HumanSign: Accurate Bot Message Detection with Reliable Human Attestation

Muhammad Asim Jamshed, Younghwan Go, and KyoungSoo Park

Department of Electrical Engineering KAIST

Technical Report, 2012

ABSTRACT

Malicious bot traffic has long been a serious threat to the stability and reliability of the Internet. Despite continued efforts and best practices, network attacks by automated bot messages are consistently on the rise. These attacks include email or instant messaging spams, remote password cracking, and even distributed denial-of-service attacks by coordinated botnets, just to name a few.

In this paper, we envision a new network environment where we can deterministically identify the human traffic from the bot traffic. We design a reliable human-attested message creation framework that proves the human existence for the messages that travel over the Internet. By tightly binding human typings to message composition, we have each message carry a human attestation and allow the remote party to verify the identity of the traffic source. Unlike previous works, our framework removes the chance of attestation forgery by establishing a secure path from the input device to the human message attester. For this, we draw the root of trust from the input hardware and a trusted platform module (TPM), and securely extend it to the software attester using the late launch capability available in modern processors. Our measurements show that the human attestation process takes about 610 milliseconds for a typical email, an average instant chat message, or an SSH password login attempt. We find that the majority of the attestation delay is contributed by relatively slow TPM operations and late-launched code execution, which we expect will improve as the demand for trusted computing technology grows.

1. INTRODUCTION

Automated bot messages have extensively permeated our Internet life. Email spamming shows no sign of reduction as almost 9 out of 10 messages in 2011 were estimated to be spams [28], and the cost for spam emails amounts to $130 billion around the world in 2009, a 30% increase from 2007 [9]. Instant messaging (IM) spams have also been prevalent as 1 in every 78 hyperlinks in chat messages leads to advertisement sites [42]. Popular blogs and social networking sites such as Facebook [13] and Twitter [47], and even video sharing sites like YouTube [54] constantly fight against massively-automated bot messages.

In response to these attacks, the spam detection technology has greatly improved over time. Gmail says that its spam filtering rate is over 99% as many of commercial spam filtering tools report similar statistics [16]. While the deployment of effective filters significantly reduces the spams, it leaves the chance of false positives where legitimate emails are misclassified as spam. Gmail reports the false positive rate as high as 1% [16], and what is worse is that the problem is not limited to email as social networking site users are often inflicted by incorrect filtering [14]. As a result, reliable messaging on the Internet still remains as a challenging problem.

Human attestation is an emerging technology that can reliably eliminate these bot attacks [15, 19, 34]. Human-attested messages carry an unforgeable proof that verifies human existence at the time of message creation. Using the proof, servers can enforce their own policy to screen unwanted bot traffic based on the identity of the traffic source. Typically, the human activity is inferred from keystroke events and cryptographic signatures from secure hardware are used as a human attestation proof for the messages. With an accurate human attestation framework, we may envision a network environment free from bot-generated messages or at least we can enforce a tighter control over the bot traffic.

Not-a-Bot (NAB) [19] is one of the first works that associates network messages with human activity. NAB uses a prototype attester running under a trusted hypervisor to generate human attestation proofs when there is a key or mouse event within a configurable time window (e.g., 1 second). However, NAB fails to meet an important security property: it does not provide *tight binding* between the message and the input events. It allows adversarial malware to easily generate attestation proofs even if the input event is not used to produce the intended message. Simply put, any keystrokes and mouse clicks for arbitrary tasks (e.g., word processing) can be abused to generate a human attestation proof for bot content. More recently, UTP [15], a secure transaction confirmation architecture provides a more accurate framework by generating the proofs to only the messages that the human user wants. However, similar to CAPTCHA tests, the system requires manual cooperation by the human user for each attestation, which is tedious and inconvenient to be applied to all network applications.

In this work, we present the design, implementation and evaluation of HumanSign, a reliable human message attestation framework for network applications without manual human intervention. HumanSign strives to achieve following two security properties. First, it securely generates a human attestation proof that is tightly bound to the composed message itself. Only when the message is confirmed to be produced from relevant hardware keystrokes, our framework allows a human attestation for the message. Second, HumanSign provides a guarantee that no attackers produce valid attestation signatures unless they run the legitimate attester with hardware-generated input events. To minimize the size of the trusted computing base (TCB) and to isolate the human message attester from the rest of the system, we run the attester in a Flicker [25] session via late launch [11] and have the on-board TPM attest to the human message and the integrity of the attester with its quote operation.

HumanSign places the root of trust on the input devices and a TPM. One requirement for secure message binding is the ability to prove if each input event is actually generated from the input device. For this, we slightly update the input devices so that they

generate a key proof (e.g., HMAC-SHA1 hash of the scan code and its timestamp) for each input event, which is later self-verified when there is a human attestation request from an application. While this may be seen as an invasive modification, given the trend of putting cryptographic operations into wireless keyboards [39], we believe the security benefit outweighs small extra manufacturing cost.

The overall attestation procedure is described as follows. (a) The client application that participates in HumanSign records all relevant input keycodes and their proofs for a human message. (b) When it needs a human attestation for the message, it launches a valid attester in a Flicker session by feeding the key proofs along with the message. (c) The human message attester first asks the input device to verify the key proofs and when they are confirmed to be valid, it requests a TPM quote signature on the message, its metadata, and the attester hash. (d) Finally, the application sends the message with the signature to the destination, and the receiving end verifies the signature and enforces its own message accepting policy.

We provide the comprehensive implementation of HumanSign on an Intel system that supports *dynamic root of trust for measurement* (DRTM) with the late launch capability. Our prototype runs on a regular Linux operating system (linux-2.6.30.6) with about 4,600 lines of the attester TCB and 205 lines of the patched keyboard driver code to emulate the input key proof generation and verification. Our evaluation shows that it takes 612 milliseconds to generate human attestation proofs for typical email messages and 609 milliseconds for IM chat messages, which is reasonable for an interactive environment. We find that the majority of the delay comes from employing the trusted computing hardware: late launch execution and TPM operations whose latency could improve as the technology further matures.

2. GOALS

HumanSign wants to provide a reliable communication environment by filtering unwanted bot messages with human attestation. We identify the design goals for a human attestation framework.

Transparency. The attestation process should be transparent to the end users. HumanSign should allow automatic attestations of human contents without any manual human verification such as in the CAPTCHA tests.

Accuracy. A human attestation proof should attest to the human activity involved in creating the intended message. Attackers should not be able to generate a valid human attestation nor reuse an existing attestation proof for an arbitrary message.

Generality. The framework should support all interactive network applications that deal with human-typed messages. Also, the attestation should not take too long to disrupt the interactivity.

Minimal TCB. The TCB of the framework should be small to minimize any potential security holes [7, 30] and to facilitate quick formal analysis on the source code of the attester. Also, it should require minimal change in the network applications to benefit from the framework. For this reason, we avoid the trusted hypervisor approach using the virtualization techniques that provide sandboxed execution environments for untrusted applications [10, 43].

Easy deployment. The framework should not depend on specific OS or system software features. Any system that is equipped with secure input devices, a system TPM, and the late launch capability (e.g., available in Intel TXT [11] and AMD SVM extensions [4]) should be able to benefit from HumanSign.

We design the framework to be incrementally deployable such that any participating clients and servers reliably exchange human messages and filter unwanted bot traffic. If one party does not employ HumanSign, it will fall back to existing filtering mechanisms. This approach should benefit a number of message-based network applications such as Email, SSH logins and commands, IM chat clients, and HTML form submissions in Web browsing. While our framework depends on new input device features for keycode validation, many wireless keyboards already encrypt the keystrokes using an algorithm like AES and have non-volatile memory [39]. We believe that extending it to support keycode validation will not add up much cost.

3. THREAT MODEL

We consider the traditional client and server communication model; the client generates messages with a human attestation, and the server, the message recipient, verifies the attestation proof and conditionally accepts the messages. We assume that HumanSign applications run on a client machine with trusted input devices (called *trustworthy* input devices), an on-board TPM and the processors that support DRTM. Trustworthy input devices run small embedded software, called keycode daemon, that converts regular scan codes into secure keycode events as described in Section 4.1. The daemon is also used to verify the secure keycode stream during the attestation process. The input devices, the TPM, and the late-launched attester are trusted, and we assume that users do not mount hardware attacks on the input hardware or the TPM.

The rest of the system (*e.g.*, BIOS, bootloader, OS and client applications) can be compromised by adversaries. The machine can be infected with malware that monitors and intercepts the key events from the input devices. Bots can attempt to assemble a message out of the input keys, but they cannot generate valid input events by themselves. While HumanSign ensures accurate human attestations, it does not defend against other attacks such as leaking sensitive information to external entities, or causing damage to the user data or to the client software. HumanSign does not provide any protection if malware obstructs the human attestation process and causes a denial-of-service attack on the client applications. In that case, the user could notice the problem and run anti-virus software or reinstall the system to remove the malware. Finally, HumanSign does not guarantee message integrity, but one can use existing protocols like SSL/TLS to prevent message tampering or reordering.

4. HUMANSIGN DESIGN

The HumanSign framework consists of three components: (*i*) a trusted client-side component that includes trustworthy input devices and the software attester, (*ii*) untrusted constituents of the client host that include the OS, and the HumanSign-aware applications that produce human-typed messages, and (*iii*) the remote server that receives client messages and verifies the authenticity of the human attestation proof.

A human user types in a message using an untrusted client application, and when the application needs a human attestation for the message, it launches the attester in a Flicker session. Candidate client messages can be any typed messages such as emails, IM contents, SSH commands, typed URLs in a web browser's address bar, and HTTP POST messages. The attester checks whether the message is indeed generated by the trustworthy input devices and issues a TPM-based digital signature over the message and its metadata. The client application sends the message along with its signature to the server, and the server verifies the signature and enforces its own message filtering policy.

4.1 Secure Keycode

To support tight binding between a message and input events, we have each input keycode carry a *key proof* that verifies the source of the input hardware. Figure 1 shows the format of the *secure keycode*

scan code (e) 1-3 bytes	timestamp (t) 6 bytes	key proof (p) 20 bytes

Figure 1: Secure keycode format. A typical keyboard scan code size ranges from 1 to 3 bytes.

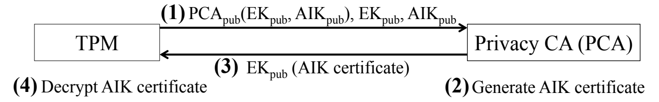


Figure 2: AIK certification. A fresh AIK public key is certified (one-time operation) before starting the attestation process.

that extends the existing scan code (e) with a timestamp (t) and a key proof (p). t records the generation time of the input event in the millisecond granularity (6 bytes to represent milliseconds passed from the unix epoch). Timestamps prevent malicious bots from replaying old keycode and differentiate the input events with the same scan code value. p is a one-way cryptographic hash of the scan code and the timestamp, namely, HMAC-SHA1(K_s, $\langle e|t \rangle$), where K_s is a 20-byte secret key that is generated inside the input device and never leaves it. We note that the secure keycode brings up to a 27 times blowup from the original scan code size. However, a secure keycode is used only between the input device and the application on a local machine. Since the keycode generation is bound by the rate of human typing, we believe that the size overhead is bearable given the trend of increasing memory capacity and CPU power.

The trustworthy input device can be implemented as an embedded system with a low-powered processor and small non-volatile memory. It runs a keycode daemon responsible for converting each scan code to a secure keycode and for verifying the integrity of the generated keycode when requested by the attester. The daemon also manages the secret key (K_s) for the key proof. The secret key rotates every n days while keeping the previous secret key in the input device for verifying old keycode streams. A secure keycode older than $2n$ days is assumed to be invalid for the purpose of human attestation. n is configurable but we believe 30 days is a reasonable number, which allows human attestation for the messages that are typed within 60 days.

4.2 Human Message Attestation

The attester in HumanSign is responsible for achieving three objectives. First, it validates and vouches for human presence during target message composition. Second, it ensures that only the application that produces the message gets the corresponding attestation. Third, it guarantees to the remote server (the message recipient) that the attestation it generates is trustworthy. That is, malware on the client machine should not be able to influence the attester nor forge the attestation to circumvent the human verification process of the message by the remote server.

Before explaining the detailed attestation process, we provide a brief background on the TPM that is used to launch the Flicker session as well as to generate a human attestation.

4.2.1 TPM Background

A TPM is an inexpensive security chip used for a number of security operations such as key (both symmetric and asymmetric) and random number generation, secure data measurement, sealed storage, remote attestation and so on. Two TPM operations that we use are *extend* and *quote* that employ a set of TPM registers called Platform Configuration Registers (PCRs). We denote $m \leftarrow SHA1(data)$ as storing a one-way cryptographic hash of the data into a PCR, m. The TPM extend operation is defined as iterative hashing like $PCR_x \leftarrow SHA1(PCR_x \| data)$. The TPM quote operation digitally signs a set of target PCRs that have the relevant measurements. Before a quote, the TPM first generates a 2048-bit RSA key pair called Attestation Identity Key (AIK) and uses its private key to sign the PCRs. The public key of the AIK is certified by the TPM vendor's certifying authority or Privacy CA as shown in Figure 2. When a TPM generates an AIK, it encrypts the public keys of the AIK and the root Endorsement Key (EK) (that is burned into the TPM when it is manufactured and its private key never leaves the TPM) with the public key of Privacy CA and sends the bundle to Privacy CA. Privacy CA decrypts the bundle, checks if the public key of the EK is valid since it holds the database of EK public keys of all published TPMs. It then signs the public key of the AIK to generate an AIK certificate and returns the AIK certificate encrypted by EK's public key to the TPM. This process guarantees that only a valid TPM produces an unencrypted AIK certificate. With this AIK certificate, a remote party can validate the TPM quote signature.

According to the TPM v1.2 specification, a TPM includes 24 PCRs divided into *static* and *dynamic* sets. The static PCRs (0-16) can be used for measurements, but the dynamic PCRs (17-23) can be measured or reset only if the CPU is in the late-launched state.

4.2.2 Late-Launched Environment with Flicker

HumanSign runs the attester in a Flicker session that employs the DRTM using the late launch CPU capability. Late launch essentially micro-reboots the machine by temporarily suspending the running OS. It creates a completely isolated execution environment for security-sensitive code by disallowing direct memory access (DMA) to physical memory by random devices and by disabling hardware interrupts to prevent previously executing code from regaining the control and interfering with executing the security-sensitive code. A special CPU instruction (*GETSEC[SENTER]* and *SKINIT* for Intel and AMD CPUs respectively) initiates late launch, which re-initializes all CPU and register states for a fresh execution environment, and resets dynamic PCRs of a TPM to 0. Then, Intel TXT extends the measurement of the security-sensitive code into PCR_{18}. This allows the remote party to verify which piece of code was actually running in the late-launched environment.

Flicker allows running the security-sensitive code with minimal TCB in the late-launched environment by removing the dependency on the large virtual machine monitor as TCB. It provides small stub code (called SLB Core) that sets up and tears down a Flicker session and that performs necessary steps to make sure that the right code is running. HumanSign uses Flicker to abstract the steps needed for late launch as well as to benefit from minimal TCB. Flicker also provides a few TPM and cryptography libraries that HumanSign uses in the late-launched environment.

4.2.3 Content for Attestation

The HumanSign application manages the typed messages for human attestation. It is responsible for collecting the secure keycodes corresponding to the portion of the message whose content is later attested by the HumanSign framework. One challenge here is that some part of the legitimate message is not directly typed by a human user. For example, replying to an email may include the received content or a mail client could automatically add text decoration typically found in the rich text format (RTF). Copied and pasted text may not have secure keycodes or may include invalid keycodes that have expired.

To address the problem, HumanSign employs two techniques.

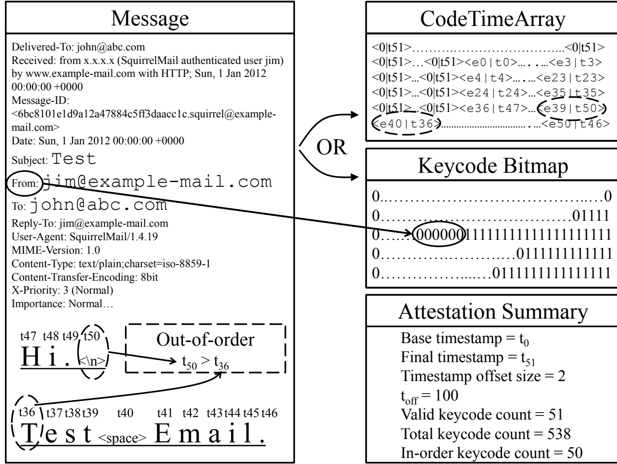


Figure 3: An example of human message attestation. Given the message and the corresponding secure keycodes, the attester first verifies the keycodes, and generates an attestation summary and a CodeTimeArray (or a keycode bitmap) that identifies the portion of typed keycodes.

First, the application identifies and attests to only the portion of the text whose keycodes are valid. Other parts are replaced by *null* keycodes ($\langle 0,0,0 \rangle$) for the purpose of attestation. In our email analysis in Section 6.4, we find that 90% of human-typed emails have at least 13 valid keycodes in the mail body, which still provides significant evidence of human presence. To identify which characters are typed by a human user, HumanSign generates and signs the bitmap of valid keycodes for the message (e.g., 0: not typed, 1: typed). For better accuracy (possibly at the cost of privacy), the application optionally chooses to send the array of the timestamps of the entire message. We call it *CodeTimeArray* of the message, which consists of the scan codes and offset-based timestamps corresponding to the original secure keycodes. The offset-based timestamp reduces the field size by taking the offset from the oldest timestamp in the message. Second, besides attesting to the typings in the message, HumanSign produces metadata called *attestation summary* that has the typing statistics for the message. Table 1 shows the attestation summary, which includes the start and end times of message composition, total and valid keycode counts, and the number of *in-order* keycodes that are typed in the same order as that of the text. The attestation summary is signed by the HumanSign attester and is delivered as extra information to the remote server. If there are not enough valid keycodes in the message, or if the number of in-order keycodes is too small for the message, the server can choose to fall back to a existing bot filter. Section 6.4 analyzes the issue in detail.

We note that HumanSign attestation brings a size overhead: 580 bytes for a TPM quote signature, 26 bytes for the attestation summary, and either $size(message)/8$ bytes for the keycode bitmap or $size(message) \times (1+n)$ bytes for the CodeTimeArray, where n is the size of an offset-based timestamp. For example, for $n = 3$, CodeTimeArray can cover 19.4 days with the 100 ms offset granularity (t_{off}), which should be enough for most short messages. To reduce the size overhead further, one can compress the attestation results.

4.2.4 HumanSign Attestation Process

Figure 4 shows the step-by-step HumanSign attestation process. (1) The HumanSign client application passes the message, M, and

FIELD	DESCRIPTION
Base timestamp	the oldest timestamp in the keycodes
Final timestamp	the latest timestamp in the keycodes
Timestamp offset size	byte count for an offset-based timestamp
t_{off}	offset-based timestamp granularity
Valid keycode count	total # of valid keycodes
In-order keycode count	total # of in-order keycodes
Total keycode count	total # of keycodes

Table 1: Attestation summary: timestamps are 6 bytes, offset size and t_{off} are one byte, and keycode counts are 4 bytes.

its corresponding secure keycodes, S, along with the attester file path to the late launch startup module. (2) The startup module loads M, S, and the attester to a well-known physical memory address, sets up the processor registers for late launch, and de-schedules all processors except the bootstrapping CPU (CPU0). CPU0 saves the current machine states and executes a special CPU instruction that suspends the running OS, and passes the control to the late launch bootstrapping module[1] (known as the SINIT authenticated code module in Intel CPUs). The SINIT module extends the attester, A, into PCR_{18} and its own module, I, into PCR_{17}. Measuring I is necessary for checking the version of the SINIT module since it is often updated against newly-discovered vulnerabilities [51, 52]. (3) The control is transferred to the attester, and it asks the trustworthy input device to validate S with M. If all valid keycodes match those in M, the attester generates the attestation summary, T, and either the valid keycode bitmap, B, or the CodeTimeArray, C, depending on the attestation type. It then extends PCR_{18} with $(M \| T \| B)$ or $(M \| T \| C)$. (4) The attester issues a TPM quote operation on both PCR_{17} and PCR_{18} using the private key of the AIK. Note that PCR_{17} takes on $SHA1(0x00^{20} \| SHA1(I))$ and PCR_{18} has $SHA1(X \| SHA1(M \| T \| (B\ or\ C))$ where $X = SHA1(0x00^{20} \| SHA1(A))$. The quote result would be $AIK_{private}(PCR_{17}, PCR_{18})$, and we call it *content proof* of the message. It is temporarily stored by the attester before returning to the application. (5) Finally, the attester extends both PCR_{17} and PCR_{18} with a random nonce to prevent further TPM quote operations over the same measurements. Then, it passes content proof, attestation summary, and keycode bitmap or CodeTimeArray back to the application. If attestation fails for any reason, an error code is returned instead. When late launch exits, the startup module restores the previous machine state and resumes the normal OS.

One potential problem lies in how we authenticate the trustworthy keyboard in the protocol. An attacker may attempt to build his own malicious keyboard that validates every keycode proof that his colluding bot on the local host randomly generates. One way to prevent this attack is to use the similar authentication technique employed by TPMs. The keyboard manufacturer can assign an asymmetric key pair to the keycode daemon and register the public key with Privacy CA. Whenever the keycode daemon handles the keycode validation request, it signs the result with the private key, and the attester verifies it. This inevitably increases the attestation latency (about 300 ms when you use a TPM inside the keyboard), but it would ensure the authenticity of the keycode validation.

Another concern is that HumanSign may hamper user privacy. While we can anonymize the AIK of the TPM with direct anonymous attestations (DAA) [8], the HumanSign attestation intentionally exposes which character in the message is typed by a human user. We believe this fine-grained typing information is the key to

[1] AMD does not require an additional bootstrapping driver as the built-in SVM extension alone executes the measured launch environment.

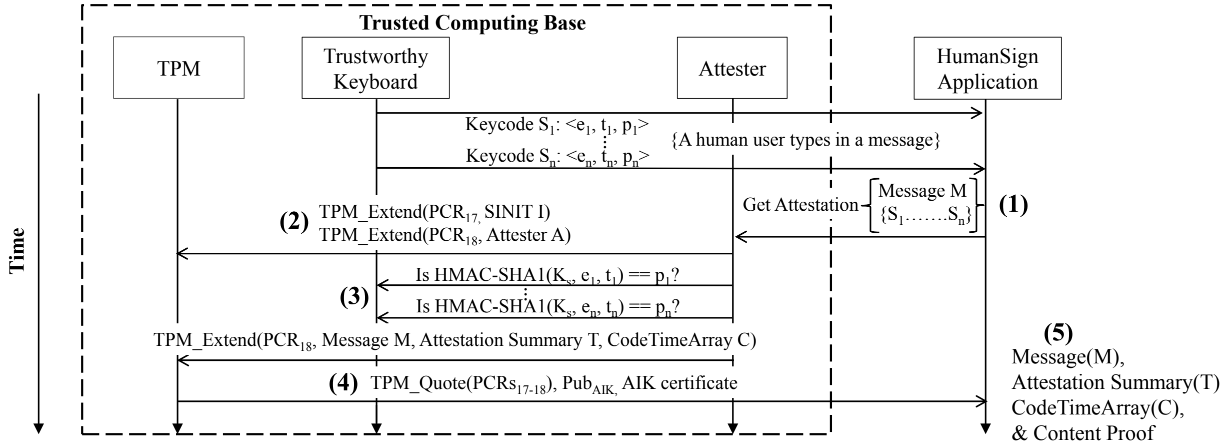


Figure 4: HumanSign attestation process - (1) The HumanSign application sends the keycodes, $S_1 ... S_n$ and message, M to the attester. (2) SINIT & Attester binaries are extended to PCRs 17 & 18 respectively. (3) The keycodes are authenticated via the trustworthy keyboard. (4) The content proof is generated after TPM_Extend and TPM_Quote operations. (5) Finally the message, M is transmitted with the Attestation Summary, T, CodeTimeArray C and the content proof.

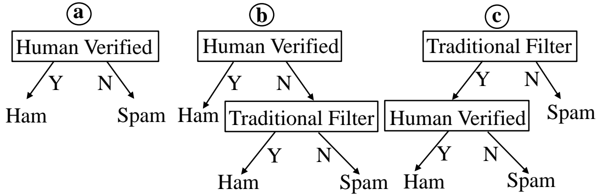


Figure 5: HumanSign-based verification policies - (a) illustrates a filtering policy where only human verification suffices. SSH or IM servers may adopt this policy to prevent logins by bots. (b) shows a policy that uses human verification as the first filtering criterion. It would minimize false positives but some human-typed spams may get through. (c) would minimize false negatives by detecting the human-typed spams and filtering the bot-generated spams by human verification.

differentiating human messages from bot-generated ones. One can choose to use keycode bitmap instead of CodeTimeArray, which excludes the timing information of each keycode. However, anonymizing the sender itself is difficult since the recipient of emails, IM chat messages, SSH commands and etc. typically know the message sender. Also, exposing the hash of the late-launched software stack is necessary to verify the authenticity of the attestation by the remote server.

4.2.5 Attester Management

We envision a universal attester that provides human message attestation to all interactive text-based network applications. A universal attester is possible since the attestation process does not have any dependency on the application context. It can be managed as public software subject to careful security review, and can be indexed by a trusted third-party such as the National Software Reference Library (NSRL) [22], which currently maintains a database of fingerprints of widely-used software applications.

4.3 Human Message Verification

The remote verification process is as follows. The server first checks whether the AIK used to sign the PCRs actually comes from a valid TPM by verifying the AIK certificate. It then determines whether the software stack used in the attestation is authentic. This is verified by validating the content proof with the attester, the SINIT module, and the message. This step would prevent a rogue software agent from invoking an isolated execution environment with the attester code of its own choosing.

Once the verification passes, the remote verifier applies its own filtering policy with supplementary information. For example, if the message composition time exceeds a certain threshold, if there are too many out-of-order keycodes in the message or if the copy and pasted keycodes in a message exceed a threshold, the human verification filter flags the message as suspicious. Figure 5 shows three plausible bot-message filtering policies with HumanSign. Some services (e.g. SSH server) could enforce a strict policy to allow only human-attested messages, but other services (e.g., Email server) may adopt a hybrid approach with traditional content-based spam filters. Human verification can be configured to minimize either the false positives or negatives. Also, in case the client message does not include a human attestation, it can fall back to existing filtering mechanisms, allowing incremental deployment.

We note that HumanSign provides a reliable way to identify any portion of text the human user actually typed in and to segregate any part that is automatically generated or copied and pasted. HumanSign itself does not guarantee to filter all spams (e.g., a spam email can be typed directly by a malicious attacker on his local host!). However, a reasonable policy in the remote verifier would suppress most of spams and deliver human-typed messages reliably to the other party. To assist with the policy, we further analyze the characteristics of human typings in section 6.4.

5. IMPLEMENTATION

We implement the HumanSign framework on an Intel machine with the TXT capability. We modify the USB keyboard driver on Linux 2.6.30.6 to export the secure keycodes directly to HumanSign applications. We also implement a common human attestation library that assists with the client application to invoke the attester in the measured late launched environment. Table 2 shows the number of code lines for each module.

MODULE NAME	LANGUAGE	SLOC
Linux USB Keyboard Driver	C	205
Attester	C/Assembly	4,683
Flicker Startup Kernel Module	C	3,383
Thunderbird Attestation Add-on	XUL/JS/C++	329
Dropbear Attestation Extensions		
Client Attestation Module	C	588
Server Verification Module	C	112
Pidgin-2 Plugins		
Attestation Plugin	C	421
Verification Plugin	C	352

Table 2: Number of source lines of code (SLOC) for human attestation modules and sample application extensions using sloccount [49]. XUL stands for XML User Interface Language and JS is Javascript.

5.1 Secure Keycode Keyboard Driver

As a proof-of-concept, we modify the Linux USB keyboard driver to expose the secure keycodes via the `sysfs` file system interface, which requires 205 lines of driver code change. We also develop a few HumanSign application extensions that pick up the secure keycodes from it. The real trustworthy input devices could run the keycode daemon in the wireless USB transceiver or in the keyboard itself to export the keycodes directly to the device driver. We note that the driver that relays the keycodes to the applications need not be trusted but the attester need to send keycode validation requests to the input devices safely. For easy migration, we have implemented an `ioctl()` function by introducing a new request code number for `/dev/event` device nodes that delivers the secure keycodes so that legacy applications could adapt to our framework with minimal code change.

5.2 Attester Implementation

We implement the attester that handles human message attestation requests. The current attester is 4,683 SLOC where the core logic takes up 1,173 lines while the cryptographic libraries (such as HMAC-SHA1) consume 3,510 lines. In this implementation, we have the attester verify the secure keycodes since the real trustworthy input device is unavailable.

We develop our attester based on Flicker-0.2 while the latest version at the time of writing is Flicker-0.5. We choose Flicker-0.2 for two reasons. First, we want to keep the size of the framework as small as possible[2]. Second, we do not see that Flicker-0.5 improves the performance since it mostly focuses on integrating Intel/AMD and Windows/Linux modules into a unified code base.

When the attester is first installed, the installation program initializes the TPM for HumanSign. It asks the TPM to generate an AIK pair and to load it to the TPM's volatile memory, and it sends the AIK public key to a Privacy CA for certification. We use `privacyca.com` to generate a level-1 X.509 AIK certificate that is later sent to the remote server as part of human attestation proofs.

5.2.1 *Flicker Optimizations for Fast Response*

We find that the Flicker-0.2 module often takes more than a second (sometimes up to 4.3 seconds in our experiments) to execute a single attester session in the late launched environment. We apply two optimizations that significantly reduce the attestation latency. The first technique is to enable the caching mode in the Flicker environment. Currently, Flicker (including the latest version) disables

[2]Flicker-0.5's kernel startup module adds 1,858 more lines of code than that of Flicker-0.2.

TASK	LATENCY
Late Launch Initialization (GETSEC[SENTER])	130 ms
Late Launch Exit (GETSEC[SEXIT])	80 ms
TPM Quote	365 ms
TPM Extends	30 ms
Total	605 ms

Table 3: Latency breakdown for hardware instructions involved in late launch. Before the first TPM_Quote, one-time operation, TPM_LoadAIK_Key is needed to load the AIK key. TPM_LoadAIK_Key takes about 2 seconds.

the caching mode bit in the memory type range registers (MTRRs) managing the memory region that hosts the measured code to be launched. This is mainly because the general-purpose Flicker framework would otherwise have to rely on the untrusted OS to pass the MTRR information that may lead to cache poisoning attacks [36,50]. In HumanSign, wrong MTRR information would not produce a valid content proof since both the attester and the SINIT module are measured and signed, and are verified by the remote server. So, we activate the write-back memory caching mode with sanity checks that guard against wrong MTRR information. We find that caching improves the memory-intensive operations significantly by a factor of 2.5. For example, we could reduce the latency of SHA1 hash calculation of 100K secure keycodes from 725 milliseconds to 291 milliseconds.

The second technique is to adopt lazy re-enabling of the CPU cores when the Flicker session ends. Flicker needs to disable all CPU cores except the bootstrapping processor (CPU0) when it starts the late launch environment. The CPU disabling process collectively takes about 83 milliseconds on our Intel processor. When the Flicker session ends, it has to re-enable all de-scheduled CPU cores. However, we see that re-enabling each core also takes a relatively long time (about 100 milliseconds on the same Intel CPU) and what is worse is that only one core can be enabled at a time. The delays are coming from a series of high latency tasks such as calibrating the frequency of the core, registering the advanced programmable interrupt controller (APIC) and re-scheduling processes onto it. Unfortunately, this presents poor user experience since the system would look frozen during the time. To hide this latency, HumanSign re-enables only one CPU core right after the Flicker session, and postpones re-enabling the rest until the CPU load falls below a certain threshold. We note that this technique does not solve the latency problem completely, but it provides much better experience to the human user.

6. EVALUATION

We evaluate the performance of HumanSign with a few popular text-based application extensions that we have developed. We first measure the delays for keycode validation and total message attestation for typical interactive applications. We then look into the human typing patterns in real email, IM, and SSH messages, which would help configure a reasonable filtering policy for the remote verifier.

6.1 Microbenchmarks

We first show the microbenchmarks with secure keycode validation and attestation performance. We run the tests on an Intel machine with a Core i7 CPU 870 (2.93 GHz), 4 GB physical memory, and an Infineon TPM version 1.2.3.16. All tests are run 10 times and we show the average values. The line in Figure 6 shows the secure keycode validation delays as the message size increases.

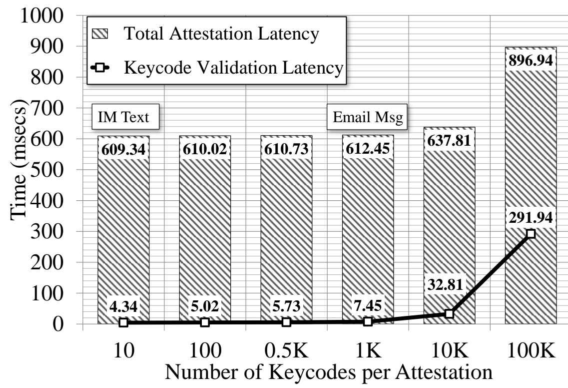


Figure 6: Keycode validation (line) and total attestation (bar) times over various message sizes. Measured on a 2.93 GHz, Intel Core i7 870 machine.

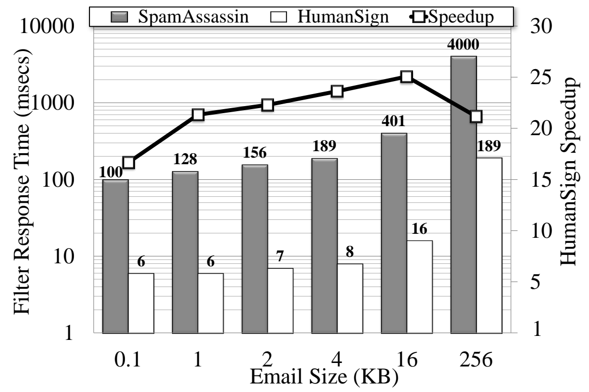


Figure 7: HumanSign verification performance compared with SpamAssassin v3.3.1. Note that the Y scale on the left is logarithmic.

The delay is within 10 milliseconds when the messages size is under 1 KB and it grows more or less linearly when the size is over 10 KB as the HMAC-SHA1 hash calculation dominates the performance. It takes about 32 milliseconds for 10 K keycodes, which is reasonable for most short messages. The bars in Figure 6 present the total attestation response times for the corresponding keycode sizes. We mark the average IM and email message sizes drawn from our user studies in Section 6.4. The results show that the majority of the attestation time is spent on late launch execution involving TPM operations.

Table 3 shows the breakdown of the late launch operation latencies. We find that the TPM operations take up most of the attestation time. A TPM quote takes about 365 milliseconds while an extend operation takes 30 milliseconds. The TPM quote is slow because it needs to calculate a 2048-bit RSA digital signature on a 33 MHz TPM processor. About 35% of the attestation time is spent on the special instructions that initialize and tear down the late launch environment. GETSEC is the Intel processor's late launch instruction where SENTER and SEXIT are the two opcodes that start and exit the late launch environment. Overall, the attestation delay still needs to be improved for highly-interactive applications like IM and SSH, but it is in the reasonable range for email message composition.

6.2 Human Attestation for Email Messages

To apply the human attestation to email, we develop a HumanSign application extension based on Mozilla Thunderbird 3.1.17 [32]. We choose Thunderbird because it is a popular email client with a rich set of developer APIs and libraries, but we believe our extension can be easily adapted to other email clients as well. The extension picks up secure keycodes from the modified keyboard driver each time a human types into the email composition window. The keycodes for the message body and email header fields (e.g., 'To:', 'From:' and 'Subject:') are stored in memory by the extension. It is responsible for keeping all the characters the human types in order even during revisions. The front end of the extension, written in Javascript, handles input events and calls the functions in the back end while the back end uses a XPCOM C++ component to manage the secure keycodes in the edit window and invokes the HumanSign attester in a Flicker session. The extension requires only 329 lines of code. Clicking the 'Send' button initiates the attestation process that generates a content proof, a keycode bitmap or a CodeTimeArray, and the attestation summary for the message. The content proof and the attestation summary are sent under a custom email header, 'X-Attestation:', while the bitmap or the CodeTimeArray is sent as an email attachment to the destination server.

We also implement a HumanSign message verification filter for Postfix 2.5.5-1 [35]. The filter verifies the content proof and classifies the email messages as human-generated or unattested. Figure 7 compares the performance of the Postfix human verification filter with that of SpamAssassin [44], a popular spam filter inspecting the content and the network properties of an message. We configure SpamAssassin with an auto-learning Bayesian classifier. SpamAssassin shows a reasonable performance for small message sizes, but its overhead becomes noticeable for large messages. HumanSign curbs the delay at 189 milliseconds for all message sizes, showing a factor of 16 to 25 latency improvement over SpamAssassin.

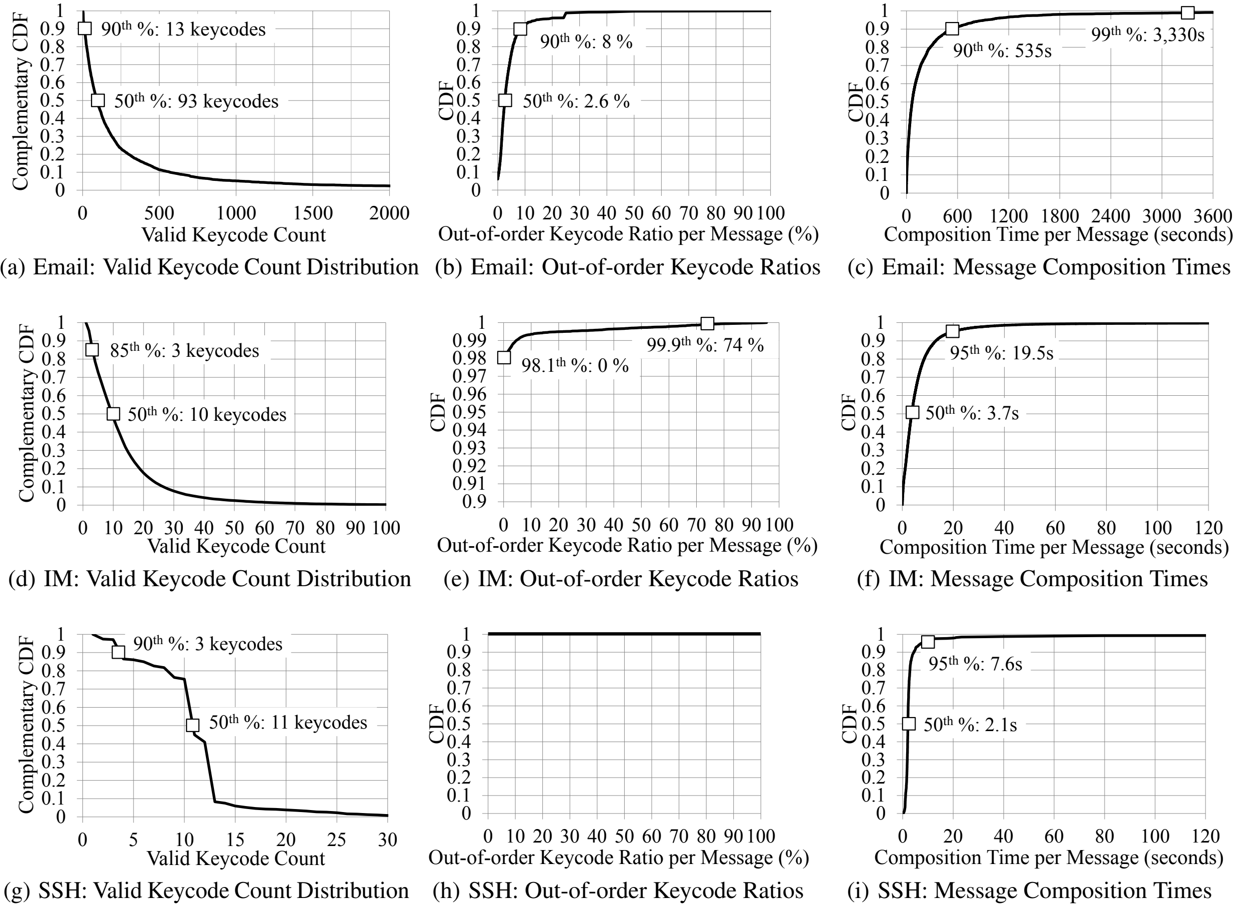
6.3 Human Attestation for SSH and IM

We have implemented HumanSign attestation/verification extensions for the Dropbear SSH client/server suite [12] and Pidgin IM messenger [2]. These two applications bear the similarity in that their message sizes are typically small and the sessions are highly interactive. The extensions share most of the code as a library and support two operational modes - (i) for attesting to the human typings only for password/key passphrase at login and (ii) for all subsequent commands/messages for enhanced security at the cost of slow operations.

Human-typed password attestation: The extension requires that each login attempt is coupled with a human proof. For this, we have the extension call the attester right after the user enters the password. Message attestation requires the human user to physically type in the password/passphrase, which would prevent automated brute-force password cracking attacks.

Per-line attestation: In more secure environments, the client can be asked to attest to the typings of each message line. This would prevent the authenticated SSH/IM sessions from being abused by an automated system after login, but it could harm the interactivity due to long attestation delays. One optimization that we have not applied yet is to postpone the attestation for a few seconds or until enough keycodes are gathered over multiple messages/commands.

If the Nagle's algorithm [33] is turned off, each typed character in an SSH session can be sent in a different packet. To avoid per-character attestation, we link the human attestation with each message. That is, we have the extension collect the secure keycodes till it meets a newline character, and generate the attestation for the full command. The server, on the other hand, maintains a per-connection state machine and validates each command against a human proof before executing it. The HumanSign extensions require



(a) Email: Valid Keycode Count Distribution (b) Email: Out-of-order Keycode Ratios (c) Email: Message Composition Times

(d) IM: Valid Keycode Count Distribution (e) IM: Out-of-order Keycode Ratios (f) IM: Message Composition Times

(g) SSH: Valid Keycode Count Distribution (h) SSH: Out-of-order Keycode Ratios (i) SSH: Message Composition Times

Figure 8: Distributions of human typing pattern statistics. Email: (a), (b), (c) / IM: (d), (e), (f) / SSH: (g), (h), (i)

STATISTICS	EMAIL	IM	SSH
Total Number of Messages	4,552	39,617	670
Average Message Size (bytes)	1,215	14	10
Median Message Size (bytes)	523	10	10
Average Composition Time (secs)	278	9.1	2.6
Median Composition Time (secs)	66.3	3.7	2.1
Average Modified Words per Message (%)	1.3	0.2	-

Table 4: Overall statistics on human-typed messages

588 and 421 lines of code change for the SSH and Pidgin clients, and 112 and 352 lines for the verifying IM and SSH servers. The implementation took less than two weeks while the majority of the time is spent on learning the SSH/XMPP protocol details.

6.4 Human Typing Patterns

To help set up reasonable filtering policy, we have measured typing patterns of human users in emails, IM messages, and remote SSH sessions. We develop customized Mozilla Thunderbird and Pidgin that measure the timestamp of each key event as a user types in the message. We also modify OpenSSH [1] clients to log the timing information of typed password characters. We distribute the extensions to 13 Thunderbird users, 8 Pidgin users, and 9 SSH users and have monitored the typing habit for 12, 6, and 1.5 months respectively. We have collected 4,552 emails, 39,617 IM messages, and 670 SSH login (failed and successful) attempts in total for analysis. The overall statistics are shown in Table 4.

Figure 8(a) shows cumulative percentages of emails that have at least $X = n$ valid secure keycodes in the message. We find that half of all emails have 93 or more valid keycodes while 90% of them have at least 13 valid keycodes. Not surprisingly, the number of valid keycodes tends to increase as the message size grows, but we do see some large messages have only a few valid keycodes, which we suspect were forwarded or replied with a short note. Figure 8(b) shows the fraction of out-of-order keycodes in the collected emails. 50% of all emails have at most 2.6% out-of-order keycodes while 90% of them carry at most 8% out-of-order keycodes. These seem to be mostly for fixing typos in an email. Figure 8(c) indicates that 90% of the emails are written within 535 seconds while 99% finish within 3,330 seconds.

Pidgin messages tend to be much shorter than emails with a smaller valid keycode count per message. It is reflected in Figure 8(d), which shows that 15% of all messages have at most 3 valid keycodes. However, few characters are typed out of order as only 1.9% of all messages include non-zero out-of-order characters, as shown in Figure 8(e). This implies that the IM users do not fix typos nor change words as frequently as in email, which would simplify the filtering policy of IM-based bot messages. Also, the message composition time is short, with 95% of them typed within 19.5 seconds as shown in Figure 8(f).

The valid keycode count distribution in SSH passwords shows a different shape compared to the exponential size decrease as in email and IM messages. Figure 8(g) shows that most instances

Prototype	Transparency	Generality	Accuracy	TCB (SLOC)	Platform
Not-a-Bot [19]	Yes	Yes	1-sec window vulnerability	∼30K	Xen VM
CAPTCHA [6]	No	Web only	Breakable	N/A	N/A
Kill-Bots [24]	No	Web only	Breakable	N/A	N/A
SORBS [40]	Yes	Email	Zero-day vulnerability	N/A	N/A
Bumpy [26]	No	Web authentication	Yes	∼10K	Flicker
UTP [15]	No	Web only	Yes	∼2.3K	Flicker
HumanSign	Yes	Yes	Yes	∼4.9K	Flicker

Table 5: Comparison with other works with similar goals

are concentrated between 10 and 13 characters, which implies that other keycode counts are due to typing mistakes. It also reflects the user's behavior to have a password long enough to guarantee strong security but not too long for typing convenience. As expected, we see that SSH passwords do not include any out-of-order keycode since users tend to retry rather than correct their password when logging into SSH (Figure 8(h)). Figure 8(i) shows that the password typing time is the shortest among all three services, with 95% of them taking less than 8 seconds.

Based on our dataset, we can configure the servers to accept 99.02% of human-typed emails that have at least 13 secure keycodes whose out-of-order keycode ratio is below 25%, which is typed within an hour. Also, we can cover 99.04% of human IM messages that have at least 2 secure keycodes even if we allow 25% out-of-order keycodes in the message that is typed within 60 seconds. 99.32% of SSH passwords can be accepted if the server is configured to allow only human-typed characters without out-of-order keycodes that are typed within 20 seconds.

7. DISCUSSION

We discuss various attack scenarios and defense strategies here.

Frankenstein attacks: Some intelligent spamming bots build grammatically-coherent messages from a collection of random words [55]. In HumanSign, bots may attempt to glean secure keycodes by eavesdropping legitimate keystrokes, and try to generate spam messages. However, we suspect that producing meaningful spam messages is hard without bloating the out-of-order keycode ratio in the message. Also, the old keycodes become invalid after two rounds of secret key rotations, preventing the bots from harvesting the valid keycodes indefinitely. Another possible attack is to add a small spam text to a bunch of unrelated human-typed characters that are blindly collected by a malicious bot. To prevent this attack, the email server could enforce a policy to confirm that at least a certain number of characters are typed in the "To:" header field with some timing bound, or the mail client can optionally highlight the portion of the text that is not human-attested.

Replaying attacks: A human spammer can attempt to compose a spam and send the same attested message to many recipients automatically. Since HumanSign attests to the typing of "To:" and "From:" email headers, the spammer may not send the same message to multiple locations without manually typing in the addresses. Mailing lists are non-trivial to defend against since HumanSign cannot tell spamming attempts from the legitimate usages. But it would still require typing in the original message for sending to each mailing list address.

Remote message composition: Humans can use Webmails to type in the email content over HTTP. Webmails typically use the HTTP POST method to send the email content in the request body. Since the HumanSign attestations are embedded in the email content (email headers and an attachment), they can be handled in the same way as in our Thunderbird extension. Attesting to the messages on remote desktop protocols [3, 29] require the HumanSign applications to invoke the attester on the local machine equipped with the trustworthy input device. While it would complicate the process, it is not impossible.

Mouse events: HumanSign does not support the tight binding between the messages and mouse events. We find it very difficult since the messages produced by mouse events (e.g., HTTP GET requests) have no inherent bearing with the events themselves. One strawman's approach would be to record and send all mouse events and the machine states that resulted in the message and to verify the resulting message by replaying the same application locally at the server, similar to [20]. However, due to high overheads, it would not be adequate for interactive applications. We leave this issue as our future work.

8. RELATED WORK

In this section, we compare our work with previous efforts. We explain previous spam detection techniques based on network-based fingerprinting and content analysis, other human detection approaches, and recent technologies that employ late-launch execution environments. Table 5 summarizes the features supported by HumanSign and other works.

Network-based fingerprinting: There have been many works that detect spamming bots by analyzing their network behavior. DNS blacklisting [40, 41, 45] has been useful for blocking spams from well-known spamming sources. However, its effectiveness has decreased over time since botnets spread the points of presence over tens to hundreds of thousands of infected zombie machines. Also, some slaves in the botnets are reported to send only one spam per day on average [37], making it challenging to identify the spamming bots by the volume.

SpamTracker [37, 38] learns the behavioral pattern of the bot spammers (such as the time of the day of the sent spam, its size, etc.) to distinguish legitimate emails from spams. Their follow-up, SNARE [21], is a distributed spam filtering engine that uses a classifier based on supervised learning. Although these techniques have greatly improved the accuracy of spamming bot detection in general, they depend on the statistical properties of the *current* spams that are subject to change over time. In contrast, we focus on a deterministic human detection method that can potentially achieve zero false positive regardless of the bot traffic pattern.

Signature analysis on message content: Existing spam filters typically rely on a combination of DNS blacklisting and content-based analysis to filter spams. For example, SpamAssassin [44] tests a number of network properties to derive spam signatures. It also uses the services that identify frequently-used spam URLs [48] and DNS blacklist such as SORBS [40]. While this approach is effective in detecting common spams, sophisticated botnets could bypass such filters by orchestrating short-lived spam campaigns and by iteratively using obfuscated URL shortening services. In addition, SORBS fails to protect against zero-day attack and its usage

is limited to emails. AutoRE [53] is a spam signature generation framework that detects spams from botnets as well as the botnet membership. It characterizes spamming botnets by spam content based on URL signatures that are obtained by machine learning. Other approaches [18, 55] map botnets using the spam email traces. The main drawback of these methods is that they do not guarantee deterministic prevention of botnets because they use machine learning techniques that are sometimes prone to false errors.

Human detection: Human detection has been a popular technique in network intrusion detection and prevention. Reverse Turing tests via manual human computation have been widely used in the Web through CAPTCHA [5, 6] for the past decade. Kill-Bots [24] also leverages CAPTCHA tests to authenticate human users at flooding attacks and allows connections through SYN cookies for a limited number of times. However, it has been shown that the CAPTCHA tests can be abused with cheap human labor [31] or some weak tests can be subverted by intelligent bots [17]. While we expect that CAPTCHA will be still useful in many areas, it would be impractical to ask for a CAPTCHA test each time a human user sends an email or an IM message. Park et al. [34] proposed identifying human Web traffic transparently by obfuscating Javascript mouse and key event handlers. They monitor if the Web page is viewed by humans by detecting mouse or keystroke events and by verifying if the Web request streams are coming from popular Web browsers. Gummadi et al. developed NAB [19] that certifies human-generated activity at the time of message creation. The NAB attester tags each request with an TPM-based signature if the attestation request is made within a small amount of time of legitimate keyboard or mouse activity. Although this helps identify the human traffic, NAB does not provide tight binding between the message and the input events, allowing a smart attacker to generate a fake attestation proof for an unrelated message.

HumanSign extends our previous work that uses a human attestation proof from the TPM-laden input devices [23]. In this work, we complete our framework by adopting the DRTM-based Flicker environment for secure execution of the attester, thus removing the need to introduce a TPM chip in devices themselves.

Late-launched secure environment: A number of projects have employed late launch to enhance the security of their systems. TBoot [46] uses late launch to securely boot a measured operating system. Bumpy [26] uses an enhanced version of Flicker that guarantees confidential passage of sensitive data from a client for authentication purposes. Unlike HumanSign, however, Bumpy requires the human user to type in special characters to alert a secure attestation request. TrustVisor [27] is a special-purpose thin hypervisor that safely executes security-sensitive code without late launch. TrustVisor could eliminate the heavy late launch operations per each HumanSign attestation but it requires a virtual machine for proper memory isolation. UTP [15] provides an accurate framework for Web authentication as it uses the isolated attester to generate the proof for human presence by introducing a test per request. However, their solution is limited to Web transactions and it would be inconvenient to manually type the characters for each Web transaction.

9. CONCLUSION

We have presented HumanSign, a novel human message attestation framework that allows the message recipient to prove the human existence at message composition. Unlike previous works, HumanSign tightly binds the input events to the message content, and leaves little room for attestation forgery by malicious bots. We have generalized the architecture so that the human attestation can be provided to any text-based network application service. Our experience with Email, SSH and IM applications shows that it is easy to write HumanSign extensions and to benefit from the framework. Our measurements on the current implementations show that HumanSign does add some noticeable delay due to heavy security hardware operations, but it is in the reasonable range for emails or password typing for SSH and IM applications.

10. REFERENCES

[1] OpenSSH - Keeping Your Communiques Secret. http://www.openssh.com/.

[2] Pidgin, the universal chat client. http://www.pidgin.im/.

[3] ReakVNC - VNC® remote control software. http://www.realvnc.com/.

[4] Advanced Micro Devices, Inc. AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual, 2005.

[5] L. Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. reCAPTCHA: Human-Based Character Recognition via Web Security Measures. In *Science*, volume 321, pages 1465–1468, 2008.

[6] L. V. Ahn, M. Blum, N. Hopper, and J. Langford. Captcha: Using hard AI problems for security. In *Proceedings of Eurocrypt*, 2003.

[7] Babcock, Charles. Open Source Code Contains Security Holes. *InformationWeek*, 2008. http://www.informationweek.com/news/security/showArticle.jhtml?articleID=205600229.

[8] E. Brickell, J. Camesnisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004.

[9] Cost of Spam is Flattening - Our 2009 Predictions. http://email-museum.com/2009/01/28/cost-of-spam-is-flattening-our-2009-predictions/.

[10] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[11] David Grawrock. Dynamics of a Trusted Platform, 2008.

[12] Dropbear SSH server and client. http://matt.ucc.asn.au/dropbear/dropbear.html.

[13] Facebook. http://www.facebook.com/.

[14] Facebook's anti-spam program catches activists. http://www.katu.com/news/tech/128125093.html.

[15] A. Filyanov, J. M. McCune, A.-R. Sadeghi, and M. Winandy. Uni-directional Trusted Path: Transaction Confirmation on Just One Device. In *Proceedings of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2011.

[16] Gmail Fires Back in the War on Spam. http://gadgetwise.blogs.nytimes.com/2012/04/11/gmail-fires-back-in-the-war-on-spam/?smid=tw-nytimestech&seid=auto/.

[17] D. Goodin. Google's reCAPTCHA busted by new attack. *The Register*, 2009. http://www.theregister.co.uk/2009/12/14/google_recaptcha_busted/.

[18] Google Postini Services. http://www.google.com/postini.

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

Implementation (NSDI), 2009.

[20] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable Virtual Machines. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[21] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser. Detecting spammers with SNARE: Spatio-temporal network-level automatic reputation engine. In *Proceedings of the USENIX Security Symposium*, 2009.

[22] Information Technology Laboratory, National Institute of Standards and Technology. National Software Reference Library. http://www.nsrl.nist.gov/Downloads.htm.

[23] M. Jamshed, W. Kim, and K. Park. Suppressing Bot Traffic with Accurate Human Attestations. In *1st ACM Asia-Pacific Workshop on Systems (APSys 2010)*, 2010.

[24] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[25] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of EuroSys*, 2008.

[26] J. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, 2009.

[27] J. M. McCune, Y. Li, N. Qu, Z. Zhou, and A. Datta. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[28] Messaging Anti-Abuse Working Group. Email Metrics Report. http://www.maawg.org/email_metrics_report/.

[29] Microsoft. Remote Desktop Connection. http://support.microsoft.com/kb/925876/en-us.

[30] S. Misra and V. Bhasvar. Relationships between selected software measures and latent bug-density. In *Proceedings of the International Conference on Computational Science and Its Applications (ICCSA)*, 2003.

[31] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. Voelker, and S. Savage. Re: CAPTCHAs – Understanding CAPTCHA-Solving from an Economic Context. In *Proceedings of the USENIX Security Symposium*, 2010.

[32] Mozilla. Mozilla Thunderbird. http://www.mozilla.org/en-US/thunderbird/.

[33] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, 1984.

[34] K. Park, V. S. Pai, K.-W. Lee, and S. Calo. Securing web service by automatic robot detection. In *Proceedings of the USENIX Annual Technical Conference*, 2007.

[35] Postfix Mail Transfer Agent. http://www.postfix.org/.

[36] Private conversation with Jonathan McCune, Research Systems Scientist for Cylab, Carnegie Mellon University & lead author of Flicker, 2011. http://sourceforge.net/mailarchive/forum.php?thread_name=CAEwYmTJ2B5rwCqNVcWjToD0nZrQh0jVKzL6Gw7xzNJE1VwvjFQ%40mail.gmail.com&forum_name=flickertcb-devel.

[37] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2006.

[38] A. Ramachandran, N. Feamster, and S. Vempala. Filtering spam with behavioral blacklisting. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

[39] T. H. Security. Microsoft wireless keyboard with 128-bit AES encryption, 2011. http://www.h-online.com/security/news/item/Microsoft-wireless-keyboard-with-128-bit-AES-encryption-1254 html.

[40] Spam and Open Relay Blocking System (SORBS). http://www.au.sorbs.net/.

[41] SpamCop Blocking List. http://spamcop.net/bl.shtml.

[42] Symantec. Messagelabs intelligence: Q2/june 2009. 2009. http://www.symanteccloud.com/mlireport/MLIReport_2009.06_June_FINAL.pdf.

[43] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[44] The Apache SpamAssassin Project. http://spamassassin.apache.org/.

[45] The Spamhaus Project. http://www.spamhaus.org/.

[46] Trusted Boot. http://tboot.sourceforge.net/.

[47] Twitter. http://www.twitter.com/.

[48] URL Blacklist. http://urlblacklist.com/.

[49] D. A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/.

[50] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel® CPU Cache Poisoning. March 2009.

[51] R. Wojtczuk and J. Rutkowska. Attacking Intel TXT® via SINIT code execution hijacking. 2011.

[52] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another Way to Circumvent Intel® Trusted Execution Technology: Tricking SENTER into misconfiguring VT-d via SINIT bug exploitation. December 2009.

[53] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2008.

[54] YouTube. http://www.youtube.com/.

[55] L. Zhuang, J. Dunagan, D. R. Simon, H. J. Wang, and J. D. Tygar. Characterizing botnets from email spam records. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET) Botnets, Spyware, Worms, and More*, 2008.