A Cryptographic Provenance Verification Approach For Host-Based Malware Detection

Deian Stefan
Department of Electrical Engineering
The Cooper Union
New York, NY 10003
stefan@cooper.edu

Chehai Wu
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
wuc@cs.rutgers.edu

Danfeng (Daphne) Yao
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
danfeng@cs.rutgers.edu

Gang Xu
AT&T
200 Laurel Ave
Middletown, NJ 07748
gangxu@att.com

ABSTRACT

We present a malware detection approach by focusing on the characteristic behaviors of human users. We explore the human-malware differences and utilize them to aid the detection of infected hosts. There are two main research challenges in this study: one is how to select characteristic behavior features, and the other is how to prevent malware forgeries. We address both questions in this paper.

A cryptographic provenance verification technique is described. Its two applications are demonstrated in keystroke-based bot identification and rootkit traffic detection. Specifically, we first present our design and implementation of a remote authentication framework called TUBA for monitoring a user's typing patterns and verifying their integrity. We evaluate the robustness of TUBA through comprehensive experimental evaluation including two series of simulated bots. We then demonstrate our provenance verification approach by realizing a lightweight framework for blocking outbound rootkit-based malware traffic.

Keywords: authentication, malware detection, cryptography, provenance, network

1. Introduction

Studies have estimated that millions of computers worldwide are infected by malware and have become bots that are controlled by cyber criminals [19]. The infected computers are coordinated and used by the attackers to launch diverse malicious and illegal network activities, including perpetrating identity theft, sending spam (estimated 100 billion spam messages every day [37]), launching denial of service (DoS) attacks, committing click fraud, etc. The victim's computing experience also suffers as the computing cycles wasted on bot-induced activities typically degrade the performance of the machine.

Most existing botnet detection solutions are quite effective in detecting existing botnets [12, 14, 19]. These approaches typically focus on analyzing the network traffic of potentially infected machines to identify suspicious network communication patterns. In particular, the traces of botnets' command and control (C&C) messages, i.e., how bots communicate with their botmasters (or botherders), are captured and their signatures and access patterns are 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. An example of a botnet detection solution that analyzes network traffic for suspicious bot-like activities (e.g., egg downloading, scanning local network, etc.) is BotHunter [13]. Network trace analysis is a critical aspect of identifying malicious bots. These solutions usually involve complex and sophisticated pattern analysis techniques, and have been demonstrated to produce low false positive and false negative rates. Furthermore, they can be deployed by local ISPs to monitor and screen a large number of hosts as part of a large-scale network intrusion-detection system.

These methods alone are suboptimal as botnets are entities that are constantly evolving to avoid detection, and thus their behaviors change accordingly. For example, although IRC is still the dominating botnet command and control protocol, recent studies have found that many botmasters are responding to detection systems by switching away from IRC to HTTP [17]. Unlike IRC C&C traffic, HTTP traffic is usually allowed through firewalls and can be easily camouflaged to be used for covert channels. Sole reliance on following and leveraging bots' behaviors for detection may require continuous modifications in order to keep up with the newest development of botnets.

In this paper, we adopt a different malware detection approach by focusing on the *characteristic behaviors of humans*. There are intrinsic differences between how a person and a bot uses and reacts to computer applications. For example, studies of online chatting behaviors of chat bots and humans have shown that bots and humans behave quite differently [11]. These human-bot differences are furthermore

utilized to distinguish humans from bots and aid the detection of infected hosts. Our ultimate goal is to design robust bot detection mechanisms that are extremely difficult for future generations of botnets to circumvent.

There are, however, two main research challenges in this study: *how to select characteristic behavior features*, and *how to prevent malware forgery*. Certain features extracted from humans' computer interactions, such as click counts or email sizes, may not be characteristic enough and cannot be used to uniquely represent an individual. In addition, advanced malware may attempt to mimic human activities to spoof the legitimate user in the detection. Thus, the host used to collect behavior features should distinguish true events from fake events that are injected by malware in hope to circumvent the authentication system (i.e., fake events need to be identified). We address both challenges by developing a general provenance verification method. We present the use of robust keystroke dynamics as a trait for intrusion detection. We also deploy TPM-based integrity measures to prevent malware forgery.

It is worth mentioning that keystroke and mouse movement analysis has been studied for authentication and user identification [6, 18, 21, 27, 35, 44, 21]. In comparison to previous studies, we evaluate and improve the robustness of keystroke dynamic authentication under malware environments, which have unique challenges. Furthermore, we develop general security frameworks that are useful beyond the specific biometrics studied.

Our contributions are summarized as follows.

1. We design and implement a host-based bot detection framework called TUBA (Telling hUmans and Bots Apart). TUBA explores the uniqueness in human keystroke dynamics, and uses it as a strong indicator for potential bot infection. In order to further improve the robustness of TUBA, we design and implement a TUBA integrity service based on on the hardware Trusted Platform Module (TPM). Our integrity verification mechanism uses a lightweight cryptographic protocol to prevent bots from injecting fake keystroke events into computer applications. The integrity service also prevents tampering attacks on the TUBA. Our experimental evaluation on the overhead, accuracy, robustness, and usability of TUBA demonstrates the efficiency and feasibility of TUBA in host-based malware detection.
2. To further illustrate the generality of our host-based provenance verification approach, we describe the design and implementation of a lightweight rootkit detection method. We detect stealthy outbound traffic of rootkits by enforcing a cryptographic provenance verification scheme for outgoing network packets. Thus, rootkits that bypass normal user-mode network functions to send traffic cannot provide their provenance proofs and are 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%.

Organization of the Paper: We give an overview for our TUBA framework in the next section. We describe the technical details of the main components in our TUBA framework in Section 3. In Section 4, we present our simulation algorithms for two intelligent bots aiming to break the classification algorithm in TUBA. Our experimental evaluation is given in Section 5. In Section 6, we describe a lightweight rootkit detection mechanism as another demonstration of our host-based provenance verification approach. Related work is described in Section 7. In Section 8, we conclude the paper and describe plans for future work.

2. TUBA Overview

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 [33], as we emphasize on the enforcement of normal system properties with lightweight cryptographic primitives and trusted computing infrastructure.

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. We illustrate two such applications in the rest of the paper.

TUBA is a remote biometric authentication system based on trusted keystroke dynamic information. The goal of TUBA is to ensure that the computer is used by its true owner, and to identify bots' activities. We refer to an individual who has legitimate access to the computer as the *owner*. Without loss of generality, we assume that a computer has one owner, as our solutions can be easily generalized to a multi-owner setting.

Security Goal: We aim to prevent unauthorized use of a personal computer by a malicious bot (as part of a botnet) 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 (whether bot or human)?

Weak Adversary Model: An adversary may infect a user's computer through social engineering and/or malware. The infected computer may belong to a large botnet controlled by the adversary in order to carry out malicious activities. The adversary considered is able to monitor, intercept, and modify network traffic between the owner of the computer and the rest of the Internet. We allow a powerful adversary to access the keystroke data of the general public, except that of the target computer's owner. In other words, the adversary is capable of collecting, analyzing, and synthesizing keystroke data from anyone except the owner. The adversary's goal is to forge and mimic the owner's keystroke patterns that pass the authentication tests. In this weak adversary model, we assume that any *malicious keyloggers* installed on the user's computer can be detected and re-

moved. We assume that our detection program, TUBA, is not corrupted or disabled on the user's computer. The latter two assumptions suggest that the adversary's unauthorized access privileges are limited to those of the victim which we assume to be non-superuser.

Strong Adversary Model: In the above description we assume that the adversary can infect the user's computer, monitor and tamper with network traffic, collect and inject keystroke information of the general public, except the owner's. However, keylogging is relatively easy to perform stealthily, and keyloggers are difficult to remove. In a strong adversary model, we allow a bot to learn the owner's keystroke patterns, attempt to inject fake keystroke events, tamper with the TUBA client, and gain superuser privileges on the user's computer, in addition to the power described in the weak adversary model. TUBA with TPM-based integrity service defends against this type of strong adversaries (described in Section 3.4).

Architecture: Our TUBA framework can be realized with a *stand-alone program* on the client's local machine. The program is responsible for collecting training keystroke data, building learning models, analyzing, and classifying TUBA challenges. This type of stand-alone architecture is easy to deploy and implement. It is, however, required that the user ensure that the program is running and that proper measures are taken if TUBA issues warnings or alerts. TUBA can also be implemented as a *client-server architecture*. The server can be run by the local ISP or a trusted company providing security services for the user. In this architecture, the server is responsible for data collection and analysis in a remote fashion, e.g., using SSH (Secure Shell) the client would remotely login to the server with X11-forwarding enabled so that the keystroke events can be monitored by the server. The connection and storage of the remote server is assumed to be secure. Our prototype implements a client-server protocol.

2.1 A Use Scenario in Client-Server Architecture

To provide a context of our authentication framework, we describe a usage scenario of TUBA in a client-server architecture as follows.

1. **Training Phase:** The remote authentication server collects keystroke data from a legitimate user. We assume that the user's computer is not infected during the training phase, but may be infected and recruited into a botnet after the training phase has ended. The training phase is as follows.
   (a) The user and the remote server authenticate each other and set up a secure connection. The user then types M strings $s_i,\ i = 1, \ldots, M$, as specified by the server, n times each.
   (b) The authentication server records the keystroke data from the user, which is possible using the X Window System. The user runs X server[1] with an extension (XTrap), which intercepts the user's keystroke events and sends the information to the application on the remote authentication server.
   (c) Once a sufficient number of samples have been collected, the authentication server processes the user's keystroke data by training a support vector machine, the details of which are presented in Section 3.
2. **TUBA challenge:** When a suspicious network event is observed, TUBA prompts the user with a window requesting him/her to type in a server-chosen string, s_i. Based on this user's keystroke timing data and the classification model built during the training phrase, TUBA decides whether the user is the legitimate owner or not.
3. **Recognize user's own traffic:** If the user passes the authentication test and is verified as the PC's owner, then TUBA informs the owner that a suspicious event has been observed and asks whether the owner is aware of the network connection. TUBA assists the user in identifying stealthy intruders on their computer by utilizing his/her own behavioral features and personal knowledge. In Section 5.3, we further describe our experiments on evaluating how well a user is able to identify her own traffic.

The suspicious events mentioned above may be triggered by existing bot detection solutions, such as BotHunter [13], BINDER [9], or according to other (simple) pre-defined policies.

We note that TUBA is more efficient than the recently-proposed Not-A-Bot (NAB) system [15], as we do not need to certify every keystroke event, only when a user responds to a TUBA challenge. Furthermore, because of our timing-based classification, TUBA also provides a fine-grained authentication ability.

3. Classification, Remote Event Collection, and Integrity Service in TUBA

In this section, we describe the technical details of our TUBA framework, including feature extraction, classification, remote event collection, and the TUBA integrity service. We implement a prototype that demonstrates the usability and feasibility of remote authentication based on keystroke dynamics. We further strengthen TUBA with the integrity service that is capable of defending against the strong adversary model defined in Section 2.

3.1 Feature Extraction

Given a sequence of key press and key release events, features represent various temporal aspects of the user's typing patterns. Features may include the total typing time of the word and inter-key timings such as the interval between two adjacent press or release events. Even for a short string such as the URL `www.amazon.com`, the dimensionality of all possible features is quite high. The TUBA classification algorithm uses principle component analysis (PCA) to reduce the dimensions of the feature vectors as a preprocessing step. PCA is an (existing) data mining and statistical technique which is commonly used to condense high-dimensional data to lower dimensions in order to simplify analysis. The premise of PCA is to reduce the dimensions of and transform the original multi-dimensional datasets so that high

[1]Note that in X Window System the user's machine is called the X server and the remote program is called the X client, which may seem counter-intuitive at the first sight.

variations within the data are retained (i.e., the principal components are retained).

A high-dimensional feature vector used in the classification, however, makes it difficult for adversaries to successfully simulate keyboard events that pass our classification test. For example, the word "botnet" is typed by two individuals and as shown in Figure 1, three keystroke features are used to distinguish the two users, including the press-to-press (PP) time of two adjacent characters, key durations of individual characters (D), and the total typing time of a word. The two users' samples are well-separated using a 3-dimensional feature vector in this example.

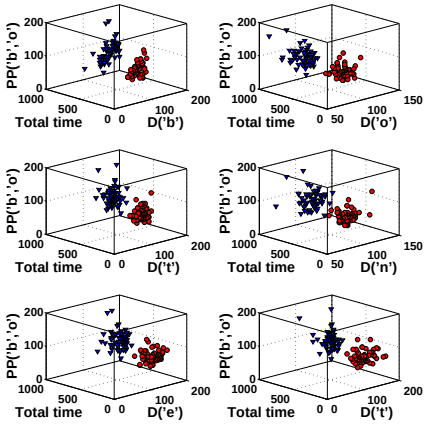

Figure 1: Distribution of three keystroke features of two users. The times and durations are given in milliseconds. One user's data is shown with the red circles, and the other user's with blue triangles.

Humans are imperfect typists and may create negative timing features in a sequence of keystroke events. For example, when typing the string "abc", a user may create negative press-to-release (PR) time by pressing 'c' before having released 'b'. More formally, if we denote the state at $i-1$ as x_{i-1} = 'b', and that at i as x_i = 'c', given that 'c' is pressed before 'b' is released then $\mathrm{PR}(x_{i-1}, x_i) = x_{i,p} - x_{i-1,r} < 0$. From our experimental data, we find that a large number of users have negative press-to-release timings in their datasets. Although an adversary can synthesize arbitrary keystroke events, we find that it considerably more difficult to create an intelligent bot which can inject keystroke events that result in negative inter-key timings (See also Section 4).

Figure 2 illustrates the practical differences in the capabilities between human and bots. Assuming that keystroke events can be modeled accurately by a first-order Markov chain, a human's key event path would be a combination of the dashed and solid lines shown in the figure. It is, however, difficult for a bot to simulate certain events, as is the case of negative timing features (paths including dashed lines in Figure 2). When considering higher-order Markov chains, it is even more challenging for the attackers to successfully mimic typing patterns with negative timing; a person may, for example, press 'c' before both 'a' and 'b' are released. Using high-dimensional data leads to higher authentication accuracy and stronger security guarantees. However, if the complexity of the model is increased (e.g., to a second- or third-order Markov chain) it is important to collect additional training instances as to avoid overfitting the data.

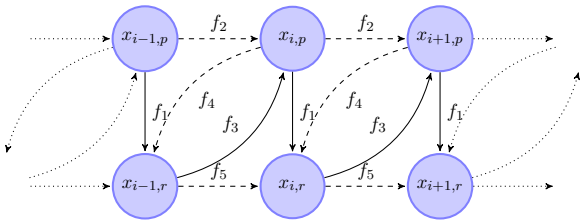

Figure 2: Comparisons between the typing abilities of a person and a bot modeled by using a first-order Markov chain. $x_{i,p}$ and $x_{i,r}$ denote the i-th letter pressed and released, respectively. Linear combinations of the f_k elements represent timing features.

3.2 Classification

Once keystroke features are collected and processed, we train and classify the data using support vector machines (SVMs). The use of SVMs is appropriate as the technique can be used to classify both linearly-separable (i.e., classes which are separable into two or more groups using hyperplanes) and non-linearly separable data [16, 20, 28]. To classify a set of data points in a linear model, support vector machines select a small number of critical boundary points from each class, which are called the *support vectors* of the class. Then, a linear function is built based on the support vectors in order to separate the classes as much as possible; a *maximum margin* hyperplane (i.e., a high-dimensional generalization of a plane) is used to separate the different classes. An SVM model can classify non-linear data by transforming the feature vectors into a high-dimensional feature space using a kernel function (e.g., polynomial, sigmoid or radial basis function (RBF)) and then performing the maximum-margin separation. As a result, the separating function is able to produce more complex boundaries and therefore yield better classification performance. In our authentication system, we use the WEKA [42] SVM implementation with a Gaussian RBF kernel. We refer readers to data mining and machine learning literature such as the book by Witten and Frank [42] or Bishop [4] for detailed descriptions of SVM techniques.

3.3 Remote Collection of Keystroke Events

Various key-logging methods for the GNU/Linux operating system exist; common implementations include user-space programs which monitor privileged I/O ports [8], kernel modules that hijack the `sys_read` and `sys_write` functions [26], and kernel modules that hijack the keyboard driver's interrupt handler [30]. However, most of the currently-available keyloggers were *not* designed with the intention to extract timing information from a user's typing pattern, and require superuser privileges to be installed or used. Addressing these issues and the need for a platform-independent utility, we implemented a keylogger for the *X Windows System* using the XTrap extension [1].

The X Windows System (X or X11 for short) is a powerful graphical user interface composed of the *X server* and *X clients*. The X server runs on the machine where the keyboard, mouse and screen are attached, while X clients are common applications (e.g. *Firefox*, *KPDF* or *XTerm*) that run on either the local machine or a remote machine, due to the inherent network capabilities of X11 [29, 31].

The X server can be extended with modules, such as the XTrap server extension used in the TUBA event collection. One of the capabilities of the XTrap extension is to intercepts the core input (keyboard, mouse) events and forward them to XTrap client applications. As such, our keylogger (client application) contains a callback function which is executed whenever a `KeyPress` or `KeyRelease` event occurs to record the event information. Some supplementary data, such as the current location of the mouse pointer and the name of the current window in focus, are obtained and formatted to be easily parsed by the feature extractor.

The output is then parsed by the feature extractor, which contains a small buffer of the last C `KeyPress` and `KeyRelease` events. Given a database of words $(s_i,\ i = 1, \ldots, M)$ to monitor[2] and feature descriptions (i.e., keystroke durations, total time to type a word, press-to-press times, etc.) of how the strings were typed, when the buffer contents of the keyboard input matches a database word, the features are extracted and again formatted to be easily parsed by the classifier.

3.4 TUBA Integrity Service

To further improve the robustness of TUBA, in particular to prevent attackers from tampering with TUBA and injecting fake keystroke events into TUBA, we provide an integrity service in TUBA. Our approach is based on lightweight cryptographic functions and our key management leverages on-chip TPM [24, 2].

The TPM is very 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 TUBA integrity service also addresses these application-level attacks efficiently. An existing approach (as in SATEM [43]) 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 [43], our architecture is more specific to key event integrity and thus is simpler. We only attest the kernel and TUBA client and disable module (re-)loading after boot. *Our main idea is to have two communication channels to the remote TUBA server, one from the application and the other from a trust device that is part of the kernel that is attested using the TPM.* If an attacker tampers with TUBA, the remote server can notice mismatches in the information sent from the two channels. We present the details of our architecture next.

Stronger Security Guarantees: With the TUBA integrity service, TUBA is robust against a whole host of advanced attacks including gaining root privilege on the computer, collecting the owner's keystroke information, fake key event injections, tampering TUBA client, and rogue kernel/libraries. The strong adversary defined in Section 2 is prevented.

3.4.1 Implementation of TUBA Integrity Service

We implement the TUBA integrity service by expanding the basic TUBA prototype in the following aspects. Our prototype is implemented using the Intel Integrated TPM, following TPM Interface Specifications 1.2 [38]. We write code that realizes a *trust agent* in kernel and a *trust client* in TUBA. 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 TUBA server. We also provide the encryption and decryption functions on the packets from the TUBA 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 3.4.2.

Using the integrity service we confirm that our synthetic GuassianBot and NoiseBot (described in Section 4) that inject X-layer fake events are recognized as rogue.

We describe the detailed procedure of starting and running TUBA integrity service between the client and the remote server as follows. A schematic drawing of the TUBA integrity service architecture is shown in Figure 3.

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 TUBA client, it requests the initial attestation. The trust client on the TUBA 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 (see Section 3.4.2). The trust client forwards the information to the TUBA 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 (see Section 3.4.2 for details). When the TUBA server requests a TUBA challenge, i.e., requiring the user to type in a specific string, the trust agent forwards the encrypted and signed keystroke events to the trust client. The trust client then simply forwards the 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 TUBA server also performs timing-based authentication analysis as required.

Our aforementioned protocol describes 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. Due to space

[2] C is adjusted to match the largest word in the database.

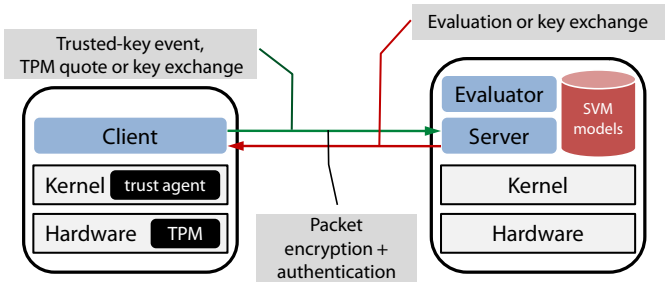

Figure 3: Architecture of TUBA integrity service. Main operations include: trust agent and remote server key exchange; trust agent signs keystroke events; client relays signed events to the server; remote server also verifies kernel configuration.

limitations, we do not provide detailed descriptions on this topic.

3.4.2 Key Management in TUBA Integrity Service

In this section, we present our key management mechanism used in the TUBA integrity service. The TPM is used to create 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 symmetric keys used for signing and encryption, the signing key is used to sign the TPM quote, and the storage key is used to store the binding and signing keys. 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 [32]. 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.

Summary of TUBA integrity service: The solution described in this section for ensuring the authentic origin of keystroke events embodies our *host-based provenance verification* approach, that is, the source that generates a user event is verified using lightweight cryptographic primitives.

4. Bot Simulation and Events Injection

With our TUBA integrity service enabled, fake key events can be completely detected and removed. However, it important to evaluate the robustness of keystroke authentication under automatic bot attacks when TPM-based integrity service is not available. We find that even if we allow for certain types of key event injection by bots, our classification method is able to identify intruders.

To that end, we play the devil's advocates and create two series of bots, the algorithms of which are described next. We assume that the goal of an adversary in our model is to create keystroke events that pass our classification tests. That is, the attacker attempts to create fake keystroke events expecting them to be falsely classified as the owner's. Under our weak adversary model (defined in Section 2), we assume that bots possess keystroke data of some users except the owner's [3].

We implement a program in C which injects keyboard events with specific timing information 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). To test the performance of a bot injecting fake events we implemented two bots which simulate human typing patterns according to the *first-order Markov model* shown in Figure 2. That is, bots consider only keystroke durations and positive inter-key timings (paths shown by the solid lines in Figure 2).

In our simulations, the keystroke duration of the ith character in a word is modeled as a random variable $X_i \geq 0$, where X_i is either

1. Gaussian with mean μ_i and variance σ_i^2: $X_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$, or
2. constant with additive uniform noise (mean 0): $X_i \sim \mu_i + \mathcal{U}(-\eta_i, \eta_i)$,

depending on the type of bot desired, GaussianBot or Noise-

[3]I.e., replaying the owner's keystroke sequence is prohibited under the weak adversary model.

Bot. The parameter μ_i is calculated as the mean key duration of the i-th character from selected instances of the user study. For example, to calculate μ_1 for the first character ('1') in the string "1calend4r" we take the `1calend4r` instances from the user study and calculate the sample mean and variance of the keystroke durations for the character '1'. Similarly, the press-release inter-key timing feature between the i-th and $(i-1)$-th character was modeled as a random variable X'_i, whose parameters are also calculated from the user study instances. Algorithm 1 below and Algorithm 2 in Appendix show the pseudocode for the bots, which inject n instances of the given string. The classification performance of these bots against users are further explained in Section 5.

It is important to note that a more complex bot would additionally consider negative inter-key timing and therefore a high-order Markov Model may be implemented. This advanced bot would require considerably greater effort from the bot designer, as the order of events would have to be calculated a priori. For example, if the bot were to correctly simulate the word "botnet" typed by a person, the probability of injecting a `KeyPress` event for the character 'o' before injecting a `KeyRelease` event of 'b' would have to be considered and therefore Algorithms 1 and 2 would need to be modified dramatically.

Algorithm 1: *GaussianBot* simulation of a human

input: string=$\{x_1, x_2, \ldots, x_N\}$,
durations=$\{(\mu_1, \sigma_1), (\mu_2, \sigma_2), \ldots, (\mu_N, \sigma_N)\}$,
inter-key
timing=$\{(\mu'_2, \sigma'_2), (\mu'_3, \sigma'_3), \ldots, (\mu'_N, \sigma'_N)\}$,
n=number of words to generate

for $n \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** N **do**
    SimulateXEvent(`KeyPress`, x_i);
    $X_i \leftarrow \mathcal{N}(\mu_i, \sigma_i^2)$; `/* key duration */`
    if $X_i < 0$ **then** $X_i \leftarrow 0$; `/* adjust for large variance */`
    Sleep(X_i);
    SimulateXEvent(`KeyRelease`, x_i);
    $X'_i \leftarrow \mathcal{N}(\mu'_i, \sigma'_i)$; `/* inter-key timing */`
    if $X'_i < 0$ **then** $X'_i \leftarrow 0$;
    Sleep(X'_i);

5. TUBA Experimental Evaluation

We carry out three types of TUBA-related experimental evaluation on the overhead of TUBA integrity service, the accuracy of classification, and user's ability of recognizing his/her own traffic, respectively.

5.1 Performance Evaluation of TUBA Integrity Service

We evaluate the overhead incurred by the event signing and encryption in TUBA 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 [5] and in English average word has 4.5 characters [34]. Each character has a press event and a release event, respectively. Therefore, we obtain 12.2 KBps maximum bandwidth overhead as follows.

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

Overall, we find that the cryptographic operations introduced by TUBA integrity service have low computational and communicational overhead.

String	GaussianBot		NoiseBot	
	TP	FP	TP	FP
`www.amazon.com`	96.29%	2.00%	100.0%	0.00%
`1calend4r`	93.74%	3.43%	97.71%	1.43%
`name2@gmail.com`	96.57%	1.71%	99.71%	0.29%

Table 2: Human vs. bots SVM classification results.

5.2 Evaluation of Classification Accuracy

We collect keystroke timing data from 20 user subjects, 10 females and 10 males on $M = 5$ different strings. We implement a program with a graphic user interface (GUI) as a wrapper to the keylogger that records the keystroke dynamics of the participants. The user is asked to type in the following strings, $n = 35$ times each: `google.com`, `www.amazon.com`, `1calend4r`, `name1@gmail.com`, `name2@gmail.com` [4]. The gender and age of each participant are recorded, as well as their familiarity ('high', 'medium', or 'low') with each string. This data is later used for analyzing the correlation between demographic data and keystroke dynamics. We perform three sets of experiments to test the feasibility and the performance of TUBA in classifying keystroke timing features. We illustrate the setup of the experiments in Table 1.

The goal of Experiment 1 is to confirm our ability to distinguish different individuals' keystroke patterns with good prediction results, as has been shown in the existing literature. We are able to achieve high accuracy in classifying individual humans; the result of Experiment 1 are shown in the Appendix.

Existing literature on keystroke authentication does not, however, provide any analysis of attacks that are based on statistical and synthetic keystroke timing; to our knowledge, there are currently no bots which are able to perform the attacks that we consider. Therefore, we design two sets of experiments to simulate some sophisticated bot attacks.

Experiments 2 & 3 (Human vs. Bots) We evaluate the robustness of keystroke analysis against artificially and statistically created sequences of events. As auxiliary information for the attacker, we give the adversary access to the keystroke data of all 19 users excluding the owner's data. Results from Experiment 2 and 3 are presented below.

The SVM classification procedure for the bot experiments is similar to that of Experiment 1 (see Appendix),

[4]Email addresses are anonymized for blind review.

#	Experiment series	Purpose	Tests on Gender
1	Human vs. Human	To distinguish between two users	Yes
2	Human vs. GaussianBot	To distinguish between a user and a GaussianBot (Algorithm 1)	No
3	Human vs. NoiseBot	To distinguish between a user and and a NoiseBot (Algorithm 2)	No

Table 1: The setup of three series of experiments. We evaluate the following strings in all experiments: `www.amazon.com`, `1calend4r`, `name2@gmail.com`. For human vs. human experiments, we also perform separate analysis on different gender groups and also evaluate additional strings: `google.com` and `name1@gmail.com`.

however only 10 user cases and $M = 3$ strings are used, with extended focus on tweaking the model parameters. The chosen strings (s_j, $j = 1, \ldots M$) included a URL (`www.amazon.com`), an email address (`name2@gmail.com`) and a password (`1calend4r`). Similar to the results of Experiment 1, gender classes only affect the results very slightly, and therefore only the class containing both genders was considered for Experiments 2 and 3. The detailed setup for Experiment 2, for word s_j of user u_j was performed as follows:

- Label each of the user's 35 instances as `owner`,
- For each character x_i, $i = 1, \ldots N$ in string s_j, calculate the parameters μ_i and σ_i, and similarly the average and standard deviation of the press-to-release times (μ_i' and σ_i') using the remaining users' ($u_k \neq u_j$) instances,
- Using the parameters as arguments for GaussianBot, Algorithm 1, generate $n = 35$ bot instances and label them `unknown`
- Perform a 10-fold cross-validation for SVM classification using the `owner` and `unknown` data sets,
- Calculate the average true positive (TP) and false positive (FP) rates.

The procedure for Experiment 3 is the same, using instead Algorithm 2, NoiseBot, as further explained in Appendix. Table 2 shows the results of Experiments 2 and 3.

In summary, the successes of the GaussianBot and NoiseBot in breaking the model are negligible, as indicated by the extremely low (average 1.5%) FP rates. Furthermore, these experiments support the results of Experiment 1 and confirm the robustness of keystroke authentication to statistical attacks that are considered.

5.3 User's Traffic Recognition Ability

In Step 3 of the TUBA use scenario described in Section 2.1, after the keystroke authentication, a user is asked whether she initiated the suspicious connection. The underlying assumption is that a user knows the websites she is currently visiting and thus can recognize malware-related traffic to/from unfamiliar servers. To experimentally evaluate whether a user is able to identify and distinguish her own traffic from bot traffic, we deploy a small-scale user study. Each of the seven participants is asked to freely surf online for 10 minutes, during which we randomly access a list of arbitrary (bot) servers. For each bot URL and user-visited URL, we prompt a window asking whether or not the user has just visited it.

From users' responses, we compute false positive and false negative rates of their performance. Here, a false negative result indicates that the participant has misclassified bot URLs for their own traffic. A false positive result, conversely, indicates that the user misclassified legitimate user-initiated traffic as bot HTTP requests. The user study code is written in Python using libpcap to sniff HTTP traffic in conjunction with the Firefox tlogger extension [39].

We note that the analysis is further complicated with the abundance of third-party content, such as advertisements and multi-media content that are hosted by content delivery providers instead of the main web server visited by the user. Thus, third-party content is retrieved from URLs that may seem arbitrary to the user, e.g., bearing no similarity to the main website URL, impacting their classification decision.

Our experiments show the following results. (i) The false negative rate is extremely low ($< 1\%$) – injected bot URLs are easily detected by users. (ii) The false positive rate is high (40%) as the participants tend to classify unknown URLs as malicious. (iii) On average, 78% of third-party URLs are classified as bot traffic, which ultimately contributed to the high false positive rate. In addition, among the URLs that participants labeled as bot traffic, more than 90% of them were due to third-party content. *This study shows that it is feasible for TUBA to leverage a user's personal knowledge on their web activities for bot detection, provided that traffic due to third-party contents can be traced and tracked.* Preventing exploits based on third-party content is beyond the scope of this paper and subject to our future work.

6. Traffic Provenance Verification For Rootkit Detection

TUBA explores the intrinsic differences between typing patterns between humans and malware and TUBA integrity service enforces the provenance (i.e., origin) of user inputs by utilizing functions provided by the TPM. This design of TUBA embodies a simple-yet-general host-based provenance verification approach that we elaborate in Section 3.4. To further illustrate the generality of such a host-based malware detection approach, we describe the design and implementation of a lightweight rootkit detection method. *We detect stealthy outbound traffic of rootkits by enforcing a cryptographic provenance verification scheme on outgoing network packets.* Rootkits that bypass normal user-mode network functions to send traffic are detected, as they are unable to provide their provenance proofs. We describe our experimental evaluation with real-world rootkits and throughput validation on upstream network traffic.

Rootkit is a mechanism that hides malware from detection; malware equipped with rootkits is extremely difficult to detect. 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. In comparison, rootkit-based malware typically bypasses higher layer inspections in the network stack by directly calling lower-level network functions, as illustrated in Figure 4 [5]. 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*.

Architecture of Traffic Provenance Verification: We assume a powerful type of malware that sends outbound traffic and is capable of hiding its presence in user space applications. We provide an add-on to the host's network stack. It consists of a *Sign Module* and a *Verify Module*, as illustrated in Figure 4. The Sign Module is at the upper edge of the transport layer while the Verify Module is at the lower edge of the network layer. Thus, all legitimate network packets initially pass through the Sign Module and then the Verify Module. The Sign Module signs every packet and sends signatures as packet provenance information to the Verify Module which verifies them. If a packet's signature cannot be verified, it is labeled as suspicious, having bypassed the Sign Module, and likely generated by stealthy malware.

Key Management and System Integrity: The key management mechanism and system integrity enforcement are very similar to those of TUBA described in Section 3.4. When the system starts up, the Sign Module and the Verify Module generate their public/private key pairs and notify each other of their respective public keys. Taking advantage of public key cryptography, the two modules securely exchange two symmetric keys; one is for signature generation and verification, while the other is used to encrypt signatures from the Sign Module to the Verify Module.

To ensure that the integrity of the detection framework and signing key secrecy, we utilize the on-chip TPM to generate the signing keys and to attest kernel and module integrity at boot. The approach is similar to TUBA integrity service described in Section 3.4, 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 [3]. Although more complex, under certain assumptions of the secure 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 [36], our solution provides an effective cryptographic alternative that leverages the available trusted computing infrastructure.

Prototype Implementation: We implement our rootkit detection technique in Windows XP. 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

[5]Directly invoking data-link layer functions to send traffic is considerably hard in practice. These functions are also hardware-dependent.

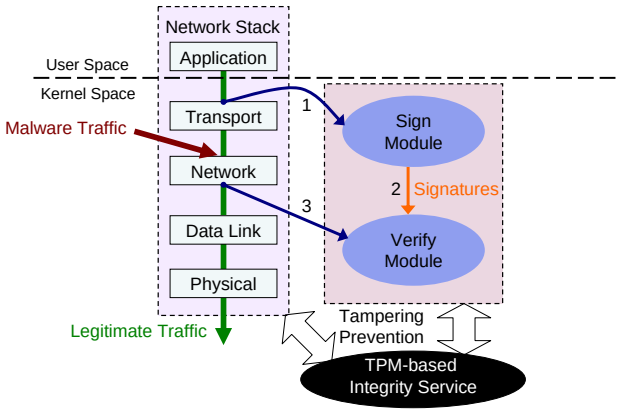

Figure 4: Schematic drawing of components in the framework and their interactions with the host's network stack. Legitimate traffic origins from application layer whereas rootkit traffic is injected into the lower layers.

of the network layer. It intercepts and verifies all packets just before they are sent to network interface card drivers. In this prototype, the signature algorithm is UMAC (message authentication code using universal hashing) which is fast and lightweight [23]. 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 encrypted signatures directly to Verify Module as shown in Figure 4. Signatures are kept in a hash table indexed by packet source address, destination address and port for fast lookup.

Experimental Evaluation: We first test 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.

Figure 5 shows the network throughput with and without using our rootkit detection mechanism. With provenance verification on each packet, the throughout decreases in general. However, as the packet size grows (e.g., 64KB), 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 (home) PCs have low upstream traffic even with peer-to-peer (P2P) applications running.

Summary on traffic provenance verification: Our above-described detection framework enforces the correct flow of

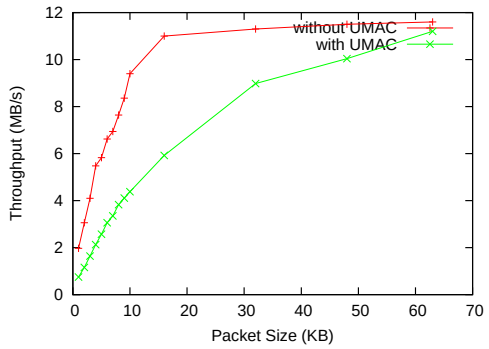

Figure 5: Performance comparison with or without the provenance-verification based traffic detection.

outbound traffic through the host's network stack. This feature can be used to realize 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. We feel that this contribution is beyond the specific rootkit problem studied.

7. Related Work

TUBA is orthogonal to existing traffic-based botnet detection tools, which makes integration easy. The detection results produced by other means may serve as triggers (see more in Section 2.1) to invoke a remote authentication session. For example, TUBA can start a verification test for the user whenever BotSniffer or BotHunter identify suspicious communication patterns. Note, however, that TUBA does *not* rely on existing botnet detection solutions to work because the verification tests may be launched periodically or according to the trigger events defined by TUBA, as previously explained in Section 2.1.

It is worth mentioning that there exists a fundamental difference between TUBA and CAPTCHA, which is a technique that attempts to differentiate between humans and machines on visual ability [40]. *TUBA's challenges are personalized and individualized*, whereas CAPTCHA challenges are generic. TUBA is a fine-grained authentication and identification framework, where CAPTCHA is a coarse-grained classification mechanism.

The work that is most related to ours is the recently-proposed Not-A-Bot system [15]. As mentioned earlier, our TUBA design also aims to distinguish among different users which provides a more fine-grained classification. In addition, we only require key event signing and verification when the user responds to a TUBA challenge, which makes TUBA very efficient in practice.

Flicker is a recently-proposed trusted computing base for allowing sensitive applications to run in isolation in an untrusted operating system [25]. 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. Our design supports integrity service without requiring the suspension of the operating system, which is required by Flicker [25].

Existing rootkit detection work largely focuses on operating system level detection, including identifying suspicious system call execution patterns [7], discovering vulnerable kernel hooks [41], exploring kernel invariants (e.g., Gibraltar [3]), or using virtual machine to enforce correct system behaviors [10, 36]. For example, Christodorescu, Jha, and Kruegel collected malware behaviors like system calls and compared execution traces of malware against benign programs [7]. 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 [10]. Wang *et al.* systematically identified potential kernel hook points in Linux kernel [41]. Although existing OS level detection methods are quite effective, they typically require sophisticated and complex examination of kernel instruction executions. Additionally, to enforce the integrity of the detection systems, a virtual machine monitor (VMM) is usually required, as in [36]. In comparison, we demonstrate a cryptographic provenance verification approach leveraging existing trusted computing infrastructure (the TPM is available on most commodity computers) for detecting stealthy rootkit traffic.

Kirda *et al.* detected malicious Internet Explorer extensions (used by spyware) by analyzing their behaviors [22]. Their approach is based on the fact that spyware usually first gets sensitive information from the browser and then sends the information to the outside. Our rootkit traffic detection work has a different assumption on malware capabilities; we enforce the legitimate traffic flow using a cryptographic approach, whereas their work characterizes spyware-like behavior through dynamic and static analysis.

8. Conclusions and Future Work

We described a cryptographic provenance verification technique for host-based malware detection. We illustrated how this technique can be used to leverage human-malware differences in the detection. Namely, two questions were addressed: *how to select characteristic behavior features*, and *how to prevent malware forgery*. Our methods for keystroke-based bot identification and rootkit traffic detection were used to demonstrate the applications of the cryptographic provenance verification technique.

We first presented our design and implementation of a remote authentication framework called TUBA for monitoring a user's typing patterns and verifying their integrity. We evaluated the robustness of TUBA through comprehensive experimental evaluation including two series of simulated bots. We then demonstrated our provenance verification approach in realizing a lightweight framework for blocking outbound rootkit-based malware traffic.

For future work, we plan to extend our cryptographic provenance verification approach to develop advanced input-traffic correlation and tracking analysis. 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 and P2P file sharing in the face of sophisticated malware exploits.

String	Female		Male		Both	
	TP	FP	TP	FP	TP	FP
google.com	93.68%	5.56%	92.00%	5.50%	91.86%	4.53%
www.amazon.com	94.00%	4.46%	94.71%	4.62%	91.71%	2.89%
1calend4r	92.29%	5.69%	92.57%	7.51%	89.29%	4.48%
name1@gmail.com	96.26%	2.90%	95.14%	3.17%	94.00%	2.26%
name2@gmail.com	95.29%	3.68%	96.00%	2.90%	94.43%	2.79%

Table 3: Human vs. human true positive(TP) and false positive (FP) SVM classification results. Real email addresses are anonymized.

APPENDIX

Our Experiment 1 in TUBA evaluation is described first. Then the NoiseBot algorithm is presented.

Experiment 1 (Human vs. Human) Among the 20 users, we set up a basic SVM test to see if our classification algorithm can distinguish each user from the others. Three different classification sets c_i, $i = 1, 2, 3$ for each word were created according to the users' gender: $c_1 = \{\text{all male instances}\}$, $c_2 = \{\text{all female instances}\}$, and $c_3 = c_1 \cup c_2$. The class i experimental setup of word s_l for user u_j was then performed as follows:

- Label each of the user's 35 instances as `owner`,
- Pick 5 random instances for every user $u_k \neq u_j$ whose instances are in the set $\{c_i\}$ and label them as `unknown`,
- Given the relabeled instances, perform a 10-fold cross-validation for SVM classification (manually adjusting the model parameters).
- Calculate the average TP and FP rates.

The classification analysis was repeated for all the user subjects, words in the database and classification sets. Finally, the average TP and FP rates for every word and class (1. male, 2. female, and 3. both) were calculated and the results are summarized in Table 3 – the average false positive rate of 4.2% confirms the robustness of using keystroke dynamics for authentication.

In general, the performance across the different classes had little effect on the performance of the SVM classifier. We note, however, that the familiarity and length *do* affect the results. From Table 3 we can see that less familiar strings such as `1calend4r`, have a lower true positive rate than the more familiar strings, like `www.amazon.com`. This is because the user is still not very comfortable with the string and the variance (which in this case may effectively be considered *noise*) in the feature vectors is quite high.

On average, the true positive and false positive rates of the longer strings (`name1@gmail.com` and `name2@gmail.com`)[6] perform better because the users have an additional "freedom" to demonstrate their unique typing style; since the strings are very long some users pause (unconsciously) mid-word, which is reflected by some of the inter-key timings.

The NoiseBot Algorithm Similar to Algorithm 1 presented in Section 4, a the pseudocode for a bot which generates noisy instances (i.e., mean ± noise) is shown in Algorithm 2. The parameters for Experiment 3 were calculated as those for GaussianBot in Experiment2, with the noise parameters $\eta_i = \sigma_i/2$ and $\eta_i' = \sigma_i'/2$.

[6]Real email addresses are anonymized for blind review.

Algorithm 2: *NoiseBot* simulation of a human

input: string=$\{x_1, x_2, \ldots, x_N\}$,
durations=$\{(\mu_1, \eta_1), (\mu_2, \eta_2), \ldots, (\mu_N, \eta_N)\}$,
inter-key
timing=$\{(\mu_2', \eta_2'), (\mu_3', \eta_3'), \ldots, (\mu_N', \eta_N')\}$,
n=number of words to generate

for $n \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** N **do**
    SimulateXEvent(`KeyPress`, x_i);
    $X_i \leftarrow \mu_i + \mathcal{U}(-\eta_i, \eta_i)$; `/* key duration */`
    if $X_i < 0$ **then** $X_i \leftarrow 0$; `/* adjust for large noise */`
    Sleep(X_i);
    SimulateXEvent(`KeyRelease`, x_i);
    $X_i' \leftarrow \mu_i' + \mathcal{U}(-\eta_i', \eta_i')$; `/* inter-key timing */`
    if $X_i' < 0$ **then** $X_i' \leftarrow 0$; `/* adjust negative timing */`
    Sleep(X_i');

A. References

[1] D. Annicchiarico, R. Chesler, and A. Jamison. Xtrap architecture. *Digital Equipment Corporation, July*, 1991.

[2] 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.

[3] 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.

[4] C. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

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

[6] S. Bleha, C. Slivinsky, and B. Hussien. Computer-access security systems using keystroke dynamics. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(12):1217–1222, 1990.

[7] 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.

[8] C. Comin. LKL linux keylogger.

http://sourceforge.net/projects/lkl/.

[9] 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.

[10] 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.

[11] S. Gianvecchio, M. Xie, Z. Wu, and H. Wang. Measurement and classification of humans and bots in internet chat. In *Proceedings of USENIX Security Symposium*, 2008.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] T. Hastie and R. Tibshirani. Classification by pairwise coupling. In M. I. Jordan, M. J. Kearns, and S. A. Solla, editors, *NIPS*. The MIT Press, 1997.

[17] N. Ianelli and A. Hackworth. Botnets as a vehicle for online crime, 2005. `http://www.cert.org/archive/pdf/Botnets.pdf`.

[18] J. Ilonen. Keystroke dynamics. *Advanced Topics in Information Processing–Lecture*, 2003.

[19] 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.

[20] S. Keerthi, S. Shevade, C. Bhattacharyya, and K. Murthy. Improvements to Platt's SMO algorithm for SVM classifier design. *Neural Computation*, 13(3):637–649, 2001.

[21] K. S. Killourhy and R. A. Maxion. The effect of clock resolution on keystroke dynamics. In R. Lippmann, E. Kirda, and A. Trachtenberg, editors, *RAID*, volume 5230 of *Lecture Notes in Computer Science*, pages 331–350. Springer, 2008.

[22] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

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

[24] B. Lampson, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, 1992.

[25] 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.

[26] Kernel Based Keylogger. `http://packetstormsecurity.org/UNIX/security/`.

[27] F. Monrose and A. Rubin. Keystroke dynamics as a biometric for authentication. *FUTURE GENER COMPUT SYST*, 16(4):351–359, 2000.

[28] J. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*, chapter 12. 1998.

[29] T. F. D. Project. *FreeBSD Handbook*, 2008.

[30] rd. Writing linux kernel keylogger. *Phrack magazine*, 12(14), jul 2002.

[31] R. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

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

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

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

[35] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and SSH timing attacks. *Proceedings of the 10th USENIX Security Symposium*, 2001.

[36] 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.

[37] J. Stewart. Top spam botnets exposed, April 2008. `http://www.secureworks.com/research/threats/topbotnets`.

[38] TCG PC Client Specific TPM Interface Specification (TIS), Version 1.2. Trusted Computing Group. `http://www.trustedcomputinggroup.org/groups/pc_client/`.

[39] tlogger. http://dubroy.com/tlogger/.

[40] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *Commun. ACM*, 47(2):56–60, 2004.

[41] 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.

[42] I. H. Witten and E. Frank. *Data Mining: Practical*

Machine Learning Tools and Techniques. Morgan Kaufmann, San Francisco, 2 edition, 2005. WEKA available at `http://www.cs.waikato.ac.nz/ml/weka/`.

[43] G. Xu, C. Borcea, and L. Iftode. Satem: Trusted service code execution across transactions. *Reliable Distributed Systems, IEEE Symposium on*, 0:321–336, 2006.

[44] E. Yu and S. Cho. Novelty detection approach for keystroke dynamics identity verification. *LNCS*, pages 1016–1023, 2003.