IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING VOL.9 NO.2 YEAR 2012

Data-Provenance Verification For Secure Hosts

Kui Xu, Huijun Xiong, Chehai Wu, Deian Stefan, Danfeng Yao *Member, IEEE*

***Abstract*— Malicious software typically resides stealthily on a user's computer and interacts with the user's computing resources. Our goal in this work is to improve the trustworthiness of a host and its system data. Specifically, we provide a new mechanism that ensures the correct origin or provenance of critical system information and prevents adversaries from utilizing host resources. We define *data-provenance integrity* as the security property stating that the source where a piece of data is generated cannot be spoofed or tampered with. We describe a cryptographic provenance verification approach for ensuring system properties and system-data integrity at kernel-level. Its two concrete applications are demonstrated in the keystroke integrity verification and malicious traffic detection.**

Specifically, we first design and implement an efficient cryptographic protocol that enforces keystroke integrity by utilizing on-chip Trusted Computing Platform (TPM). The protocol prevents the forgery of fake key events by malware under reasonable assumptions. Then, we demonstrate our provenance verification approach by realizing a lightweight framework for restricting outbound malware traffic. This traffic-monitoring framework helps identify network activities of stealthy malware, and lends itself to a powerful personal firewall for examining all outbound traffic of a host, which cannot be bypassed.

Keywords: Authentication, malware, cryptography, provenance, networking.

I. Introduction

Compared to the first generation of malicious software (malware) in late 1980's, modern attacks are more stealthy and pervasive. Network- or host-based signature-scanning approaches alone were proven inadequate against new and emerging malware [15]. We view malicious bots or malware in general as entities stealthily residing on a human user's computer and interacting with the user's computing resources. For example, the malware may issue network calls to send outbound traffic for denial-of-service attacks, spam, or botnet command-and-control. However, conventional operating systems typically allow flexible execution pathways and data flow patterns, and are not specifically designed to distinguish legitimate user-initiated networking or file-system activities from malware-triggered ones.

Our goal is to improve the trustworthiness of the OS-level data flow; specifically, we provide mechanisms that ensure the correct origin or provenance of critical system data, which prevents adversaries from utilizing host resources (e.g., networking API). We define a new security property – *data-provenance integrity*. It states that the source from which a piece of data is generated can be verified. We give the concrete illustration of how data-provenance integrity can be realized for system-level data – namely keystroke events and outbound network packets – in a host-based setting.

For outbound network packets, we deploy special cryptographic kernel modules at strategic positions of a host's network stack, so that packets need to be generated by user-level applications and cannot be injected in the middle of the network stack. We implement our solution and demonstrate its low overhead. The significance of network-packet provenance is that one can deploy a sophisticated packet monitor or firewall at the transport layer such as [34] *without* being bypassed by malware – malware bypassing or disabling transport-layer personal firewalls is a typical problem for PCs.

We illustrate how to sign and verify keystroke events that are from external keyboard devices in a client-server architecture, i.e., verifying the provenance of keystrokes. We discuss the application of this system for distinguishing user inputs from malware inputs, which is useful in many scenarios including keystroke-dynamics based authentication. Our method has general application beyond the specific keystroke and network traffic problems studied.

Our Contributions We present a new cryptographic provenance verification (CPV) approach, and demonstrate its applications in realizing *i)* keystroke-integrity service, and *ii)* robust host-based traffic-monitoring. We apply basic cryptographic mechanisms to ensure the

A preliminary version of this work appears in the Industrial Track of the *Eighth International Conference on Applied Cryptography and Network Security (ACNS '10)* – Industrial Track papers were not archived in ACNS's LNCS proceedings. This work has been supported in part by DIMACS REU programs, NSF grant CCF-0728937, CNS-0831186, CAREER CNS-0953638.

Yao is the corresponding author. Her email address is danfeng@cs.vt.edu and mailing address is 2202 Kraft Dr. KWII, Virginia Tech, Blacksburg VA 24060. Yao, Xu, and Huijun are with the Department of Computer Science at Virginia Tech. Stefan is with Stanford University. Wu is with AppFolio.com. Work done by Stefan and Wu when they were a visitor and a student at Rutgers University, respectively.

correct data flow and system properties of a host, especially on verifying the provenance of dynamic system-related data. We describe how to integrate cryptographic components with operating systems and how to use hardware tools for the integrity of cryptographic keys in our provenance verification operations. Our contributions are summarized as follows.

1) We propose the security model and operations in cryptographic provenance verification. We describe its important applications in ensuring dynamic data flow in hosts, in particular system-related data and properties. We point out the technical challenges in realizing cryptographic provenance verification.
2) We illustrate our cryptographic provenance verification in the design of a keystroke integrity service that utilizes the hardware Trusted Platform Module (TPM). We construct a lightweight cryptographic protocol that prevents malicious bots from injecting keystroke events into a host's applications. This keystroke integrity service also prevents certain types of tampering on the host's kernel. We implement our prototype with an enabled on-chip TPM, and experimentally evaluate both the computation and communication overheads. Our cryptographic operations have low computation overhead ($<<$ 1ms for each keystroke) and reasonable bandwidth overhead (12KBps upper bound).
3) We apply our cryptographic provenance verification approach in realizing a host-based traffic-monitoring framework. The framework is capable of detecting stealthy outbound traffic of OS-level malware by enforcing the provenance verification for outbound network packets. Malware traffic that bypasses normal network-stack functions can be effectively detected.
   We describe our experimental evaluation with two proof-of-concept pieces of malware. When running 20 Windows network services and 15 networked applications for 7 days, our traffic-provenance verification mechanism triggered no false alarms. Our throughput validation on upstream network traffic shows that for 64 KB packet size the overhead for cryptographic operations is low.

Our work enables the authentication of two important data types: user inputs and network flow. Such a framework can be used to realize the temporal correlation between user inputs and network traffic under more powerful malware than what was considered in [5], [14]. In addition, our work can also enable host-based semantic-based correlation analysis between inputs and network packets. By verifying the data provenance, we are able to regulate the way data flows within a system. This regulation restricts activities that malware may perform on a host.

Organization of the Paper: Related work is described in Section II. We give an overview of our cryptographic provenance verification (CPV) approach and our security models in Section III. To illustrate our host-based provenance verification approach, in Section IV we present a cryptographic protocol that ensures that users' keystroke events cannot be forged by malware. In Section V, we describe a cryptographic traffic-monitoring framework and also demonstrate its effectiveness in catching OS-level malware traffic. We conclude the paper in Section VI.

II. Related Work

Information flow control has been an active research area in computer security. As early as in the 70s, Denning *et al.* [6], [7] has proposed the lattice model for securing the information flow and applied it to the automatic certification of information flow through a program. Data tainting, as an effective tracking method, is widely used for the purposes of information leak prevention and malware detection. Taint tracking can be performed at different levels, for example within an application [8], within a system [37], or across distributed hosts [16]. Our use of TPM as a signature generator may be viewed as a special type of data tainting. In addition to conventional taint tracking solutions such as hardware memory bit or extended software data structure, our TPM-based solution uniquely supports the cryptographic operations to enforce data confidentiality and the *integrity* of taint information. The important feature about TPM is its on-chip secret key. Therefore, the client device can be uniquely authenticated by a remote server.

Our paper focuses on a host-based approach for ensuring system-level data integrity and demonstrates its application for malware detection. In comparison, network trace analysis typically characterizes malware communication behaviors for detection [11], [12], [13], [23]. Such solutions usually involve pattern-recognition and machine learning techniques, and have demonstrated effectiveness against today's malware. Traces of botnets' command-and-control (C&C) messages – i.e., how bots communicate with their botmasters – are captured and their signatures and access patterns analyzed. For example, a host may be infected if it periodically contacts a server via the IRC (Internet Relay Chat) protocol and sends a large number of emails afterward. Network traffic analysis can be realized by local Internet Service

Providers to monitor and screen a large number of hosts as part of a network intrusion-detection system.

The element of human behavior has not been extensively studied in the context of malware detection, with a few notable exceptions including solutions by Cui, Katz, and Tan [5] and Gummadi *et al.* [14]. They investigated and enforced the temporal correlation between user inputs and observed traffic. BINDER [5] describes the correlation of inputs and network traffic based on timestamps. It does not provide any security protection against the detection system itself, e.g., how to prevent malware from forging input events. Our work provides a hardware-based integrity service to address that problem. In comparison to NAB [14] which is designed specifically for browser input verification, our work provides a more general system-level solution for keystroke integrity that is *application-oblivious*.

Existing rootkit detection work includes identifying suspicious system call execution patterns [4], discovering vulnerable kernel hooks [32], exploring kernel invariants (e.g., Gibraltar [1]), or using a virtual machine to enforce correct system behaviors [9], [24]. For example, Christodorescu, Jha, and Kruegel collected malware behaviors like system calls and compared execution traces of malware against benign programs [4]. They proposed a language to specify malware behavior and an algorithm to mine malicious behaviors from execution traces. A malware analysis technique was proposed and described based on hardware virtualization that hides itself from malware [9]. Wang *et al.* systematically identified potential kernel hook points in Linux kernel [32]. Although existing OS level detection methods are quite effective, they typically require sophisticated and complex examination of kernel instruction executions.

To enforce the integrity of the detection systems, a virtual machine monitor (VMM) is usually required in particular for rootkit detection (e.g., [24]). In this paper, we utilize the existing trusted computing infrastructure. TPM is available on most commodity computers. The advantage of using TPM in comparison to VMM is the ease of accessing a host's kernel, and the ability to construct *application-level* fine-grained detection solutions, as described in the future work section. How to address the run-time limitation of TPM is still an active research topic. For example, Flicker is a recently-proposed trusted computing base that allows sensitive applications to run in isolation in an untrusted operating system [19]. In comparison, our trusted computing architecture supports functions beyond application integrity including enabling remote collection and verification of dynamic system data such as user-input events.

TPM has also been used for providing a secure passage for sensitive user data such as passwords in *BitE* [20] and *Bump* [21]. These two systems encrypt keystrokes in order to prevent attackers' keyloggers from learning secret personal information. Although we use TPM for ensuring system integrity as in Bump and BitE, the goal of our solution is different — our work verifies the keystroke origin and prevents malware injection of fake key events.

The work by Srivastava and Giffin [30] on application-aware blocking of malware traffic may bear superficial similarity to our solution. They used a virtual machine monitor (VMM) to monitor application information of a guest OS without using any cryptographic scheme.

III. Models and Definitions

We define cryptographic provenance verification as a robust mechanism that ensures the true origin of the data produced by an entity such as a system device or a program. Such a system can be implemented by certifying (i.e., signing) the data generated at the source. However, our provenance verification has a fundamental difference from the traditional cryptographic signature scheme. In most signature schemes the signer is assumed to be a *person* who exercises discretion in signing documents and also in protecting his or her signing keys. In the context of malware detection, the signer and verifier are programs, e.g., kernel modules, which may be fooled or tampered with in the certifying process. Prevention against these attacks is critical. The techniques in cryptographic provenance verification are also different from the language-based or policy-based tainted inference analysis [26], as we emphasize the *enforcement* of normal system properties with lightweight cryptographic primitives and trusted computing infrastructure.

Security Goal: We aim to prevent unauthorized use of a personal computer by a malicious bot (or by an individual who is not the owner). Specifically, our goal is to address the following important question: *Is the computer being used by the authenticated owner or by an intruder?*

Malware attack models We assume that malware actively makes outside connections for command & control or attacks. For example, malware may attempt to log user inputs, inject traffic bypassing the host's firewall, forge input events, tamper with network traffic, modify kernel modules and file systems, access secret keys of the detection framework, tamper with the browser or P2P client. We consider both application-level malware and limited kernel malware. Kernel malware differs from application-level malware – it is typically in the form of drivers (for Windows) or loadable kernel modules

(for Linux). In our security model, we assume that the clean OS may be later infected by some kernel malware after loading, so that kernel-level malware may attempt to *i)* inject inputs to the system (keystroke injection), or *ii)* invoke low-level network stack interface to send outbound traffic (traffic-checkpoint bypassing). These behaviors allow certain types of malware to gain useful functions, such as circumventing security filters deployed at the network interface.

It is feasible to inject keystrokes directly into the keyboard buffer in both Windows and Linux. In keystroke-dynamics based authentication systems [31], collected keystroke sequences are physically entered to the keyboard device and cannot be replayed. The security critically depends on the integrity of keystroke events, which is uniquely offered by our technique. There exist other security mechanisms that rely on the integrity of keystrokes, in particular the line of work that analyzes the correlation between user inputs and system/network events for anomaly detection such as in [5].

Therefore, we aim to detect the two specific behaviors (traffic-checkpoint bypassing and keystroke injection) under the assumption that our detection system is not compromised at the run time (See security assumptions next). Our load-time integrity is guaranteed by TPM attestation. For the Linux operating system, we assume that the malware may attempt to inject keystrokes at the application level or at the kernel level. Specifically, the buffers lower than the TTY driver are not corrupted, which include the keyboard event driver and the keyboard device driver, and the on-chip storage on hardware controllers are secure. For network-traffic monitoring, malware may attempt to send traffic by directly invoking functions at the network-layer, but not at the lower-level data-link or physical layers. The purpose of these restrictions on the malware behaviors is to accurately reflect our security guarantees offered. We write and evaluate several such malware in our work.

Security assumptions We assume that the on-chip Trusted Platform Module (TPM) is tamper-resistant; the cryptographic operations are implemented correctly; and the remote server is trusted and secure. TPM provides the guarantee of load-time code integrity. It does not provide any detection ability for run-time compromises such as buffer overflow attacks [10]. Thus, we assume that the kernel code and our detection modules are not tampered with at the run-time. Advanced attacks [2], [33] may still be active under this assumption, indicating the importance of our solutions.

It is worth mentioning that virtualization based introspection technique pioneered by Payne and Lee [22] may be used to relax the assumption on kernel integrity in the host-based malware detection. Introspection allows the isolation of the detection engine, i.e., virtual machine monitor (VMM), separated from the guest OS being monitored to ensure the integrity of VMM.

We consider two types of data under the context of provenance verification on a host as follows. We also introduce the concepts of data producer and data consumer in our model.

- *Application data*: data generated by an application that may be a direct or indirect result of user actions. E.g., outbound network requests of an application, or file system requests from an application. The data producer is the application; the data consumer is the kernel or another application.
- *Kernel data*: data generated by the operating system that may be consumed by other kernel components or applications. E.g., keyboard and mouse events, incoming network packets, or file I/O events. The data producer is a kernel component; the data consumer is another kernel component or an application.

We define three operations for data-provenance verification on a host: *setup*, *sign* and *verify*.

- *Setup*: the data producer sets up its signing key k and data consumer sets up its verification key k' in a secure fashion that prevents malware from accessing the secret keys.
- *Sign*(D, k)*:* the data producer signs its data D with a secret key k, and outputs D along with its proof sig.
- *Verify*(sig, D, k')*:* the data consumer uses key k' to verify the signature sig of received data D to ensure its origin, and rejects the data if the verification fails.

Although simple, the cryptographic provenance verification method can be used to ensure and enforce correct system and network properties and appropriate workflow under a trusted computing environment. Next, we illustrate two such applications in Section IV and V for enforcing the origin of keyboard inputs and outbound packets of a host, respectively.

IV. Provenance Verification For Keystroke Integrity

In this section, we demonstrate our cryptographic provenance verification approach in realizing an efficient framework for ensuring the keystroke integrity in a client-server architecture. The need for verifying the keystroke integrity is motivated by the line of existing security solutions based on user inputs, for example keystroke-dynamics authentication [31], the user-behavior based drive-by download detection [18], [36],

and causality inference among user actions and network traffic [5].

With our solution, keyboard events entered by human users from the external keyboards are uniquely identified and their provenance is cryptographically verified. Fake inputs injected by malware – such as in attacks against keystroke-dynamics authentication or click-fraud attacks – can be detected. Our protocol utilizes lightweight cryptographic functions, and our key management leverages the on-chip TPM. We use the TPM to ensure the secrecy of signing and verification keys, as well as the integrity of a host's kernel and framework modules.

The TPM attestation [28] is useful in proving the load-time kernel integrity. However, it is worth noting that the TPM alone is not sufficient in preventing the injection of fake key events, as these type of attacks can originate from *applications* and are thus beyond the kernel integrity. For example, any X application can inject events without any communication with the keyboard driver. Our keystroke-integrity service addresses these application-level attacks efficiently. An existing approach (as in SATEM [35]) to prevent application-level attacks, e.g., substituting libraries with compromised versions, is to have applications as part of the trusted system that gets loaded and attested by TPM. In comparison to the SATEM approach [35], our architecture is more specific to the key-event integrity problem and thus is simpler.

Besides injecting fake inputs, strings sent to function calls may be modified by attackers through interception. An attacker may also attempt to record user keystroke events and replay them at a later point. Our integrity verification method can detect both of these attacks – the replay attack can be detected due to the mismatch in timestamps, and the modification attack can be detected due to the failed verification of digital signatures. This detection of replay attacks assumes that the system clock used to record timestamps is secure and accurate, and the recorded timestamps of keystroke events cannot be tampered with before they are signed. Both assumptions are valid in our security model in Section III.

A. Architecture For Keystroke Integrity

The goal of our design is to establish a secure channel between the host and a remote server during the initialization, and this channel is used for relaying packets. Each packet contains an encrypted version of the keystroke event and its signature. The origins of signed keystrokes are proved by TPM and authenticated by the server. Our prototype includes a trust agent in kernel, a trust client in user-space, and a remote trusted server.

- The *trust agent* is a kernel module that signs each keystroke event. This client-side trust agent and the remote trusted server share a secret session key. The keystroke events are obtained from the keyboard event driver by registering a keyboard notifier.
- The *trust client* is a user-space program that forwards messages between the (kernel-level) trust agent and the remote server.
- The *remote server* is assumed to be trustworthy and verifies the signatures of keystrokes. The server also monitors the client's boot-time integrity through TPM-based operations, including its kernel and components. In case such a remote server is not available, we give alternative solutions at the end of Section IV.

Our main operations include: *i)* key exchange between the trust agent and remote server; *ii)* keystroke signing by the trust agent; *iii)* relaying signed events by the client to the remote server; and *iv)* verifying the boot-time kernel integrity by the remote server. Our prototype is implemented in a client-server architecture in Linux using the Intel Integrated TPM, details of which are described in Section IV-C. This integrity service can defend against malware attacks such as the replay of prior-captured user keystrokes, fake key-event injections, and tampering with our client. This keystroke-integrity service can also be applied to ensure the integrity of mouse events. The capability of ensuring user-input integrity and preventing malware forgery of input events has general applications and serves as a fundamental component in constructing security systems with trusted user inputs. Next, we give the technical details in our design and implementation.

B. Key Management in Keystroke Integrity Service

The key management in our keystroke integrity service is based on TPM specifications. However, directly using the on-chip master secret key for cryptographic operations is slow for time-sensitive applications such as ours. Therefore, we generate a set of secret keys from the master key for our cryptographic operations in order to improve the efficiency. Our design involve creating three private/public RSA key pairs: a *signing key*, a *binding key*, and a *storage key*. The signing key is used to sign and encrypt outbound packets as well as TPM quotes. TPM quote refers to the hash values stored in the platform configuration registers (PCR) of the TPM. These hash values are computed at the boot time – they are chained digests of the system code that is loaded into the memory of the host. The TPM quote cannot be forged or tampered with [1]. The binding key is

[1]We do not consider physical attacks (such as described in [29]) against the TPM in this paper.

used to securely store the signing key. The storage key is similar to the binding key but is more general, as the latter is specifically for storing symmetric keys whereas the storage key does not have any constraints. We use the storage key to secure both the binding key and signing keys. Our key management mechanism leverages the on-chip TPM secret key to derive additional keys. We use TPM's sealed storage, which provides a secure location to store secret keys, until system integrity can be verified. Thus, the secrecy of keys is guaranteed, and the efficiency of kernel-level cryptographic operations is also largely improved. Our key exchange is as follows.

1) The trust agent uses the TPM to generate two pseudorandom numbers (a_0, a_1). The trust agent generates the TPM quote and uses the signing key to sign it. The generated data in this step are encrypted using the server's public key.
2) The server generates two random numbers (b_0, b_1) and encrypts them using the public key of the trust agent.
3) Server and trust agent exchange the encrypted random numbers and XOR the decrypted values with the sent bits to use as two symmetric keys (e.g., $a_0 \oplus b_0$, $a_1 \oplus b_1$), using one key for signing, and the other for encryption; this key exchange protocol follows from [25]. Finally, the server verifies the values of the TPM quote against the correct quote values, and also verifies its digital signature.

When the trust agent disconnects, the binding key is used to bind the symmetric keys and securely store them so the key exchange is not required during the next connection; the server requests a new key exchange when necessary (after a certain number of messages are exchanged). The secrecy of keys is guaranteed, as they are encrypted (and stored on the hard disk) with on-chip TPM key when not used. The TPM quote procedure is repeated at each boot time. Our implementation follows the Trusted Computing Group (TCG) Service Provider Interface. This interface is part of the TCG Software Stack. The TCG Service Provider (TSP) layer is on top of the TCG Trusted Software Stack (TSS) core service layer which interfaces with the TCG device driver library used to communicate with TPM chip.

C. Implementation of Keystroke Integrity Service

Our prototype is implemented in the Linux operating system using a Lenovo Thinkpad T400 with Intel Integrated TPM and Intel Core 2 Duo (INT-C0-102, 2.53GHz, 6MB cache, 1066MHz FSB), 3GB RAM, following TPM Interface Specifications 1.2. We implement the trust agent as a Linux kernel module and the trust client as an application. The trust client parses the non-encrypted messages and forwards them between the kernel-level trust agent and remote server. Based on the Linux kernel cryptographic API, we implement cryptographic functions on key events, including signing key events by the trust agent and verifying key events by the remote server. We provide the encryption and decryption functions on the packets from the client to the remote server to prevent network snooping of keystrokes. Last but not least, we implement the key management mechanism for the integrity service that leverages TPM storage keys, as described in Section IV-B.

We describe the detailed procedure of starting and running the integrity service between the client and the remote server as follows.

1) **Trusted boot:** A kernel module, which we call *trust agent*, is loaded on boot or can be compiled in the kernel. The module creates a device `/dev/cryptkbd`. We disable `/dev/kmem` and module loading after boot to prevent any tampering with the agent. A user-space *trust client* opens the device `/dev/cryptkbd` and concurrently opens a socket to the trusted server, waiting for communication. When the trust client opens the device `/dev/cryptkbd`, the trust agent – which has the signature of the correct client program – verifies the integrity of the client. This verification also prevents any other program from opening the device in our security model.
2) **Initial authentication:** When the remote server gets a connection from a client, it requests the initial attestation. The trust client uses the `write` system call to request the required information from the agent. The trust agent forwards the signed TPM quote. The trust client forwards the information to the server which verifies the information.
3) **Key exchange and monitoring:** The trust agent and the remote server set up a shared key through a RSA key exchange protocol based on the TPM keys. At the trust agent, each keystroke event is encrypted and a corresponding signature is generated. Both the encrypted event and its signature are wrapped in a packet. The trust client forwards the packets to the remote server which verifies the integrity of events by checking the signatures using a keyed hash function. If signatures associated with events do not pass the server's verification, then the trust agent is notified.

D. Performance Evaluation of Keystroke Integrity Service

To estimate the bandwidth overhead, we assume that a fast typist enters 212 words per minute [3] and in English an average word has 4.5 characters [27]. Each character has a press event and a release event, respectively. Therefore, we obtain 12.2 KBps maximum bandwidth overhead as follows.

$$\frac{212 \text{ words}}{60 \text{ sec}} \times 4.5 \frac{\text{chars}}{\text{word}} \times 2 \frac{\text{events}}{\text{char}} \times 384\text{B} = 12.2\text{KBps} \quad (1)$$

We experimentally evaluated the overhead incurred by the event signing and encryption in the keystroke-integrity service. The purpose is to measure the overhead associated with TPM encryption and authentication components. We collected data from 10 participants in a user study. The participants continuously typed characters of their choice for 5 minutes, with our program running in the background. We asked the participants to type actively, in order to evaluate the overhead incurred.

We compute the average processing time over 28,612 keystroke events with the TPM key initiation amortized. Each key press and key release event is in a separate packet of 384 bytes. Signing a packet with a 256-bit key takes 33.5 microseconds, and encrypting a packet using standard AES-CBC with a 256-bit key takes 50.6 microseconds on average. The entire keystroke callback processing takes 1.30 milliseconds.

We compute the communication overhead between trust client and authentication server during the user study. Packet relay is measured and shown in Table I. At the peak input-speed of a user, 153 packets are sent during a 10-second interval, which gives a bandwidth consumption of 5.87 KBps. The average bandwidth consumption observed is 2.92 KBps. The distribution of all the 10-second intervals over the possible bandwidth consumption is shown in Figure 1. Throughout the user study, participants had no visible delay or any usability issues. The cryptographic operations introduced by this integrity service have low computational and communication overheads. The bandwidth utilization can be further optimized, for example, by grouping multiple keystroke events in one packet.

Attack simulation We implement a program in C which injects keyboard events in order to simulate keystroke forgeries. Our program creates fake keyboard events and injects them into the *core-event-stream* as if they are typed on the external keyboard. The core-event-stream runs above the TTY-level driver and can be viewed as an application-level service. From the application's perspective, the fake keyboard events cannot be distinguished from actual key events, even though the keyboard is not touched. Using our integrity service we confirm that forged keystroke events are recognized as rogue.

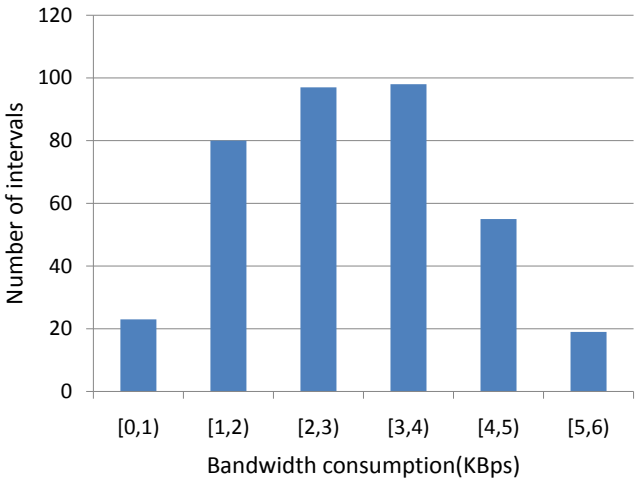

Fig. 1. The distribution of bandwidth consumption over 10-second intervals.

Summary Directly using on-chip master secret keys for cryptographic operations is slow. In comparison, our implementation generates secret session keys from the on-chip master key and uses session keys for the cryptographic operations, which speeds up the signing and encryption operations on the client side. Our protocol ensures the authentic origin of keystroke events, and embodies our provenance verification approach. It also yields a general approach that can be used for the attestation of other devices. In particular, it can be developed to prevent malware from injecting fake events into other applications. One may expand the TPM support for the applications to be included in the attestation.

The remote trusted server is for the verification of the load-time system integrity and run-time keystroke integrity of the host. If there is no remote server available, both types of verification need to be realized locally with additional security assumptions. The TPM `seal` and `unseal` operations can be used to protect the integrity and secrecy of signing keys – the keys are accessed or unsealed, only when the host's load-time quote matches the quote when the key is stored or sealed. One needs to add to the host a new kernel module that verifies the signatures of keystrokes. This new component's load-time integrity is verified as part of the quote. One needs to assume that it cannot be tampered with at the run-time.

In the next section, we describe a traffic provenance verification mechanism for a host to monitor all outbound packets including those sent by stealthy malware.

(KBps)	Maximum	Minimum	Average
Per person	4.21	1.14	2.95
Per 10-second interval	5.87	0.34	2.92

TABLE I

NETWORK BANDWIDTH CONSUMPTION IN OUR KEYSTROKE-AUTHENTICATION USER STUDY.

V. TRACKING PROVENANCE OF OUTBOUND TRAFFIC

In this section, we illustrate our cryptographic provenance verification approach in a network setting, in particular for ensuring the integrity of outbound packets, as they flow through the host's network stack. We describe the design and implementation of a lightweight traffic-monitoring framework. It can be used as a building block for constructing powerful personal firewalls or traffic-based malware detection tools. Malware disabling or bypassing personal firewalls on a host renders the firewalls useless.

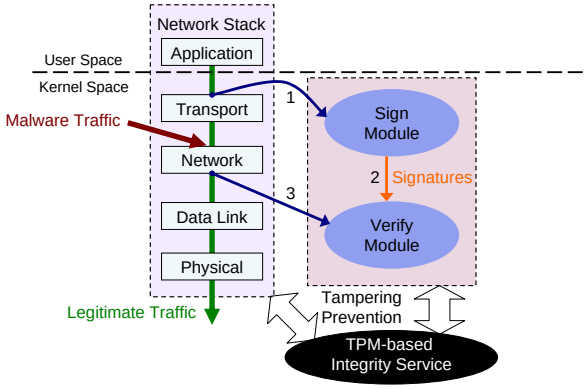

Fig. 2. Schematic drawing of components in the framework and their interactions with the host's network stack. Legitimate traffic origins from the application layer whereas kernel-level malware traffic may be injected to the lower layers. Traffic checkpoints are placed at the Sign and Verify modules.

We demonstrate the effectiveness of our traffic-monitoring framework in identifying the network activities of stealthy malware. Specifically, our provenance verification scheme requires outgoing network packets to flow through a checkpoint (e.g., a kernel module) on a host, to obtain proper provenance proofs for later verification. Any traffic sent through disabling or bypassing the firewall can be detected, as the packets are unable to provide their provenance proofs. Thus, we effectively prevent any traffic to be sent without passing through a certain checkpoint – significantly improving the assurance of traffic-based malware detection on hosts. Such a simple-yet-powerful traffic-monitoring framework can yield advanced detection mechanisms such as input-traffic correlation analysis on application-level traffic such as [32]. In particular, our solution provides an effective defense against common bypassing attacks.

Malware may communicate with the outside world, with the intent of exporting sensitive data. Legitimate outbound network traffic passes through the entire network stack in the host's operating system. We develop a robust cryptographic protocol for enforcing the proper *provenance of a packet* on a host.

A. Architecture of Traffic Provenance Verification

The Internet protocol stack or network stack is part of the host's operating system and consists of five layers – application, transport, network, data link, and physical layers. User-space outbound traffic (e.g., browser or email packets) travel all five layers on the stack from the top to the bottom before being sent out. System services (e.g., Windows updates) are typically implemented as applications, thus their network flow also traverses the entire Internet protocol stack (which we validate with experiments).

Our design of the traffic-monitoring framework extends the host's network stack and deploys *two* kernel modules, *Sign* and *Verify* modules, as illustrated in Figure 2. Both signing and verification of packets take place on the same host but at different layers of the network stack – the Sign module is at the transport layer, and the Verify module is at the network layer. The two modules sharing a secret cryptographic key monitor the integrity of outbound network packets. All legitimate outgoing network packets first pass through the Sign module, and then the Verify module. The Sign module signs every outbound packet, and sends the signature to the Verify module on the same host, which later verifies the signature with a shared key. The signature proves the provenance of an outgoing packet. If a packet's signature cannot be verified or is missing, then the packet is labeled as suspicious. Directly invoking the lower data-link layer or the physical layer functions to send traffic is hardware-dependent and difficult in practice.

Therefore, installing the Verify module at the network layer is adequate.

For the key management, when the system starts up, the Sign module and the Verify module generate their respective public/private key pairs and notify each other of their respective public keys. Then the two modules securely exchange a symmetric key, which is used for signature generation and verification. In the Windows OS, modules communicate by sending I/O request packets (IRPs). `IoCallDriver` is the function for sending IRPs to a specific driver. Specifically, the Verify module sends a random number encrypted with Sign module's public key and the Sign module replies with another random number encrypted with the Verify module's public key. The XOR result of the two numbers is the symmetric key which is used to sign and verify network packets. Our protocol has three main operations: SYSTEM BOOT, KEY EXCHANGE, and SIGN AND VERIFY.

SYSTEM BOOT: After the Verify module is started, it randomly generates a public/private key pair. Then, the Sign module is started, which randomly generates a public/private key pair.

KEY EXCHANGE:

1) The Sign module initiates the connection with the Verify module. The two modules exchange their public keys. The Sign module generates two random numbers a_0 and a_1, and encrypts a_0 and a_1 using the Verify module's public key. The Sign module sends encrypted a_0 and a_1 to the Verify module.
2) The Verify module receives and decrypts a_0 and a_1 with its private key. It then generates two random numbers b_0 and b_1. The Verify module encrypts b_0 and b_1 using the Sign module's public key. The Sign module decrypts them with its private key.
3) Both the Sign and Verify modules have a_0, a_1, b_0, and b_1. They compute the signing key as $a_0 \oplus b_0$ and the symmetric key for their communication encryption as $a_1 \oplus b_1$.

SIGN AND VERIFY:

1) The Sign module gets the packet payload from the application layer and generates a signature for the data using UMAC. The Sign module encrypts the signature and packet information (e.g., source and destination addresses and ports) with the communication key.
2) The Sign module sends the encrypted data to the Verify module. The Verify module decrypts the received information, and inserts the records into a hash table indexed by the packet information.
3) The Verify module intercepts every network packet before it is sent to the network interface card and computes its signature. The Verify module then retrieves the stored signature corresponding to the packet from the hash table. If the retrieved signature matches the one computed, then the corresponding packet is allowed to be passed down to the next layer. Otherwise, the Verify module reports the packet as being suspicious.

In our evaluation in Section V-D, we measure the overhead associated with signing the entire packet payload or part of the payload (referred to as partially signed packets). To ensure the integrity of the detection framework at load time and signing key secrecy, we may utilize the on-chip TPM to generate the signing keys and to attest kernel and module integrity at boot. The detail is similar to keystroke-integrity service described in Section IV, where the attestation of kernel and module integrity may use a remote trusted server. Enlisting a remote server for integrity purpose was also previously used in [1]. In comparison to the virtualization-based traffic detection approach by Srivastava and Giffin [30], our solution provides an effective cryptographic alternative that leverages the available trusted computing infrastructure.

B. Windows Network Architecture and Issues With Firewalls

We have implemented our traffic provenance-tracking mechanism in Windows XP, and experimentally evaluated it with two proof-of-concept malware and assess the throughput of upstream network traffic. Windows personal firewalls typically work in Winsock SPI, TDI, or NDIS layers from the top to the bottom. The Winsock SPI (Service Provider Interface) layer is in the user level. The TDI layer is typically used in commercial host-based security solutions. At this layer, the firewall may add a filter device in the kernel to filter outbound network packets. The firewall at the TDI layer has the access to the process information, which provides useful contextual data for filtering packets. However, TDI filter devices can be disabled or bypassed, which renders the firewall useless. The NDIS layer captures all network traffic, and is hard to bypass firewalls at the NDIS layer. However, it is difficult to retrieve high-level process information about network packets at the NDIS layer.

We note that the keystroke integrity and traffic provenance of our work are independent of each other – each illustrating an application of our proposed CPV approach. Due to the internal organization of Linux OS, its network stack is not as modular as Windows network stack, specifically, there is no clear-cut boundary

between the transport layer and network layer. Thus, we opt for Windows OS for building our traffic-provenance prototype, which is presented next.

C. Prototype Implementation

Our implementation is realized in C/C++ in Windows XP operating system. The Sign module is realized as a TDI filter device and the Verify module is realized as part of the NDIS driver. In our prototype, we use UMAC (message authentication code using universal hashing) in lieu of a public-key digital signature scheme for proving message integrity due to UMAC's efficiency and simplicity [17]. Similar to Section IV, TPM may be used to ensure the secrecy of signing keys and the integrity of the operating system at the load time, which is not implemented for this prototype for traffic-provenance verification. Due to the lack of access to Windows XP kernel cryptographic library, we implement our own cryptographic operations.

Sign module captures and signs network packets that flow through the Winsock API. Our implementation is based on an open-source TDI filter device in the kernel space called *TDIFW*. It is a Windows kernel driver and can filter all network traffic from the application layer. We modify the function `tdi_send` to support the signing operation. This function intercepts outbound network packets including UDP datagrams and TCP segments. All TCP control messages such as `SYN`, `FIN`, `ACK` packets are not captured by Sign module, because they are generated by the TCP/IP stack which is below our filter device. The process ID can be learned by calling `PsCurrentProcessID`, since the filter driver works in the context of the process which calls Winsock APIs to send outgoing packets.

In function `tdi_send`, we capture all TCP segments, and compute the message authentication code for each segment. Signatures are sent directly to the Verify module through the system call `IoCallDriver`.

Verify module is an NDIS intermediate miniport driver. It intercepts and verifies all packets just before they are sent to network interface card drivers. The Verify module is implemented on the *Passthru*, which is a NDIS intermediate miniport device in Windows. We modify the function `MPSendPackets` to support the verification operation. TCP control messages are put on the white list and are not subject to our verification test.

Our prototype does not verify any TCP control packet, but signing TCP control packets may be done if the Sign module is placed at a lower level on the network stack. The SYN flood attack is a problem in TCP where the attacker exhausts the resource of a victim server by sending many SYN packets and resulting in many half-opened connections. Our traffic provenance service aims at enforcing the regulation of the network stack utilization and is not directly related to the prevention of SYN flood attack. Server-side solutions have been proposed such as using the SYN cookie and adaptive timeout to defend against the SYN flood.

Intuitively, the Verify module at the network layer has to reassemble Ethernet frames in order to reconstruct the original transport layer data segments and then compute signatures. Fortunately, because UMAC computes signatures incrementally and outgoing Ethernet frames in the network stack are sequential, the Verify module does not need to reassemble fragments. It updates the corresponding signature for each fragment on-the-fly, which significantly reduces the time and memory costs. It is important to note that the packet signature is *not* appended to each packet, as this would result in unnecessary checksum recalculations by the Verify module. Instead, the Sign module sends UMAC directly to the Verify module, as shown in Figure 2. UMAC values are kept in a hash table indexed by the packet's source address, destination address and port for the fast look-up.

D. Performance Evaluation

To evaluate the running time, we use `jperf` to generate workload, which is a wrapper of `iperf` with a graphic user interface. For each setup we performed multiple runs ($\geq$ 3) and compute the averaged time. Data is collected while the throughput is stabilized. We use a Windows virtual machine as the destination, i.e., `iperf` runs on the host machine and generates traffic to the virtual machine. The secret key is hard-coded in our experiments. The key length is 128 bits (default for UMAC).

Our experimental evaluation in Figure 3 shows that the overhead imposed by the cryptographic integrity verification on the outbound traffic streams is minimal when transport-layer segment size is large (e.g., 64KB). For small packet sizes (which are typical in web browser traffic), Table II shows the throughput with UMAC divided by the throughput without UMAC. The slow down observed in small packets is slightly higher than large packets due to our cryptographic overhead, but is still tolerable. In our experiments with the web browser and other common networked applications, there is no visible slow down observed by the user. Instead of signing the whole packet, an alternative is to sign a portion of the packet, which reduces the computation overhead and increases the throughput, as is shown in

Figure 3 (bottom). An attacker may attempt to tamper with the unsigned part of packet. We can randomize the choice of signed portion to make this attack more difficult. Furthermore, malware piggybacking its payload to other packets is useless for the attacker, unless the packets' destinations are modified to be the attacker's intended destination.

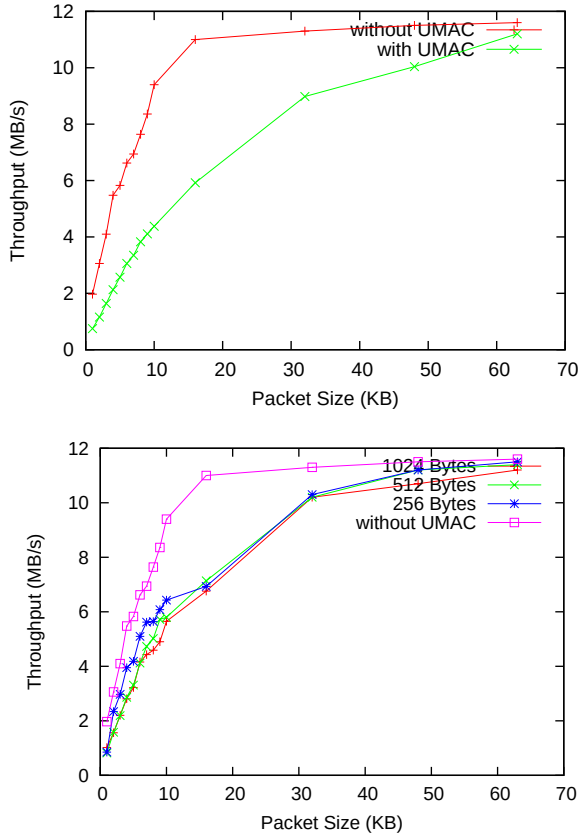

Fig. 3. Comparisons on outbound packet throughput with or without signing at the top and with partially signed packets at the bottom.

In summary, with the provenance verification on each packet, the throughout decreases in general. However, as the packet size grows, the costs of signing and verification are amortized. The observed performance degradation is acceptable in practice. Most personal computers have low upstream traffic even with peer-to-peer applications running. Our detection framework enforces the correct flow of outbound traffic through the host's network stack. This feature enables advanced traffic inspection solutions at the transport layer, without worrying about malware bypassing the inspection checkpoint. Installing sophisticated traffic inspection at the transport layer of a host is desirable, due to the ease of accessing user-space information.

Packet Size(KB)	Throughput (in %)
1	38.07%
2	37.90%
3	40.00%
4	38.86%
5	44.08%
6	46.22%
7	48.27%
8	50.13%
9	49.16%
10	46.59%

TABLE II
THE THROUGHPUT WITH UMAC AS THE PERCENTAGE OF THE THROUGHPUT WITHOUT UMAC FOR SMALL PACKET SIZES.

E. Two Proof-of-Concept Attacks Against Personal Firewalls

We develop two pieces of proof-of-concept malware that can turn off or bypass the typical transport layer firewall and send outgoing packets in Windows XP. We evaluate our prototype for traffic-provenance verification against them. The malware can disable URL filtering functionality of Trend Micro OfficeScan Client. In comparison, our Verify module can detect the network activities of such malware.

Bypass malware I. This experiment demonstrates an attack that attempts to disable the Sign module in our traffic-provenance verification mechanism. The attack can be detected by our Verify module because of the missing proofs due to the disabled Sign module. We implement a malicious driver with the ability to remove all the hooks attached to Windows OS TDI drivers without reboot. As a result, the Sign module is disabled. This attack is loaded dynamically by any Windows driver loader application and does not require restarting the machine to execute the malicious code.

We confirm that the Verify module is able to catch the outbound packets that do not have corresponding proofs immediately after the malware is launched.

Bypass malware II delivered through drive-by download. In this attack, malicious code is delivered to the victim's machine and executed through a drive-by-download exploit. In the drive-by-download [36], malicious code can be fetched and executed through compromised applications (such as the browser) on a victim's machine without the user's permission. This exploit is capable of removing the Sign module from the file system, thus disabling it after the machine restarts. Restarting the computer is necessary in order to remove the copy of the Sign module in the memory. Our traffic-provenance verification mechanism can successfully detect this attack.

The Verify module is part of the NDIS driver, which is an intermediate driver between the network driver interface in transport layer and network physical device. Disabling the Verify module would disable the entire NDIS driver, which cripples the basic network functions. In contrast, the Sign module places a hook to the TDI driver, which is much simpler to remove without affecting other parts of the network stack. Thus, our assumption on the integrity of the Verify module is valid in practice.

F. False Positive Evaluation of Traffic-Provenance Verification

This experiment is to find out empirically the number of false alarms triggered by our traffic-provenance verification. A false alarm may be due to a network service or an application that does not send outbound traffic through the typical transport-layer entry point. We ran common networked applications on a Windows 7 personal computer for 7 days. The applications include browsers (Chrome, Firefox, IE8), Skype, Yahoo Messenger, Gtalk, AIM, Windows Media Player, Real Player, Google Earth, eMule, Thunderbird, Wireshark, and TrendMicro Office Scan. We manually interacted with each application for a certain time period (30 minutes), such as using an instant message application to chat, using peer-to-peer file sharing application to download and upload files, and browsing webpages with browsers. We recorded the number of alerts that our traffic-provenance verification generates. The collected data is of size 12 MB.

None of the applications that we evaluated generates any false alarm. During the experiments, there are a total of 20 active Windows services (e.g., automatic update) with network activities running on the host. None of the Windows services triggered any alert either. Our results indicate that legitimate applications and network services typically follow the network stack to send outbound traffic, and thus our traffic-provenance verification has no false positives.

VI. Conclusions

We described a general approach for improving the assurance of system data and properties of a host, which has applications in preventing and identifying malware activities. Our host-based system security solutions against malware complement network-traffic based analysis. We demonstrated CPV's application in identifying stealthy malware activities of a host, in particular how to distinguish malicious/unauthorized data flow from legitimate one on a computer that may be compromised.

We made the following technical contributions in this paper. *i)* We proposed the model and operations of cryptographic provenance verification in a host-based security setting. We pointed out its important usage for achieving highly assured kernel data and application data of a host, and associated technical challenges. *ii)* We demonstrated our provenance verification approach in a lightweight framework for ensuring the integrity of outbound packets of a host. This traffic-monitoring framework creates checkpoints that cannot be bypassed by malware traffic. *iii)* We described an efficient TPM-based keystroke-integrity verification protocol in a client-server architecture that prevents malicious bots from forging keystroke events. This keystroke-integrity service serves as an important building block for the future construction of human-behavior driven security solutions.

References

[1] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *24th Annual Computer Security Applications Conference (ACSAC)*, 2008.

[2] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Symposium on Security and Privacy*, pages 246–251. IEEE Computer Society, 2007.

[3] B. Blackburn and R. Ranger. Barbara Blackburn, the World's Fastest Typist, 1999.

[4] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 5–14, New York, NY, USA, 2007. ACM.

[5] 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.

[6] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976.

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, July 1977.

[8] M. Dhawan and V. Ganapathy. Analyzing information flow in Javascript-based browser extensions. In *ACSAC*, pages 382–391. IEEE Computer Society, 2009.

[9] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.

[10] 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.

[11] J. Goebel and T. Holz. Rishi: Identify bot contaminated hosts by IRC nickname evaluation. In *Proceedings of the First USENIX Workshop on Hot Topics in Understanding Botnets*, April 2007.

[12] 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 USENIX Workshop on Hot Topics in Understanding Botnets*, April 2007.

[13] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[14] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: Improving service availability in the face of botnet attacks. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NDSI)*, 2009.

[15] M. G. Jaatun, J. Jensen, H. Vegge, F. M. Halvorsen, and R. W. Nergård. Fools download where angels fear to tread. *IEEE Security & Privacy*, 7(2):83–86, 2009.

[16] H. C. Kim, A. D. Keromytis, M. Covington, and R. Sahita. Capturing information flow with concatenated dynamic taint analysis. In *ARES*, pages 355–362. IEEE Computer Society, 2009.

[17] T. Krovetz. UMAC: Fast and Provably Secure Message Authentication. `http://fastcrypto.org/umac/`.

[18] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. BLADE: An attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of 17th ACM Conference on Computer and Communications Security*, 2010.

[19] 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.

[20] 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.

[21] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *NDSS*. The Internet Society, 2009.

[22] B. D. Payne and W. Lee. Secure and flexible monitoring of virtual machines. In *ACSAC*, pages 385–397. IEEE Computer Society, 2007.

[23] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. My botnet is bigger than yours (maybe, better than yours). In *Proceedings of the First USENIX Workshop on Hot Topics in Understanding Botnets*, April 2007.

[24] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In R. Lippmann, E. Kirda, and A. Trachtenberg, editors, *RAID*, volume 5230 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2008.

[25] B. Schneier and N. Ferguson. Practical cryptography, 2003.

[26] R. Sekar. An efficient black-box technique for defeating web application attacks. In *ISOC Network and Distributed Systems Symposium (NDSS)*, February 2009.

[27] C. Shannon. Prediction and entropy of printed English. *Bell System Technical Journal*, 30(1):50–64, 1951.

[28] S. W. Smith. *Trusted Computing Platforms: Design and Applications*. New York: Springer, 2005.

[29] E. Sparks. A security assessment of trusted platform modules, 2007. Undergraduate thesis.

[30] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 39–58, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] D. Stefan and D. Yao. Keystroke-dynamics authentication against synthetic forgeries. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, November 2010.

[32] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 21–38, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] J. Wei, B. D. Payne, J. Giffin, and C. Pu. Soft-timer driven transient kernel control flow attacks and defense. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 97–107, Washington, DC, USA, 2008. IEEE Computer Society.

[34] H. Xiong, P. Malhotra, D. Stefan, C. Wu, and D. Yao. User-assisted host-based detection of outbound malware traffic. In *Proceedings of International Conference on Information and Communications Security (ICICS)*, December 2009.

[35] G. Xu, C. Borcea, and L. Iftode. Satem: Trusted service code execution across transactions. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 321–336, Washington, DC, USA, 2006. IEEE Computer Society.

[36] K. Xu, D. Yao, Q. Ma, and A. Crowell. Detecting infection onset with behavior-based policies. In *Proceedings of the Fifth International Conference on Network and System Security (NSS)*, September 2011.

[37] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *In Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS)*, 2007.