Knowing Where Your Input is From: Kernel-Level Data-Provenance Verification *

Deian Stefan[1], Chehai Wu[2], Danfeng (Daphne) Yao[3], and Gang Xu[4]

[1] Department of Electrical Engineering
The Cooper Union
New York, NY 10003
deianstefan@gmail.com
[2] AppFolio, Inc.
55 Castilian Dr.
Goleta, CA 93117
wuchehai@gmail.com
[3] Department of Computer Science
Virginia Tech
Blacksburg, VA 24060
danfeng@cs.vt.edu
[4] AT&T
200 Laurel Ave
Middletown, NJ 07748
gangxu@att.com

Abstract. 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 malware traffic detection and keystroke-based bot identification. Specifically, we first demonstrate our provenance verification approach by realizing a lightweight framework for blocking outbound malware traffic. This traffic-monitoring framework leverages the differences in legitimate user-traffic and kernel-level malware-traffic, and provides a powerful checkpoint for examining all outbound traffic of a host, which cannot be bypassed. Then, we 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.

Key words: Authentication, malware detection, provenance, network packet

1 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

* This work has been supported in part by REU programs, NSF grant CCF-0728937, CNS-0831186, CNS-0953638.

signature-scanning approaches were proven ineffective against new and emerging malware [15]. As pointed out by Christodorescu *et al.* [5], the fatal weakness in these pattern-matching approaches is that they are purely syntactic.

We view malicious bots or malware in general as ghost 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 system typically allows flexible execution pathways and data flow patterns, and is not designed to distinguish legitimate user-initiated from malware-triggered networking or file system activities. SELinux is a Linux feature that supports access control policies and confines programs in their privileges of accessing system resources. SELinux requires manual specification of policies which may be cumbersome for users.

Our goal is to improve the trustworthiness of kernel-level data-flow path, specifically, we provide automatic mechanisms that ensure the correct origin or provenance of critical system data, which prevents adversaries from utilizing host resources (e.g., networking API). *Data-provenance integrity* is a security property defined by us. It states that the source where a piece of data is generated cannot be spoofed or tampered with. We give concrete illustration of how data-provenance integrity can be realized for kernel-level data, namely keystroke events and outbound network packets, in a host-based setting, through a cryptographic provenance verification technique.

For outbound network packets, we deploy special cryptographic kernel modules at strategic positions of a host's network stack, so that packets need to generated by user-level applications and cannot be injected in the middle of the network stack. We implement our solution in Windows operating system and demonstrate its low overhead even with large network workloads. The significance of network-packet provenance is two-fold: *i)* stealthy malware that hides its user-level presence may be detected when it attempts to communicate to remote servers, as this type of malware typically injects network traffic directly into network layer; and *ii)* we are able to deploy sophisticated packet monitor or firewall above or at the transport layer such as [30] *without* being bypassed by malware – malware bypassing transport-layer personal firewalls is a typical problem for PCs.

We also illustrate how to sign and verify keystroke events that are from external keyboard device 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. Our method has general application beyond the specific keystroke and network traffic problems studied, and can be used as a fundamental building block for constructing high-assurance mini operating system such as MINIX.

One of the technical challenges for host-based security is how to guarantee the integrity of detection system itself and prevent it from being tampered by advanced malware. We illustrate an effective hardware-based approach that leverages trusted platform module (TPM) to attest the system integrity includ-

ing the security modules used for provenance signing and verification. TPM has been used recently for providing a secure passage for sensitive user data such as passwords in *BitE* [20], which is later improved by *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 keystroke origin and prevents malware injection of fake key events. More related work is discussed in Section 5.

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

1. 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 kernel-level malware by enforcing the provenance verification for outbound network packets. Malware traffic that bypasses normal network-stack functions is not accompanied with provenance proofs and is effectively detected. We describe our experimental evaluation with real-world and synthetic rootkits. Our throughput validation on upstream network traffic shows that for 64 KB packet size the overhead for cryptographic operations is less than 5%.
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 host's applications. This keystroke integrity service also prevents tampering attacks 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 proposed cryptographic provenance verification mechanism is useful beyond keystroke integrity. We demonstrate the use of simple cryptographic mechanisms for ensuring other kernel-data integrity (namely network packet), and its effectiveness against rootkits. Our work enables the authentication of two important data streams: user inputs and network flow. Such a frameworks can be used to realize the temporal correlation under more powerful malware than what was considered in [6, 14]. In addition, our work can also enable host-based *semantic-based* correlation analysis between inputs and network packets. Fine-grained input-traffic correlation is an open question (see also Section 6) that has not been addressed in existing malware-detection literature.

Organization of the Paper: We give an overview of our cryptographic provenance verification (CPV) approach and our security models in the next section. In order to illustrate our host-based provenance verification approach, in Section 3 we describe a cryptographic traffic-monitoring framework and also demonstrate its effectiveness in catching kernel-level malware traffic. In Section 4, we present a Trusted Platform Module (TPM)-based cryptographic protocol, which ensures that users' keystroke events cannot be forged by malware. Related

work is described in Section 5. In Section 6, we conclude the paper and describe plans for future work.

2 Models and Definitions

We define cryptographic provenance verification as a robust attestation mechanism that ensures the true origin of data produced by an entity such as a system device or a program. Such a system can be realized by cryptographically 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. As such, prevention against these attacks is critical. As it will soon become clear, the techniques in cryptographic provenance verification are also very different from the language-based or policy-based tainted inference analysis [24], as we emphasize on 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 and security assumptions We consider a strong malware attack model as follows. The malware may attempt to tamper with legitimate user-applications and the client's operating system including kernel-space components of our detection framework. Malware may run as a user-level application or conceal itself within the kernel as rootkits. Malware is active in making outside connections for command & control or attacks. For example, malware may attempt to log user inputs, inject traffic bypassing 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. Our framework aims to be secure against all these attack attempts. Hardware attacks will be studied by us but not included in this model.

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 detection ability for run-time compromises such as buffer overflow attacks [8]. Thus, we assume that the kernel and its modules are clean and do not contain run-time vulnerabilities. Although this assumption may appear to be strict, recent advances in minimal trusted operating system such as MINIX are making it become a reachable reality. In addition, advanced rootkits may still be active under this assumption, indicating the importance of our solutions.

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 3 and 4 for enforcing the origin of outbound packets on the Internet protocol stack and for ensuring the correct origin of keyboard inputs, respectively.

3 Provenance Verification For Host's Traffic Integrity

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 fundamental and necessary building block for constructing powerful personal firewalls or traffic-based malware detection tools. Malware bypassing commercial personal firewalls has been a common problem.

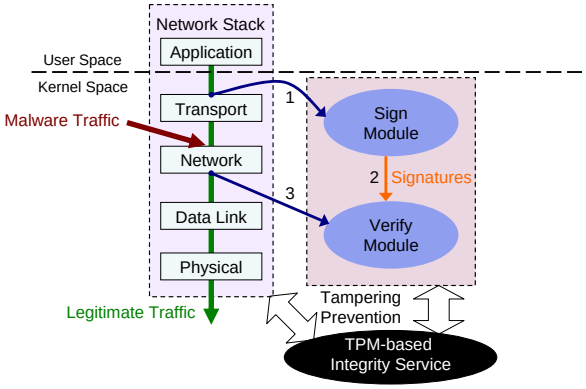

Fig. 1. Schematic drawing of components in the framework and their interactions with the host's network stack. Legitimate traffic origins from application layer whereas kernel-level malware traffic is injected into the lower layers.

We demonstrate the effectiveness of our traffic-monitoring framework in detecting stealthy outbound traffic of kernel-level malware, namely rootkits. 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. Therefore, outbound malware traffic that bypasses normal user-mode network functions to send is detected, as it is unable to

provide their provenance proofs. Thus, we effectively prevent any traffic to send without passing through the checkpoint – significantly improving the assurance of traffic-based malware detection on hosts. Such a simple-yet-powerful traffic-monitoring framework conveniently yields several advanced detection mechanisms such as input-traffic correlation analysis on application-level traffic such as [28, 29], which we briefly describe in Section 6. In particular, our solution provides an effective defense against common bypassing attacks.

Most malware constantly communicates with the outside world, with the intent of exporting sensitive data. Our detection is, recurrently, based on the observation that there are intrinsic differences between how a person and malware interacts with a computer. Legitimate outbound network traffic initiated by humans passes through the entire network stack in the host's operating system. Rootkit is a type of malware that hides its presence in operating systems making it difficult to detect. Rootkit-based malware typically bypasses higher-layer of inspections in the network stack by directly calling low-level network functions, as illustrated in Figure 1. We explore the network stack and packet properties of outgoing traffic generated by humans and malware, and develop a robust cryptographic protocol for enforcing the proper *packet provenance on the network stack*.

3.1 Architecture of Traffic Provenance Verification

We design a traffic-monitoring framework in a stand-alone architecture and demonstrate the feasibility of cryptographically enforcing that all outbound traffic flows through a transport-layer entrance on a host's Internet protocol stack. 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) travels all five layers on the stack from the top to the bottom before sending out. System services (e.g., Windows updates) are typically implemented as applications, thus their traffic also traverses the entire Internet protocol stack.

Our design of the traffic-monitoring framework extends the host's network stack and realizes *two* kernel modules, *Sign* and *Verify* modules, as illustrated in Figure 1. Both signing and verification of packets take place on the same host but at different layers of the kernel network stack – the Sign module is at the upper edge of the transport layer, and the Verify module is at the lower edge of the network layer. The two checkpoints 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 through 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 stack provenance of a packet. If a packet's signature cannot be verified, then it is labeled as suspicious, having bypassed the Sign Module, and likely generated by stealthy malware.

Directly invoking lower data-link layer or physical layer functions to send traffic is hardware-dependent and hard in practice. We find that installing the

Verify module at the network layer is sufficient. We note that our framework alone cannot detect *application-level* malware such as malicious browser extensions, as we aim to provide a general infrastructure for traffic-monitoring. Detecting application-level malware by semantic-based input-traffic correlation using our framework is currently being developed by us.

For 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. Specifically, Verify Module sends a random string encrypted with Sign Module's public key and Sign Module replies with another random string encrypted with Verify Module's public key. The XOR result of the two strings is the symmetric key which is used to sign and verify network packets.

To ensure the *integrity of the detection framework* and *signing key secrecy*, we can utilize the on-chip TPM to generate the signing keys and to attest kernel and module integrity at boot. The approach is similar to keystroke-integrity service described later in Section 4, where the attestation of kernel and module integrity requires a remote trusted server. Enlisting a remote server for integrity purpose was also previously used in [2]. There are two main goals in such a TPM-based integrity service: *i)* monitoring the integrity of client's kernel including components of our traffic-monitoring framework, and *ii)* ensuring the integrity of the Sign/Verify module's secret keys. For achieving *i)*, standard TPM operations for attesting client's kernel integrity via hash-based trusted boot, RSA-based initial authentication, and chained attestation. We also require the server to continuously monitoring the client's kernel state through periodic measuring its TPM quotes summarized in platform configuration registers (PCRs) – a number of 160-bit registers intended to enable the server to obtain unforgeable information about the client's platform state. Although more complex, under certain assumptions of the sealed storage and evaluation of attestation values, it is also possible to realize the integrity service on the same host in a stand-alone architecture. Details are omitted due to space limit. In comparison to the virtualization-based traffic detection approach by Srivastava and Giffin [26], our solution provides an effective cryptographic alternative that leverages the available trusted computing infrastructure.

3.2 Prototype Implementation and Experiment Evaluation

We implement our rootkit detection mechanism in Windows XP, and experimentally evaluate it with real-world rootkits and assess the throughputs on upstream network traffic. The Sign Module is realized as a TDI filter device at the upper edge of the transport layer in the Windows TCP/IP stack. All legitimate network packets from the Winsock API is captured and signed by the Sign Module. The Verify Module is an NDIS intermediate miniport driver at the lower edge of the network layer. It intercepts and verifies all packets just before they are sent to network interface card drivers.

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]. 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 and signature-stripping by the Verify Module. Instead, the Sign Module sends UMAC directly to Verify Module, as shown in Figure 1. UMAC values are kept in a hash table indexed by packet source address, destination address and port for fast lookup.

We first test our tool against a piece of proof-of-concept malware that can bypass the transport layer to send outgoing packets. Our experiments show that the Verify Module detects such an attack. However, the malware can disable URL filtering functionality of Trend Micro OfficeScan Client. An extended version of our detection implementation is able to identify real-world rootkits (weaker than our proof-of-concept malware), including `Fu_Rootkit`, `hxdef`, and `AFXRootkit`, all of which hide process information and opening ports.

Our experimental evaluation in Figure 2 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). Figure 2 shows the network throughputs with and without signing (left), or with partially signed packets (right). With provenance verification on each packet, the throughout decreases in general. However, as the packet size grows, the costs of signing and verification are amortized and the throughput approaches the ideal value. The observed performance degradation is minimal and acceptable in practice, since most personal computers have low upstream traffic even with peer-to-peer applications running.

Our above-described detection framework enforces the correct flow of outbound traffic through the host's network stack. This feature enables other 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 data. Thus, we feel that this contribution is beyond the specific rootkit problem studied.

4 Provenance Verification For Keystroke Integrity

In this section, we apply our cryptographic provenance verification approach in realizing an efficient framework for ensuring keystroke integrity in a client-server architecture. The work is motivated by wanting to utilize user inputs in malware detection. Yet, in a strong adversary model as described earlier in Section 2, a

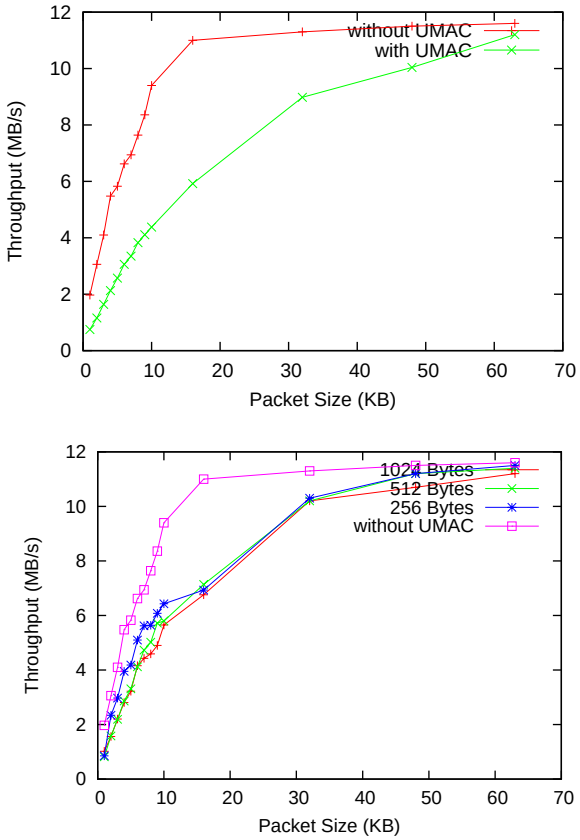

Fig. 2. Comparisons on outbound packet throughputs with or without signing in the left and with partially signed packets in the right.

necessary requirement for utilizing user inputs for security purposes is to prevent attackers from *i)* injecting (fake) input (e.g., keystroke and mouse) events into the system, and *ii)* tampering attacks on the detection framework itself. We provide an integrity service that achieves both goals. 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 can be detected. Our protocol utilizes lightweight cryptographic functions, and our key management leverages on-chip TPM [18, 1]. We use 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 is useful in addressing kernel- and root-level attacks. We, however, note 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 is thus

beyond kernel-level security. For example, any X application can inject events without any communication with the keyboard driver. Our integrity service also addresses these application-level attacks efficiently. An existing approach (as in SATEM [31]) to prevent application-level attacks, e.g., substituting libraries with compromised versions, is to have kernel libraries as part of the trusted system that gets loaded and attested by TPM. In comparison to the SATEM approach [31], our architecture is more specific to key event integrity and thus is simpler.

4.1 Architecture For Keystroke Integrity

A schematic drawing of our keystroke integrity service architecture is shown in Figure 3. Our design is to have two channels attached to the remote server: *plain keystroke events* from a client and *signed keystroke events* from a module (i.e., trust agent) that is part of the kernel attested to using the TPM. Signatures accompanying legitimate keystroke events prove their provenance information.

Our prototype realizes the *trust agent* in kernel and a *trust client* in user-space as shown in Figure 3. The trust client is a program that forwards messages between the (kernel-level) trust agent and remote server. The client-side trust agent and the remote trusted server shared a secret session key. The trust agent signs each keystroke event and the server verifies the signature. The server also monitors through TPM-based operations the client's integrity including its kernel and components. If an attacker tampers with the client, the remote trusted server notices mismatches in the information sent from the two channels, by verifying the signatures of keystroke events.

This integrity service defends against advanced malware attacks including the replay of prior-captured user keystrokes, fake key-event injections, and tampering with our client. Main operations include: *i)* trust agent and remote server key exchange; *ii)* trust agent signs keystroke events; *iii)* client relays signed events to the server; and *iv)* remote server also verifies kernel configuration. Our prototype is implemented in a client-server architecture in Linux using the Intel Integrated TPM, details of which are described in Section 4.3. This capability in 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. This keystroke-integrity service can also be applied to ensure the integrity of mouse events, e.g., mouse clicks, which we do not demonstrate here.

4.2 Key Management in Keystroke Integrity Service

We present our key management mechanism used in our keystroke integrity service. TPM operations using the on-chip master secret key is slow for time-sensitive applications such as ours. Therefore, we introduce a set of secret keys derived from it for our cryptographic operations in order to improve the efficiency. Our design involve creating three private/public RSA key pairs: a *binding key*, a *signing key* and a *storage key*. The binding key is used to securely store the

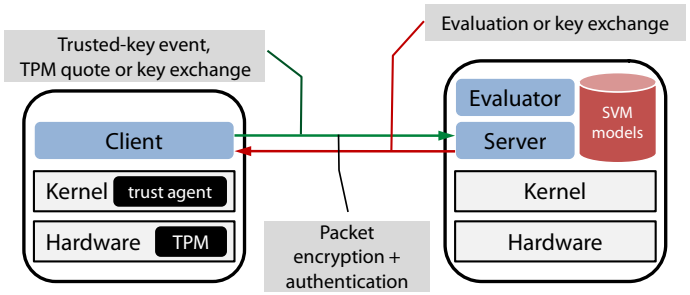

Fig. 3. The architecture of our keystroke-integrity service.

symmetric keys used for signing and encryption. The signing key is used to sign and encrypt outbound packets as well as TPM quotes. The storage key is used to store the binding and signing keys. Our key management mechanism leverages the on-chip TPM secret key and 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 or quote signing follows the following procedure.

1. The trust agent uses the TPM to generate two random strings (a_0,a_1). The trust agent generates a 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 strings (b_0, b_1) and encrypts them using the trust agent's public key.
3. Server and trust agent exchange random strings and XOR the received bits 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 [23]. Finally, the server verifies the TPM quote.

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 TPM quote procedure is repeated periodically during each connection. The secrecy of keys is guaranteed, as they are encrypted (and stored on hard disk) with on-chip TPM key when not used; additionally, when the keys are decrypted and loaded into kernel memory, because `/dev/kmem` is disabled, reading of the keys is also prevented. The latter is enforced by the server's verification of signed quotes representing machine states.

4.3 Implementation and Evaluation of Keystroke Integrity Service

Our prototype is implemented using the Intel Integrated TPM, following TPM Interface Specifications 1.2. We write code that realizes a *trust agent* in kernel and a *trust client*. The trust client is a simple-yet-essential program that parses the non-encrypted messages and forwards them accordingly between the kernel-level trust agent and remote server. We provide cryptographic functions on key events, including signing key events by the trust agent and verifying key events by the remote server. We also 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 provide key management mechanism for the integrity service that leverage TPM storage keys, as described in Section 4.2.

We implement a program in C which injects keyboard events in order to simulate forgeries. Our attack simulator has two components: the data synthesizer and typing event injection. To simulate an (intelligent) bot's attack, we write a program to create fake keyboard events and inject them into the X server core-event-stream (using the XTrap extension) as if typed on the actual keyboard. From the application's (or X client's) perspective, the fake keyboard events cannot be distinguished from actual key events (even though the keyboard is not touched). Using the integrity service we confirm that our synthetic bots that inject X-layer fake events are recognized as rogue.

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 compiled in the kernel. The module creates a device `/dev/cryptkbd`. We disable `/dev/kmem` and module loading after boot as to prevent any tampering with the agent. A user-space *trust client* opens device `/dev/cryptkbd` and concurrently opens a socket to the trusted server, waiting for communication. When the trust client opens `/dev/cryptkbd`, the trust agent attests the trust client, which also prevents any other program from opening the device.
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 TPM platform configuration registers (PCRs), a *TPM quote* (i.e., signed hash of the PCRs), and trust client signature, all signed using the TPM signing key. 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. The trust client forwards keystroke events to remote server that verifies the integrity of events. If signatures associated with events do not pass the server's verification, the trust agent is notified. The server also performs timing-based authentication analysis as required.

We evaluate the overhead incurred by the event signing and encryption in the keystroke-integrity service. We compute the average time over 1312 keystroke

events with the TPM key initiation amortized. Each key press and key release event is in a separate packet of 384 bytes. The signing of a packet using SHA-1 with a 256-bit key takes 18.0 microseconds while encrypting a packet using standard AES-CBC with a 256-bit key takes 67.6 microseconds. To estimate the bandwidth overhead, we assume that a fast typist enters 212 words per minute [3] and in English average word has 4.5 characters [25]. Each character has a press event and a release event, respectively. Therefore, we obtain 12.2 KBps maximum bandwidth overhead. Overall, we find that the cryptographic operations introduced by this integrity service have low computational and communicational overhead.

This integrity service is robust against a range of advanced attacks including gaining root privilege on the computer, collecting the owner's keystroke information, fake key event injections, tampering local client, and rogue kernel/libraries. 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 bots from injecting fake events into other applications. One needs to expand the TPM support for the applications to be protected, by writing a trusted wrapper for the application to interface with the trust client and verify the events.

5 Related Work

Our paper focuses on a host-based approach for ensuring kernel-level data integrity and demonstrates its application for malware detection. In comparison, network trace analysis typically characterizes malware communication behaviors for detection [9–13, 16, 22, 27]. 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 IRC (Internet Relay Chat) protocol and sends a large number of emails afterwards [12]. 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 [6] and Gummadi *et al.* [14]. They investigated and enforced the temporal correlation between user inputs and observed traffic. The BINDER work [6] 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 for that problem. In comparison to NAB [14] which is designed specifically for browser input verification, our

work provides a more general kernel-level solution for keystroke integrity that is *application-oblivious*.

Existing rootkit detection work largely focuses on operating system level detection, including identifying suspicious system call execution patterns [4], discovering vulnerable kernel hooks [28], exploring kernel invariants (e.g., Gibraltar [2]), or using virtual machine to enforce correct system behaviors [7, 26]. 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 [7]. Wang *et al.* systematically identified potential kernel hook points in Linux kernel [28]. 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., [7, 26]). In this paper, we leverage existing trusted computing infrastructure (the TPM is available on most commodity computers) for enforcing various system integrity, namely outbound-traffic integrity and keystroke integrity. 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. The limitation of TPM under run-time compromises as mentioned in Section 2 is still an active research area. For example, Flicker is a recently-proposed trusted computing base for allowing 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 user input events, thus preventing fake keyboard activities.

The work by Srivastava and Giffin [26] on application-aware blocking of malware traffic may bear superficial similarity to our host-based rootkit detection solution. They used virtual machine monitor (VMM) to monitor application information of guest OS without using any cryptographic scheme. It is important to note that our cryptographic provenance verification is a more general kernel-level mechanism for ensuring data provenance that can be applied beyond the specific network traffic monitoring and keystroke authentication problems studied.

6 Conclusions and Future Work

We described a general cryptographic provenance verification (CPV) technique for ensuring the correct provenance (i.e., origin) of dynamic data generated in the kernel. We demonstrated CPV's application in host-based malware detection, in particular how to distinguish malicious/unauthorized data flow from legitimate one on a computer that may be compromised.

Specifically, we made the following technical contributions in this paper. **i)** 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 a checkpoint that cannot be bypassed by malware traffic, in particular from kernel-level malware. **ii)** 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.

For future work, we plan to extend our cryptographic provenance verification approach to develop advanced input-traffic correlation and tracking analysis in both stand-alone and client-server architectures. In almost all client-server or pull architectures (e.g., Web applications), users initiate the requests, which typically involve keyboard or mouse events. Few exceptions such as Web server refresh operations can be labeled using whitelists. We will investigate how to characterize and enforce normal traffic and input correlations in applications such as Web browsing, legitimate downloading activities, and P2P file sharing activities. We will study robust defenses against sophisticated malware exploits in these new user-behavior driven security systems.

References

1. W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *In Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71. IEEE Computer Society, 1997.
2. 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.
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 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. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. *Security and Privacy, IEEE Symposium on*, 0:32–46, 2005.
6. 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.
7. 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.
8. 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.
9. J. Goebel and T. Holz. Rishi: Identify bot contaminated hosts by IRC nickname evaluation. In *Proceedings of First USENIX Workshop on Hot Topics in Understanding Botnets*, April 2007.
10. J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon. Peer-to-peer botnets: Overview and case study. In *Proceedings of First USENIX Workshop on Hot Topics in Understanding Botnets*, April 2007.
11. 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.
12. G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. Bothunter: Detecting malware infection through IDS-driven dialog correlation. In *Proceedings of the 16th USENIX Security Symposium (Security)*, 2007.
13. G. Gu, J. Zhang, and W. Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 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. A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 7–7, Berkeley, CA, USA, 2007. USENIX Association.
17. T. Krovetz. UMAC: Fast and Provably Secure Message Authentication. `http://fastcrypto.org/umac/`.
18. B. Lampson, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, 1992.
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. 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.
23. B. Schneier and N. Ferguson. Practical cryptography, 2003.
24. R. Sekar. An efficient black-box technique for defeating web application attacks. In *ISOC Network and Distributed Systems Symposium (NDSS)*, February 2009.
25. C. Shannon. Prediction and entropy of printed English. *Bell System Technical Journal*, 30(1):50–64, 1951.
26. 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.

27. P. Wang, S. Sparks, and C. C. Zou. An advanced hybrid peer-to-peer botnet. In *Proceedings of First USENIX Workshop on Hot Topics in Understanding Botnets*, April 2007.
28. 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.
29. H. Xiong, C.-C. Chang, and D. Yao. Exploring the human-behavior driven detection approach in identifying outbound malware traffic, 2009. USENIX Security Symposium. Extended Abstract.
30. 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.
31. 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.