and *ii)* we focus on application semantics that provides a rich and concrete context for the analysis.

In order to realize user-intention based traffic sanitization, we also need to interpret the semantic meanings of user inputs, specifically to understand the implication associated with user actions. For example, it is straightforward to observe a mouse click by the user and learn its timestamp, coordinates, and the process to which it is fed at the kernel level. In comparison, *application-specific* information associated with a user action provides more concrete and useful. In this paper, we formally define the *semantic gap* of user intention and describe the associated technical challenges.

Our Contributions Our technical contributions are summarized as follows.

- We describe a new security methodology – referred to as the storytelling security – for inspecting outbound network traffic of applications. The analysis can be used as a forensic tool for diagnosing personal computers for abnormalities caused by malicious software or corrupted applications. We formalize our storytelling-security approach in the context of protocol-state machine (PSM). The uniqueness of our model is that we integrate dynamic user actions in constructing protocol-state machine and enforcing traffic-sanitization policies.
- We give a concrete realization of our storytelling-security approach in the context of P2P application. We describe the architecture and implementation of a host-based tool for monitoring and analyzing user-input events and traffic associated with a P2P file-sharing application in Windows operating system.
- We evaluate the performance of our tool in two deployment scenarios: real-time and off-line traffic sanitization. Our experiments show that both types of deployment are feasible. The overhead in the real-time analysis decreases the throughput of outbound traffic, indicating the tradeoff between security and usability. Our offline analysis incurs very low computational overhead. Our evaluation results suggest that our solution is better used as a diagnostic and forensic tool that either runs as needed in a diagnostic session or runs in the offline collection-and-analyze mode.

The rest of this paper is organized as follows. The definitions and models used in our work are given in Section II. Section III presents an overview of our solution and describes details of our approach. We present our prototype implementation in section IV and describe the experimental evaluation in section V. Finally, we discussed related work in section VI and conclude the paper in section VII.

II. Definitions and Models

In this section, we describe an abstract resource management model to represent typical operations of a peer-to-peer (P2P) client. This simplified model allows us to better present and explain our traffic sanitization solution in Section III. Then, we give our security model, including our attack model and security assumptions.

A. An Abstract Model for a P2P Client

For a typical file-sharing P2P client on a host, its operations include *setup*, *publish*, *search and download*, and *maintenance*.

- *Setup* All resources including peers and resources (i.e., data objects) in a peer-to-peer network are identifiable by a unique hash value. The hash value associated with a peer – *PeerID* is computed and assigned to it when it joins the peer-to-peer network. For practical purposes, the PeerID can be generated by P2P file-sharing software.
- *Publish* A data owner generates hash values – *ObjectID* – for data objects to be shared. The process of publishing is to announce to neighboring nodes the ownership of certain objects. An ObjectID is computed based on corresponding file's meta information.
- *Search and download* ObjectIDs are used for searching objects in peer-to-peer networks. Each P2P node maintains a distributed hashtable (DHT) that is used for searching and routing purposes. The table contains the ObjectIDs of neighboring nodes that are close according to certain virtual distance measure in the P2P overlay network. We refer readers for P2P literature such

as Tapestry [31], Chord [26], and Kad [13] for more distance-computation information.

- *Maintenance* A P2P client periodically sends to its neighbors in DHT *heartbeat* messages to inform its availability, which is used to keep an updated distributed hashtable.

For traffic sanitization, we focus on analyzing operations that involve network activities, such as search and download. We consider two common types of explicit user actions on a P2P client that may trigger network activities as follows.

- *Searching for a data object:* User enters one or more keywords to a textbox via the external keyboard device. Upon receiving the keywords, the P2P client computes the hash values $\{h_1, \ldots, h_n\}$ for each of the n keywords. It queries its neighbors (in the DHT) for the requested ObjectIDs. Specifically, the P2P client sends outbound packets with the ObjectIDs $\{h_1, \ldots, h_n\}$ included in the payload. The P2P client receives from the neighboring nodes a list of file information (file names and their corresponding ObjectIDs) as well as the information of owners (IP addresses and PeerIDs). A schematic drawing of the search process is in Figure 1.
- *Downloading a file:* The user selects one or more files to download via explicit mouse clicks. The P2P client sends outbound requests to corresponding peers and retrieves files from one or more peers (for example, in the BitTorrent protocol pieces of a file may be retrieved from multiple peers).

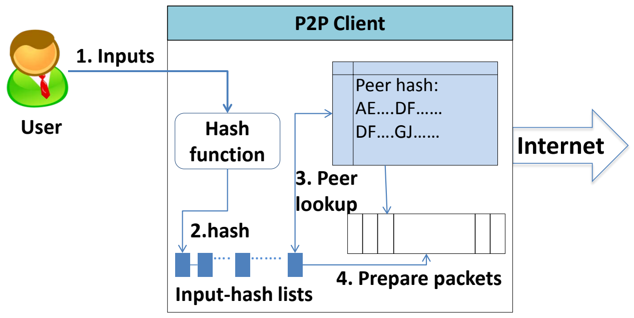


Fig. 1. A schematic drawing of how user inputs are transformed into outbound search requests in a P2P file-sharing client.

A P2P client may perform network operations without explicit user actions or permissions, such as sending heartbeat messages for maintaining DHT or uploading files in the user's shared folder to other peers upon requests. These automatic operations make our traffic sanitization challenging, because there is no causal relation between user actions and these network events. Whitelists can be used to filter some of the traffic, which we describe in Section III.

B. Attack Model and Security Assumptions

We consider stealthy malware that is aware of P2P file sharing applications installed in the host and secretly sending outbound P2P traffic with the same format. The malware can corrupt the P2P application. Thus, *the P2P file sharing application is not assumed to be trusted.* Malware may run as a user-level application – Type-0 malware according to the stealthy malware taxonomy [21]. Malware is active in making outside connections for command & control, attacks, or exfiltrating. Exfiltration refers to exporting stolen information such as sensitive personal data or proprietary corporate information as in `Hydraq` malware.

We assume that the kernel as well as the components in our sanitization framework along with its files are not corrupted by the malware. This trust assumption on secure kernel can be relaxed by utilizing trusted computing infrastructure such as Trusted Platform Module (TPM) [28], [27]. TPM is available on most commodity PCs through a standard attestation procedure [22]. TPM provides the guarantee of load-time code integrity. System integrity can also be achieved through the use of virtual machine monitor (VMM). At the VMM-level, it is technically complex to interpret the semantics of application-specific guest OS data and inputs as demonstrated in [11]. Our analysis would not be efficient in VMM-based systems, due to the technical difficulties in accessing and reconstructing dynamic application-level data. TPM does not provide detection ability for run-time compromises such as buffer overflow attacks [6]. This issue is still an open research problem in the security community with some recent promising development [14], [15], [16].

In this work, we assume the packets belonging to a specific (P2P) application can be identified on a host. This assumption can be realized by several means, for example, to associate network flows with their corresponding process information (such as PID and process name) using constant `netstat` queries. Because the P2P application is not trusted, our deep packet inspection is necessary for filtering out malware-triggered traffic disguised as legitimate P2P traffic. This method can be generalized to host-wide monitoring. This topic is subject of our future work. Our analysis works only for unencrypted traffic.

III. Overview of Our Architecture

In this section, we first define protocol-state machine, and describe how they are used to realize storytelling security. Then, we briefly explain the technical challenges associated with inferring user intention. Finally, we present our architecture for user-intention based P2P traffic sanitization.

A. A Protocol-State Machine for P2P Application

To detect malware's network activities, we enforce the properties and data dependencies of the protocol states of an application. Our techniques inspect the network traffic and system events of a host. We aim to identify the causal relations between user intention and network events. We assume that all the traffic belonging to a specific application have been identified, as explained in Section II-B. We define *protocol-state machine* (PSM) in Definition 3.1 following Wang *et al* [29].

Definition 3.1: The protocol state machine is a finite state automaton illustrating all possible states in the protocol and conditions for the transitions among states.

Given the specification of a protocol, protocol-state machine can be obtained. PSM is useful for network security, in particular anomaly detection. For example, malformed packets (e.g., TCP packet with both SYN and FIN bits set) can be identified and rejected. Recent work [29] also demonstrated the feasibility of using statistical methods to infer probabilistic protocol-state machine from unencrypted traffic, when specifications are unknown.

Our analysis on PSM has a unique flavor, as we focus on the states and their transitions that have two properties: *i)* generating outbound network traffic and *ii)* transitions triggered by user inputs. Figure 2 illustrates the PSM of a typical P2P application. Some state transitions are implicit and happen without any external inputs, e.g., from `init state` to `waiting for user inputs`. For the simplicity of description, we do not consider file uploading, which can be easily included to our work. Our PSM model described in this paper is a simplified abstraction of real-world applications. Enforcing the protocol state machine may be complex in general, especially for modern applications that support asynchronous user-interaction architecture such as AJAX.

B. Traffic Types and Analysis Complexity

Based on the protocol-state machine in Figure 2, we distinguish four types of outgoing packets in P2P applications: *Type I: heartbeat*, *Type II: keyword-search request*, *Type III: peer-connection request*, and *Type IV: file-download request*. Different sanitization policies are applied to different types of traffic. In this work, we focus on inspecting Types II, III, IV traffic, because they are directly or indirectly triggered by user actions. Types I traffic does not involve explicit user actions, which is discussed in Section III-C.

Sophisticated sanitization policies can be generated based on evaluating the current protocol state with respect to the previous states. Each state is associated with some information, e.g., user inputs, incoming or outgoing traffic. Inter-state comparison and analysis based on the state information incurs overhead.

Consider a path on a protocol-state machine of length k. A path represents a traversal of certain protocol states following allowed transitions. The length of the path is the number of states in the traversal. Denote n_i as the size of data items associated with the i-th state on the path ($i \in [1, k]$). Policies that involve linear-correlation based analysis on this path of length k incur computation complexity $O(\Pi_{i=1}^{k} n_i)$ (i.e., $O(n_1 n_2 \ldots n_k)$. We define linear correlation as the analysis that involves comparing each data item in its linear form – only linear transformation of data items is used, e.g., $y = ax + b$. Our architecture provides the support for general traffic-sanitization policies. We demonstrate a simple policy that compares two sets of hash values (representing PeerIDs and ObjectIDs): one set is generated from user inputs and incoming traffic, and the other set appears in outgoing packets. Further details can be found in Section V-B.

C. Technical Challenges of Storytelling Security

One technical challenge in our approach is *how to capture user intention on a computer*. In our work, we collect the input events from keyboard and mouse devices to represent user intention. To predict legitimate P2P traffic based on observed user inputs, a straightforward solution is to compare the timestamps of input events and outbound packets. In that case, outbound packets are allowed only when they happen shortly after some user inputs. However, this simple approach is coarse grained, and easy for malware to circumvent. We perform payload inspection with policies, which is more fine-grained than temporal-based comparison such as in [3].

User-input events collected at the kernel level are usually transformed by the destination applications. As a result, events such as mouse clicks have specific meanings or semantics in their destination application. For example, in a browser a user clicks on a hyperlink; the semantics of the mouse-click event includes the content of the hyperlink. Computation on user inputs is another type of semantic transformation. For example, in P2P file-sharing application, a search keyword (e.g., Harry Potter) is entered by a user, the hashes of which (e.g., *H(Harry)* and *H(Potter)*), is used to form a P2P search request for the corresponding files. We define the *semantic gap* of inputs in Definition 3.2.

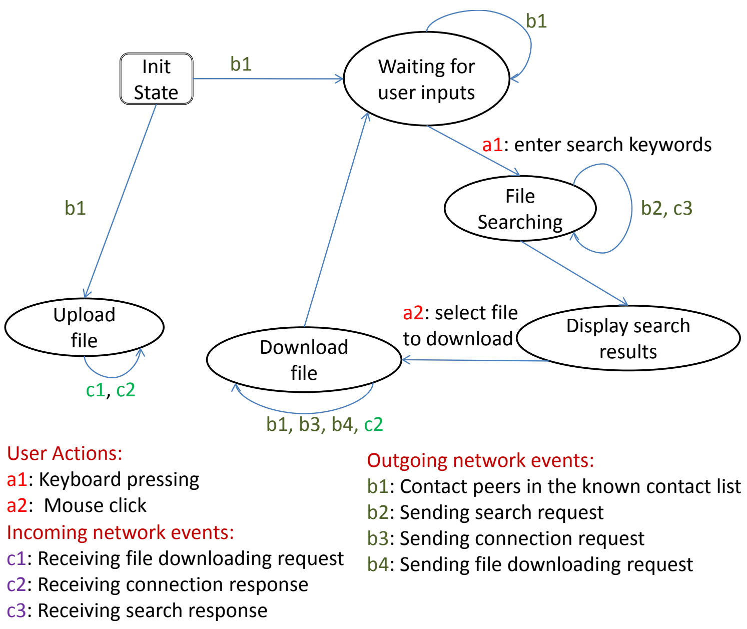
Definition 3.2: The semantic gap of inputs refers to the differences in the meanings of user-input events at the kernel level and the application level. It captures the lack of semantic information of user-input events collected within the kernel.

Our analysis is independent of the P2P application, thus it is robust against compromised P2P applications. Yet, the problem of semantic gap exists for mouse clicks. To tackle the problem in our specific P2P file-sharing context, we use the protocol-state machine to carefully propagate the trust from the data of `init state` to the data of subsequent states. The trust chain is extended – a piece of information is trusted if and only if it is either entered by the user or directly or indirectly caused by the user actions. Specifically, we assume that the initial keyboard inputs to the application are search keywords and trusted; to eliminate the need for interpreting semantic meanings of mouse-click events during the file-selection-and-download phase, we extract legitimate PeerIDs and ObjectIDs from the legitimate incoming traffic. More details are explained in the next section.

To realize P2P traffic sanitization, we collect two types of data flows: user activities and outbound traffic, and then perform a content-based analysis according to pre-defined policies. Figure 3 shows the architecture of our system, which has two main components *correlation engine* and *traffic monitor* as described next. We explain our correlation engine in the next section. The traffic monitor is described in the following section.

D. Correlation Engine and Traffic-Sanitization Policies

The correlation engine performs the input-traffic correlation on two data streams – user-input data and network packets. We monitor both incoming and outbound packets of a specific application at the transport layer on a host. We focus on inspecting traffic of Types II, III, and IV. Type I (heartbeat) messages are sent automatically without direct or indirect user actions, and thus are not considered. They can be filtered



Init State
b1
Waiting for user inputs
b1
a1: enter search keywords
File Searching
b2, c3
b1
Upload file
c1, c2
Download file
b1, b3, b4, c2
a2: select file to download
Display search results
User Actions:
a1: Keyboard pressing
a2: Mouse click
Incoming network events:
c1: Receiving file downloading request
c2: Receiving connection response
c3: Receiving search response
Outgoing network events:
b1: Contact peers in the known contact list
b2: Sending search request
b3: Sending connection request
b4: Sending file downloading request

Fig. 2. Schematic drawing of a protocol state machine of a typical P2P application.

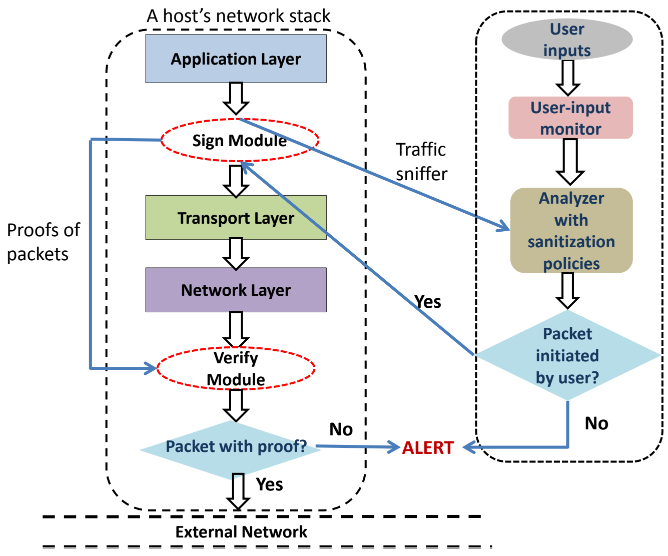


Fig. 3. Schematic of components in our traffic sanitization framework and their interactions with the host's network stack.

based on a whitelist, for example, a host can only send Type-I heartbeat messages to the remote peers specified by a white list. The whitelist prevents a host from initiating connection to any arbitrary remote peer. Yet, the host can request objects from previous unknown peers *if and only if* that peer's ID appears in the incoming-hash list, which indicates that the peer (namely its PeerID) has the requested object. The incoming-hash list is populated as a result of previous object search requests. Thus, the dependency and the causal relation between the peer ID and previous search response/request are verified. We formalize our policy in Step 3 below.

A schematic drawing of the detailed workflow regarding our traffic sanitization is shown in Figure 4. The detailed operations including the specific traffic-sanitization policies that we use are explained below.

1) Step 1a of Figure 4. Our user-input monitor runs at the kernel level and intercepts all user's keyboard events. It records the content, timestamp, and application information, e.g., {eMule.exe,4327,20:12:34,'w'} where 4327 is the process ID and 20:12:34 is the timestamp of the event. Keys entered at the kernel level may need pre-processing. For example, a user may backspace to delete typos; our input monitor records every key press including backspace. Thus, we need to sanitize and reconstruct user inputs before they are analyzed.
   In Step 2a, the input monitor passes user inputs for application-specific transformation, namely to generate corresponding hash values. The hash values are stored in a data structure that supports fast search, e.g., hashmap. The data structure referred to by us as *input-hash list* is accessed by the *Analyzer* described later.
2) Step 1b and 2b. A *Traffic collector* intercepts all the outbound traffic (i.e., requests) as well as incoming packets, specifically all incoming and outgoing Type II and Type III traffic as defined in Section III-B. It extracts the payload from packets for further inspection. We maintain two lists of hash values: *outgoing-hash list* and *incoming-hash list*. Outgoing-hash list is a data structure consisting of the hash values appearing in outbound packets of the P2P file-sharing application.
   The incoming-hash list contains the trusted hash values that are allowed to appear in the outbound traffic. The content of outbound packets depends on the preceding inbound traffic, which is captured in the list.
   The incoming-hash list is a data structure consisting of

the hash values appearing in the payload of incoming packets. The incoming-hash list contains either *i)* the ObjectIDs of requested files, and *ii)* the PeerIDs of those who have the requested objects. In our work, the enforcement of inter-packet dependency is embodied and realized by the comparison of the lists of hash values (See Step 3 below). Timestamp and process information corresponding to each hash value are obtained and stored. We performed a case study on eMule traffic, a popular P2P file-sharing application, to confirm our hypothesis that the inter-packet dependency in the application traffic can be identified and enforced with proper sanitization. The details of the case study is not shown due to page limit.

Extracting data from packets makes use of the knowledge of the packet format with no hidden data fields [1]. We ensure that packets are well formed according to the protocol specification, and obtain the fields corresponding to the hash-values in both requests and responses messages. These hash values are then carefully compared according to sanitization policies in Step 3 next.

3) Step 3. *Analyzer* takes the processed input data in *input-hash list* and packet payload in *outgoing-hash list* and *incoming-hash list*, and applies pre-defined sanitization policies that compare the two types of data flows. The policies are generated based on the protocol-state machine of the application.

In general, there is a tradeoff between the security and efficiency – more complex rules impose higher computation and data-storage overhead. Our traffic-sanitization policy specifies that *i)* the comparison is between hash values h, h' associated with the same process ID, i.e., $PID_h = PID_{h'}$; *ii)* each hash value (PeerID or ObjectID) in the outgoing-hash list H_{out} should previously appear in the input-hash list H_{user} or the incoming-hash list H_{in}; and *iii)* the timestamps T_h and $T_{h'}$ of two comparable hash values h, h' should differ within a threshold τ, which is formalized as follows.

$$\forall h \in H_{out}, \exists h' \in H_{in} \text{ or } H_{user}, \text{such that}$$
$$h = h', T_h - T_{h'} \leq \tau, PID_h = PID_{h'}.$$

For those hash values that fail the above verification, their corresponding outgoing packet information is logged and reported.

Our design shows how static protocol-specification information can be used to dynamically enforce system properties of a host. Our architecture is general and can be used to support complex policies on application-level traffic. Our model provides a concrete embodiment of the storytelling-security approach in the context of P2P application – *our security analysis tells a story about the user's interaction with the application*. Our prototype implementation and evaluation are presented in the next two sections.

[1]Hidden data fields in payload may be used for covert channels. The general discussion of covert channels is omitted due to page limit.

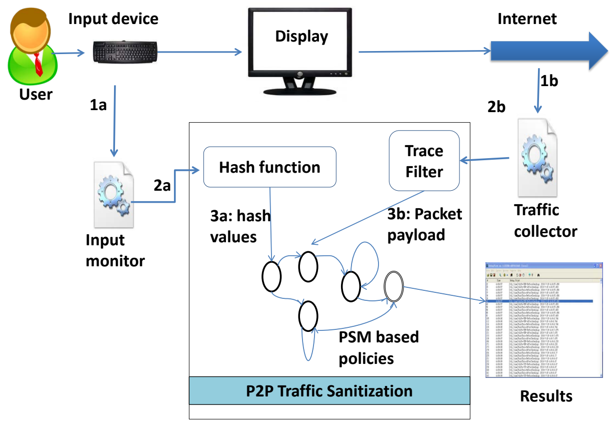


Fig. 4. An illustration of the workflow in our traffic sanitization framework as a concrete example of our storytelling-security approach.

E. Traffic Monitor With CompareView

Our scheme requires outgoing network packets to flow through two verification checkpoints. One checkpoint at the transport layer of the network stack of a host is for causal relation verification (as described in Section III-D above). The other is for traffic-integrity verification, which is explained here. A schematic drawing is shown in Figure 3.

We inspect every outgoing packet of a P2P application. Thus, we need to guarantee that no malicious bot packet bypasses our cause-relationship verification. However, malware may attempt to hide its traffic by circumventing or bypassing our transportation-layer checkpoint. Malware may disable the checkpoint all together. For the traffic-integrity verification, we examine every single outgoing network segment for its provenance proof. The goal is that no packet can circumvent our traffic integrity verification.

To prevent these attacks and ensure the *completeness* of collected traffic, we utilize an existing host-based traffic-monitoring approach called *CompareView* [25]. It forces all outbound traffic to pass through a transport-layer entry point, and identifies those that do not – referred to as provenance verification by [25]. The entry point can be used to deploy sophisticated personal firewalls that enforce application-specific policies. This entry point is where we deploy our sanitization policies for real-time traffic inspection.

Specifically, there are two kernel modules in CompareView, *Sign* and *Verify* modules. They extend the host's existing network stack. This traffic-monitor architecture is shown in Figure 3. The two kernel modules share a secret cryptographic key. The key is used to ensure the integrity of outbound network packets. All legitimate outgoing packets first pass through the Sign module, and then through the Verify module. The Sign module signs every outbound packet, and sends a short *provenance proof* to the Verify module on the same host, which later verifies the proof with a shared key. The proof indicates the provenance or origin of an outbound packet, i.e., the packet passed through the transport-layer entry point [25]. If a packet's proof is missing or cannot be verified, then it is labeled as suspicious. This packet bypasses the Sign modules, and likely is generated by stealthy malware. The proof can be realized with keyed-hash such as HMAC.

Real-time analysis is more difficult to realize that off-line analysis, as data from multiple sources needs to be pulled together. For efficiency and scalability considerations, our approach is more suitable for intrusion detection systems (IDS) as opposed to intrusion prevention systems (IPS). In IPS, the analysis results have to be applied to block malicious traffic at real time.

IV. System Implementation

In this section, we describe our prototype implementation in Windows operating system, including the detailed realization of traffic sanitization and input-traffic correlation. We also present a simple P2P client and the specific filtering rules that we use for evaluation.

A. Implementation Details

We implement a prototype of the P2P traffic sanitization framework by expanding the TCP/IP network stack in Windows XP. We describe our implementations of the main components *input monitor*, *traffic monitor*, and *analyzer* in our prototypes next.

Input monitor uses *Inputhook.dll* to record all the user inputs with timestamps and application information (e.g., process ID and name). All the collected data is stored in kernel file and protected by kernel. Monitoring keyboard inputs is realized using existing windows hook mechanism, such as *input hook*. Deliminators are used to process each record and to produce a list of words input from the user. Our prototype does not require the logging of mouse events.

The collected user inputs are stored character-by-character. We reproduce meaningful words from these raw inputs. This transformation is done by grouping the characters based on certain deliminators such as `0x20` (space), `0x0d` (enter), and `0x08` (backspace). This process yields the search keywords that user enters. The input monitor computes hash values on the collected search keywords. This input-hash list is stored in the memory, which can be accessed by the analyzer (described below). The list represents the *trusted values* that we use to infer legitimate traffic.

Traffic monitor and analyzer intercept and filter outbound traffic sending from the application layer. It can capture all outbound traffic and detect stealthy traffic that bypass our sanitization. We realize it by hooking on the Windows TDI (Transport Device Interface) driver's *tdi_send(), tdi_receive()* function. TDI is the interface in Windows OS that imposes the transport-layer access to applications. We use a TDI hook for the traffic-monitoring purpose. Our traffic monitor uses a special packet identifier (namely string `kwhv`) as a filter to capture traffic sent by our simulator (described in Section IV-B). It extracts the payload (namely hash values) from the outgoing packet. The hash value is passed down to the causal-relation analyzer for further investigation. Timestamp is recorded for each outgoing packet. Similarly, the incoming-hash list can be maintained.

Our prototype expands the TDI hook to realize an analyzer that enforces a simple traffic-sanitization policy – whether or not the outbound packet contains a hash value in the input-hash list and the incoming-hash list. In our implementation, we use the debug view to display the verification results.

Our prototype is implemented as an online tool for real-time detection. We also implement an offline collection-and-analyze version, which can be used as a forensic tool to diagnose computers in both personal or enterprise environments when needed.

Our prototype is implemented on top of TDIFW, which is a lightweight personal firewall for Windows operating system. A screenshot of the output of our tool is shown in Figure 5.

B. A Simple P2P-Client Simulator

To test the functionality of our traffic-sanitization prototype at real time, we implemented a simple peer-to-peer file sharing simulator in C++. This program simulates basic functions of peer-to-peer file-sharing software, including *i)* generating hashes based on user inputs, *ii)* sending keyword-search requests, *iii)* receiving search responses, and *iv)* generating file-search requests. Our traffic-sanitization prototype is used to inspect the outbound traffic in Steps *ii)* and *iv)*.

i) Hash-value generation. We use MD5 that outputs 128-bit hash value for a keyword entered or for a peer. MD5 is used in many P2P file-sharing applications including eMule. When a user launches a keyword search in our simulator, the simulator calls the hash function to generate corresponding hash value of each keyword.

ii) Submitting keyword-search requests. The simulator prepares and sends outgoing search requests based on the hash values in UDP packets. The packets are sent to a remote sink. To identify the outgoing packets sent by our simulator, we put a unique string *kwhv* at the beginning of our packet headers as shown in Figure 5. A more general approach is to use the native Windows API (e.g., IPHelper) that allows one to obtain the process ID and name information associated with a packet based on destination IP and port number.

iii) Receiving search responses. To be able to simulate the incoming traffic, we store the file-location information in each peer's file in a special folder on the localhost, including *status, keyword hash, file name, file hash*, and *file size*. The *status* field means the peer is active or not. *Keyword hash* is the hash value computed from the search keywords. *File name*, *file hash*, and *file size* are the name, hash value, and size of the file, respectively. Our client searches the hash values in this predefined folder for possible matches. Each result corresponds to a file whose hash value matches the keyword. The results are displayed to the user.

iv) Generating file-search requests. A user selects a file in the list of the search results. The client prepares an outgoing UDP packet with the hash value corresponding to the requested file. The packets are sent to a remote sink.

At the real-time detection, our traffic-sanitization tool analyzes the network flow generated by the simulator as follows. Our tool populates the input-hash list with the keywords entered via the keyboard in Step *i)*. Similarly, it populates the incoming-hash list with the hash values of keyword-search results returned in Step *iii)*. The outgoing-hash list is populated by parsing and extracting the hash values in Steps *ii)* and *iv)*. The traffic-sanitization policy of the simulator is that the payload (namely hash value) in an outgoing packet should appear either in the input-hash list or the incoming-hash list. In addition, a delay threshold can be specified to constrain the interval between the timestamps of the two events and hash values being compared. The purpose is to prevent stale hash values from being replayed.

C. *Synthesized malware traffic*

We write a proof-of-concept malware that is capable of injecting well-formed UDP packets to outbound traffic. The malware payload contains arbitrary data. Our traffic-sanitization tool is able to detect the malware activities, because they do not correlate to the required hash values.

A legitimate application may be compromised by malware, where the malware calls the application's APIs to send its traffic, for example malicious Firefox extensions (e.g., FormSpy) that function as spyware. In this case, the malware packet has the same format as the application's. In addition, the malware traffic and the legitimate application traffic share the same port and belong to the same process (and with the same PID). In our test, the malware payload contains the unique string `kwhv` for identification, similar to the regular traffic sent by our simulator. Alternatively, malware may be standalone application running on a unique port. Thus, the traffic monitor needs to intercept network activities at all open ports to ensure the completeness.

V. Performance Evaluation

In this section, we describe the evaluation experiments that aim to evaluate the efficiency of our solutions. We ran our experiments on Windows XP Professional with SP3 installed in a virtual machine with Intel Core 2 Duo CPU 2.50GHz, 512 MB memory. The computer was connected to the department wireless network through 802.11g protocol.

In our feasibility study (not shown), we manually inspected traffic from eMule peer-to-peer application. The results confirmed our hypothesis of the inter-packet dependencies and the causal relation between user's input activities and traffic payloads.

A. *Performance of Real-Time Packet Inspection*

We conducted series of experiments to evaluate network performance of our solution under the simulation environments. We ran the experiments on different scenarios to investigate the overhead of our solution. We vary the sizes of packets and the number of hash values, respectively, to evaluate their impact on the efficiency of our solution. We ran each experiment six times with the same hash values and packet sizes, and computed average results. The 128-bit hash values used in all of our experiments are randomly generated with MD5 algorithm.

We focus on evaluating the overhead of our hash-comparison based deep-packet inspection on the network throughput. We utilize a network performance measurement tool `Iperf` to collect performance data. We vary the packet payload from 2-Kbit to 60-Kbit consisting of random bytes. We prepared lists of randomly-generated input hashes that simulate hash values computed from user inputs. The numbers of values on the input-hash lists are 50, 100, 150, 200, 250, and 300. Because the hash values in the input list and in packets are random and have no correlation, the run time represents the *worst case* scenarios – none of the hash values matches between the input-hash list and payload. We also compare the throughput with the original CompareView framework that performs provenance verification on packets, but without any payload inspection.

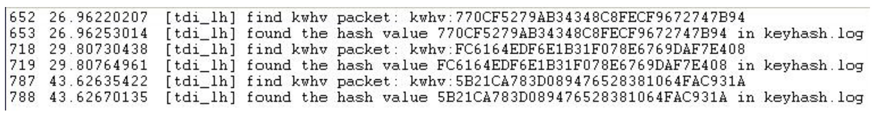
Figure 6 shows our results with TCP packets. Throughputs decrease as the number of hash values increase, which is expected. This fact is because the comparison of long strings takes longer time. Packet size has a small impact on the throughput. Larger packets yield higher throughput as expected, because of better amortized overhead. Clearly, deep-packet inspection slows down the network flow significantly in our experiments. Our results are not optimized. Fast string-comparison algorithms such as Boyer-Moore algorithm [1] that have been used for virus scan and genome-sequence comparison can be applied to improve the performance.

For run-time evaluation, we wrote a program that sends UDP packets with random-generated hash values. For each experiment, we sent 1,000 UDP packets for 6 times and computed the average time needed to send all packets. Again, the run time corresponds to the worst case scenarios with all mismatches. We compare our performance with CompareView and vanilla TDIFW – the latter involves no security mechanism. Our results are shown in Figure 7. TDIFW sends packets much faster than other settings. The run time of deep-packet inspection experiments is slightly slower than that of CompareView, but comparable. Larger packets take longer to transmit as expected. The increase of run time with the number of hash values is not monotonic.

B. *Performance of Off-line Packet Inspection*

Real-time execution of our traffic-sanitization analysis incurs overhead that may slow down outbound network flow from a host, as shown in Figures 6 and 7. The delay is mainly from two sources: *i)* computation overhead due to comparison, and *ii)* the synchronization delay between input-data stream and packet stream. In order to isolate the computation overhead associated with comparing hash values, we performed an off-line packet-inspection analysis as follows.

We evaluated the efficiency of our solution when executed offline, as opposed to real-time traffic inspection. We stored hash-value lists and UDP packets in the memory. The payload of each packet contains a 128-bit MD5 hash and some header bits. The list represents the input-hash list. We ran a Python script to measure the run time for comparison. We define *hit rate* as the percentage of packets that contain matching hash values with the input-hash list. For example, 50% hit rate means that half of the packets contain hash values that appear in the input-hash list. The worst-case scenario corresponds to



```
652  26.96220207  [tdi_lh] find kwhv packet: kwhv:770CF5279AB34348C8FECF9672747B94
653  26.96253014  [tdi_lh] found the hash value 770CF5279AB34348C8FECF9672747B94 in keyhash.log
718  29.80730438  [tdi_lh] find kwhv packet: kwhv:FC6164EDF6E1B31F078E6769DAF7E408
719  29.80764961  [tdi_lh] found the hash value FC6164EDF6E1B31F078E6769DAF7E408 in keyhash.log
787  43.62635422  [tdi_lh] find kwhv packet: kwhv:5B21CA783D089476528381064FAC931A
788  43.62670135  [tdi_lh] found the hash value 5B21CA783D089476528381064FAC931A in keyhash.log
```

Fig. 5. A screenshot of the output from our kernel monitor. kwhv is the unique string that we use for identifying outgoing packets belonging to our simulator.

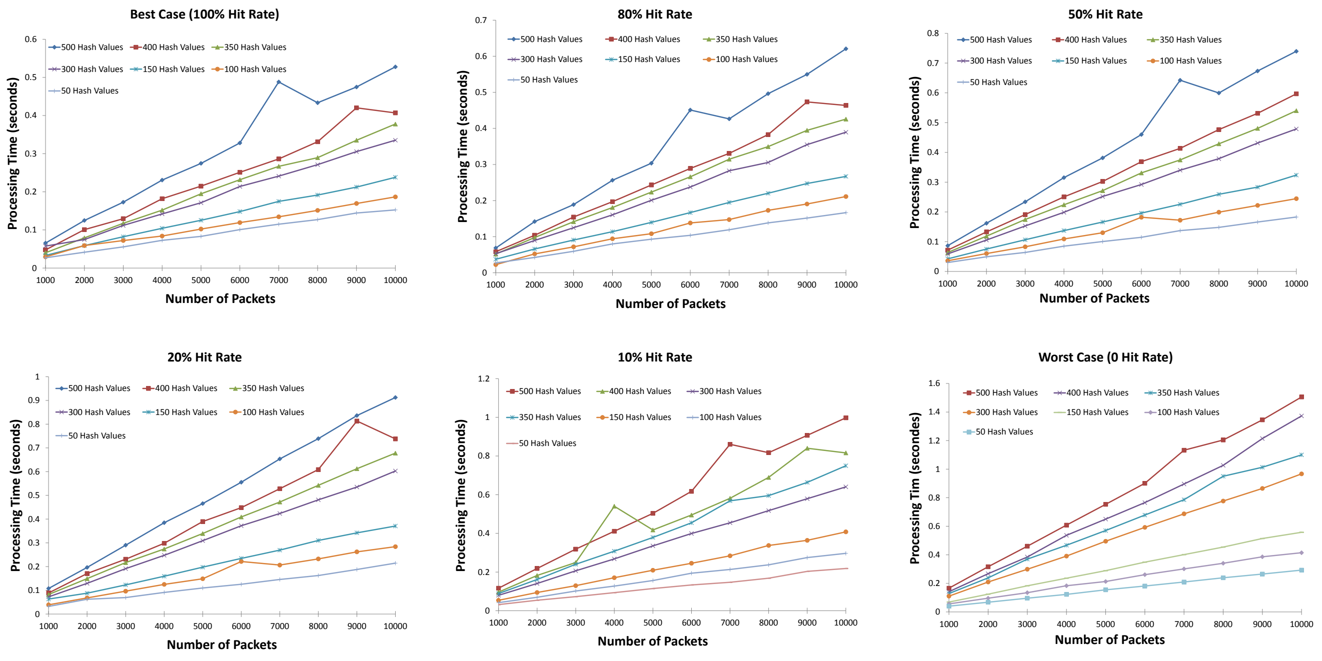


Fig. 8. The runtime of off-line payload-inspection analysis that compares UDP packets against lists of hash values.

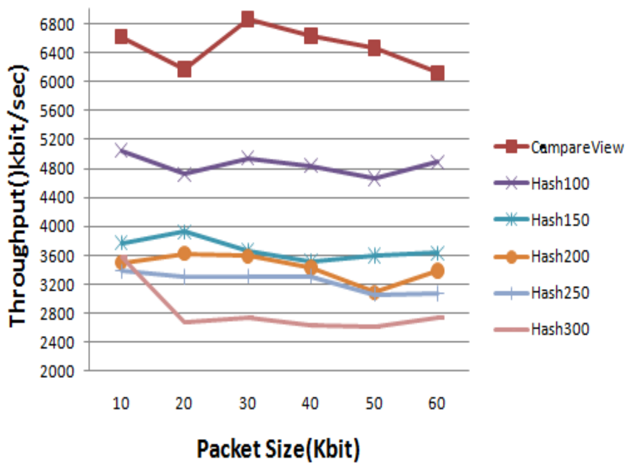


Fig. 6. Comparing the network throughputs of CompareView alone and our content sanitization with input-hash lists of various sizes. E.g., Hash250 indicates that there are 250 hash values on the input-hash list. The CompareView series of experiments do not involve any payload inspection.

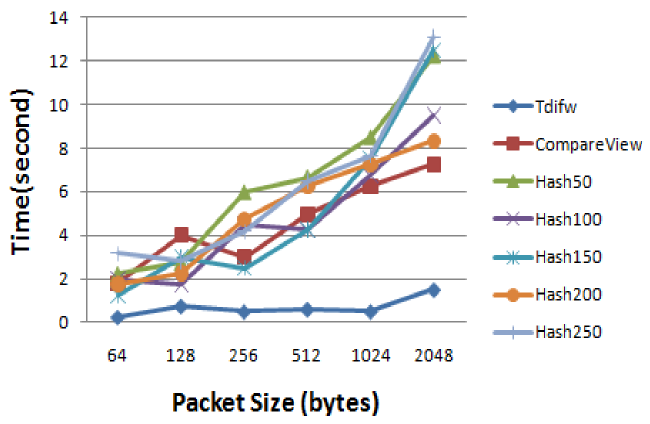


Fig. 7. Worst-case (all mismatches) run time for sending 1,000 UDP packets with random content and compare them against input-hash lists of various sizes. Tdifw refers to the experiments with no security checking on outbound packets. The CompareView series of experiments verify packet provenance, but not payload inspection.

0% hit rate, whereas in the best case the hit rate is 100%. The runtime of comparison with different numbers of hash values and UDP packets are evaluated. Figure 8 shows our results with varying hit rates.

Summary Specifically, the hash-value comparison in our experiment is between an input-hash list of size n and m number of outbound packets (i.e., outgoing-hash list of size m). The comparison complexity is bounded by $O(nm)$. The complexity can be generalized as shown in Section III-B.

The offline evaluation results show that the comparison-based payload inspection can be efficiently executed once the data is collected. Compared to the real-time traffic inspection, the offline analysis is much faster. It does not affect the network throughput of the host, and incurs no delay due to data-stream synchronization.

The disadvantage of offline evaluation is the storage overhead. To mitigate the problem, periodic data shedding can be scheduled to erase older entries. Another solution is to use our tool for a short-term (e.g., for 24 hours) for diagnostics and forensics purposes, a process conceptually similar to periodic virus scanning on a host.

VI. Related Work

User-intention based security has not been extensively studied in the computer-security literature as a general approach to protect a host and detect malware activities. However, there are several notable exceptions [8], [24]. Gummadi *et al.* [8] proposed a bot-detection solution on a personal computer that used hardware-assisted certification mechanism to distinguish human-generated traffic from malware-generated activities. Their solution requires a trusted proxy server to certify keystroke events entered by the user. Shirley and Evans [24] proposed to generate and enforce access-control policies for file systems based on user intentions that are inferred from the context of a transaction on a host. Cui *et al* [3] proposed BINDER that is a framework for detecting extrusion or exfiltration by measuring the delay between timestamps of user inputs and network packets. These solutions are for anomaly detection that is similar to ours.

The main feature that distinguishes our work from the above user-behavior inspired approaches is that our analysis is application specific, which enables more fine-grained and semantic-aware inspection on the network traffic. We uniquely leverage the protocol-state machine for security and our sanitization policies allow more sophisticated enforcement of the causal relations between user actions and network events.

Data-loss prevention (DLP) is a term used by computer-security industry to refer technologies that prevent sensitive data from leaked out of a computer or network. Network-based DLP solutions typically inspect outbound network packets for sensitive data and compute the correlation coefficients between the two types of contents. For example, a recent solution was proposed to compute the Pearson Correlation Coefficient (PCC) between artificially injected user inputs with outbound traffic to detect any keylogger on a host [18]. That approach is robust to linear transformation of the exported data, and will not be effective if strong encryption algorithm is used on the stolen data. The PCC-based analysis is complementary to our PSM-based policies and can be combined to provide more scalable application-specific traffic sanitization (See also our future work in the next section).

Analyzing and characterizing peer-to-peer traffic has been traditionally studied for identification peer-to-peer traffic with or without masqueraded techniques [2], [23]. For example, Collins and Reiter [2] proposed to identifying peer-to-peer file sharing traffic by observed service behaviors. Sen, Spatscheck, and Wang [23] applied application signatures on real-time detection of peer-to-peer traffic from large-scale traffic. The uniqueness of our approach is the human-behavior inspired network analysis on a host and its concrete realization based on the protocol-state machine.

Yen and Reiter [30] described an effective and elegant network-level traffic inspection solution for distinguishing malicious P2P bots from legitimate P2P file-sharing clients. Their analysis – based on properties such as volume, peer churn, and interstitial time distribution can be deployed at the router level. In comparison, we aim to construct a host-based tool for protecting and diagnosing personal computers. Our solution is able to utilize process, kernel, and payload information to provide a fine-grained control over the traffic. One specific difference between human-driven P2P traffic and machine-driven P2P traffic that was studied by [30] is the periodicity of activities – a higher self correlation indicates the likelihood of programmed bot. In comparison, our anomaly-detection approach does not target any specific malware-behavior characteristics.

VII. Conclusions

In this paper, our goal is to identify human-generated outbound traffic from what malware generates. We proposed a novel storytelling-security approach that verifies the dependencies among critical system events associated with an application. The analysis is based on the protocol-state machine, the data items associated with each state, and the transitions among the states. We gave a concrete example of using the storytelling-security methodology in sanitizing P2P traffic of a host. The policies are generated based on observing the causal relations between user actions and network events of a P2P application. This deep packet analysis allows the detection of suspicious outbound traffic that violates the causal relations, and is useful for detecting P2P-based malware. We implemented a prototype of our solution in Windows operating system and performed extensive analysis to evaluate its online and offline efficiency. Our experimental results showed that the offline performance provides fast analysis without creating any bottleneck on the network stack. Thus, our deep-packet inspection tool is better used as an intrusion-detection system as opposed to a real-time intrusion-protection system.

For future work, we plan to use statistical measures such as Pearson correlation coefficient to analysis the input-traffic correlation, which will allow us to perform the analysis in a much larger scale. We will also investigate the tradeoff between precision and efficiency. Another intriguing topic is to generalize our approach to the analysis and enforcement of composable protocol-state machines, which will allow us to represent the user's simultaneous interactions with multiple applications.

VIII. Acknowledgment

We thank the authors of [19] for sharing with us the malware `Trojan.Peacomm` binary code.

REFERENCES

[1] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of ACM*, 20:762–772, 1977.

[2] M. P. Collins and M. K. Reiter. Finding peer-to-peer file-sharing using coarse network behaviors. In *Proceedings of the 11th European Symposium on Research in Computer Security*, pages 1–17, 2006.

[3] W. Cui, R. H. Katz, and W. tian Tan. Design and implementation of an extrusion-based break-in detector for personal computers. In *ACSAC*, pages 361–370. IEEE Computer Society, 2005.

[4] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2), February 1987.

[5] D. Dittrich and S. Dietrich. P2P as botnet command and control: a deeper insight. In *Proceedings of the 2008 3rd International Conference on Malicious and Unwanted Software (Malware)*, October 2008.

[6] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 199–210, New York, NY, USA, 2008. ACM.

[7] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon. Peer-to-peer botnets: overview and case study. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007. USENIX Association.

[8] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-bot: improving service availability in the face of botnet attacks. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2009. USENIX Association.

[9] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling. Measuring and detecting fast-flux service networks. In *NDSS*. The Internet Society, 2008.

[10] S. Jha, R. Sommer, and C. Kreibich, editors. *Recent Advances in Intrusion Detection, 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings*, volume 6307 of *Lecture Notes in Computer Science*. Springer, 2010.

[11] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 128–138. ACM, 2007.

[12] G. Masters. Mariposa botnet mastermind nabbed, July 2010. urlhttp://www.scmagazineus.com/mariposa-botnet-mastermind-nabbed/article/175721/.

[13] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In P. Druschel, F. Kaashoek, and A. Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin / Heidelberg, 2002.

[14] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal tcb code execution. In *IEEE Symposium on Security and Privacy*, pages 267–272. IEEE Computer Society, 2007.

[15] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, New York, NY, USA, 2008. ACM.

[16] J. M. McCune, A. Perrig, and M. K. Reiter. Bump in the ether: A framework for securing sensitive user input. In *USENIX Annual Technical Conference, General Track*, pages 185–198. USENIX, 2006.

[17] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. BotGrep: Finding P2P bots with structured graph analysis. In *Proceedings of USENIX Security Symposium*, August 2010.

[18] S. Ortolani, C. Giuffrida, and B. Crispo. Bait your hook: A novel detection technique for keyloggers. In Jha et al. [10], pages 198–217.

[19] Trojan.Peacomm. http://www.symantec.com/.

[20] P. Porras, H. Saïdi, and V. Yegneswaran. A multi-perspective analysis of the Storm (Peacomm) worm. Technical report, Computer Science Laboratory, SRI International, 2007. http://www.cyber-ta.org/pubs/StormWorm/SRITechnical-Report-10-01-Storm-Analysis.pdf.

[21] J. Rutkowska. Introducing stealth malware taxonomy, November 2006.

[22] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[23] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of P2P traffic using application signatures. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 512–521, New York, NY, USA, 2004. ACM.

[24] J. Shirley and D. Evans. The user is not the enemy: fighting malware by tracking user intentions. In *NSPW '08: Proceedings of the 2008 workshop on New security paradigms*, pages 33–45, New York, NY, USA, 2008. ACM.

[25] D. Stefan, C. Wu, D. Yao, and G. Xu. Cryptographic provenance verification for the integrity of keystrokes and outbound network traffic. In *Proceedings of the 8th International Conference on Applied Cryptography and Network Security (ACNS)*, June 2010.

[26] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[27] TCG PC Client Specific TPM Interface Specification(TIS), Version 1.2. Trusted Computing Group.

[28] October 2003. Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands. Version 1.2, Revision 62.

[29] Y. Wang, Z. Zhang, and L. Guo. Inferring protocol state machine from real-world trace. In Jha et al. [10], pages 498–499.

[30] T.-F. Yen and M. K. Reiter. Are your hosts trading or plotting? telling P2P file-sharing and bots apart. In *ICDCS*, pages 241–252. IEEE Computer Society, 2010.

[31] B. Y. Zhao, L. Huang, S. C. Rhea, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.