Suppressing Bot Traffic with Accurate Human Attestation

Muhammad Jamshed
Computer Science Department
University of Pittsburgh
210 S. Bouquet St
Pittsburgh, PA 15260
USA
ajamshed@cs.pitt.edu

Wonho Kim
Computer Science Department
Princeton University
35 Olden Street
Princeton, NJ 08544
USA
wonhokim@cs.princeton.edu

KyoungSoo Park
Electrical Engineering Department
KAIST
335 Gwahangno, Yuseong-gu
Daejeon, 305-701
Republic of Korea
kyoungsoo@ee.kaist.ac.kr

ABSTRACT

Human attestation is a promising technique to suppress unwanted bot traffic in the Internet. With a proof of human existence attached to the message, the receiving end can verify whether the content is actually drafted by humans. This technique can significantly reduce bot-generated abuse such as spamming, password cracking or even distributed denial-of-service (DDoS) attacks. Unfortunately, existing methods rely on the probabilistic characteristics of attestations and can be exploited by smart attackers.

In this paper, we propose deterministic human attestation based on trustworthy input devices. By placing the root of trust on the input device, we tightly bind the input events to the content for network delivery. Each input event is generated with a cryptographic hash that attests to human activity and the message consisting of such events gets a third-party verifiable digital signature that is carried to the remote application. For this, we augment the input device with a trusted platform module (TPM) chip and a small attester running inside the device. We focus on trustworthy keyboards here but we plan to extend the framework to other input devices.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Reliability

Keywords

Human Attestation, Bot Traffic Suppression, Trusted Computing, Networked System Security

APSys 2010, August 30, 2010, New Delhi, India.

1. INTRODUCTION

Unsolicited messages have widely permeated into our daily Internet life. Despite decade-long efforts, the volumes of email, blog, and instant messaging (IM) spam are continually on the rise, and millions of infected botnet machines aggravate the situation. 87.7% of the total emails, that is, 107 billion messages being sent globally on a daily basis, are estimated to be spam in 2009 [7]. University of Pittsburgh received 51 million spam emails in November 2009 alone while only 17 million emails were delivered as legitimate to the users [10]. While best practices such as content-based filtering [2, 19, 21], DNS blacklisting [12] and network-based fingerprinting [4, 11] greatly reduce the spam delivery, they often create false positives - legitimate emails classified as spam, making the Internet message delivery less reliable.

Human attestation is a promising technique that can potentially exterminate unwanted bot traffic. By carrying a non-forgeable proof of human existence with the message, the receiving end can reliably determine the identity of the traffic source and adjust her filtering policy to better accommodate human traffic. Existing methods typically infer the human activity from key clicks or mouse events and use TPM-generated signatures as human attestation. This approach is shown to be effective in reducing spam, DDoS attacks, click frauds, and so on [3, 8]. However, one serious problem is that they depend on the probabilistic characteristic of the attestation. For example, Not-a-Bot [3] allows any bot to get a human attestation for its own content within an allowed timing window after key or mouse clicks are generated. Though the rate of illegitimate attestations is bounded by that of human activity, smart attackers can deterministically bypass any spam filter based on human attestation. That is, bots could generate sophisticated spam messages for targeted users or act like humans when they access Internet banking or E-commerce sites if they adopt existing methods as human identification.

The key question this paper poses is how strongly one can bind human activity to the content that is sent over the network. We argue that the content should carry a signature that can be independently verified that its content was actually created by a human. We build a framework in which each element in the content is confirmed to be human-generated. In our framework, only the part of the content that is made of keystrokes gets a human attestation and the rest of it is tagged as unattested before delivering to the remote application. Based on the accurate human content attestation, the remote application can make an informed

scan code (ξ)	timestamp (τ)	proof (π)
1-3 bytes	8 bytes	3 bytes

Table 1: Secure Keycode Format

decision whether to accept or block the content. This scheme prevents malicious bots from getting attestations for automated content and limits the input for content to human typing.

The root of trust in our framework is the input device. The input device generates each input event along with its own "proof". The input event proof is simply a cryptographic hash of the input value and its timestamp, and can be verified only by the input device that generates it. When the application needs human attestation on the content, it asks the input device to generate a signature of the content by having it verify the proofs. For the digital signature, we embed a TPM chip in the input device and benefit from its built-in attestation functionality. Our framework does not assume any trusted computing base (TCB) from the operating system or applications, but it still eliminates the bot abuse except random content sending. We implement SMTP and SSH clients and servers that use our framework and show that it is easy to support accurate human attestation in network message delivery. We focus on the keyboard as the input device in this paper, and we plan to extend the framework to other input devices.

2. ADVERSARY MODEL

We assume that the only trusted component in the host machine is the input device. We assume that the input device is equipped with a TPM and a small attester daemon running inside the device. The TPM is used solely for publishing independently-verifiable digital signatures. The attester monitors the keystroke interrupts and generates key events along with the proofs and timestamps. These proofs are later verified again by the attester, and the built-in TPM signs the message content from the application. We assume that the attester daemon and the TPM are trusted and cannot be compromised while they can be physically damaged.

The rest of the system (e.g., the OS and applications) can be compromised by adversaries. Bots such as spyware can be installed in the host machine and can monitor and intercept the key events from the input device. They can attempt to assemble a message out of the key events but we assume bots cannot generate valid proofs for random key events by themselves. The valid keystroke events are generated only by physically pressing the input device. We do not defend against the attacks that disrupt the network access or proper functioning of innocent applications.

3. ATTESTATION FRAMEWORK

Our attestation framework builds on the assumption that input devices produce the key events that can be verified to be typed by humans. Only the content made of such input events can get the accurate attestation from the device and can be verified by remote applications. One can think of securing inputs with a system TPM in the trusted virtual machine hypervisor, but such a scheme may not be an ideal solution on client machines that do not use virtual machines.

3.1 Trustworthy Keyboard

We define an extended key event, which we call secure keycode, and show its format in Table 1. Secure keycode consists of a scan code (ξ), a timestamp (τ), and a proof (π). τ records the secure keycode generation time in the millisecond granularity (4 bytes for a date and 4 bytes for milliseconds past on that day). Timestamps prevent replaying of old keycode by malicious bots and give hints to the remote applications about when each character in the content is typed. π is a cryptographic hash of the scan code and the timestamp. We use the last three bytes of HMAC(K_s, $\langle\ \xi\ |\ \tau\ \rangle$), where K_s is a secret key that never leaks out of the keyboard. Since we use constantly-changing timestamps in hash calculation, the same characters would produce different hash values. The proof also limits the probability of generating valid keycode by random guess to 2^{-24}. We note that the secure keycode brings a 4 to 12 times blowup in the key event size, but the secure keycode is used only in the local host between the input device and the application. Since the key generation rate is bounded by human typing, we believe the size overhead is bearable given the trend of increasing CPU power and ample memory of modern computers. We call a keyboard trustworthy if it generates secure keycode and attests to the human-generated content consisting of secure keycode.

The attester in the keyboard is responsible for generating the secure keycode and human attestation. It is assumed to be running on a low-powered processor with small nonvolatile storage space to save the secret keys. The attester monitors keystroke interrupts and wraps the generated scan code in the secure keycode. It creates the secret key for HMAC, and rotates the key every n days (e.g., n=2) while keeping the previous secret key. Secure keycode older than 2n days are assumed to be invalid for the purpose of human attestation.

3.2 Human Content Attestation

The attestation to the human-generated content requires two steps: secure key code validation and TPM-based signature generation. The user application provides the content for attestation in a sequence of secure code to the attester. The attester verifies if the content consists of valid secure keycode by rehashing the scan code and the timestamp. If the verification succeeds, the attester uses the built-in TPM to sign on the content. If the timestamp is too old or the proof does not match the last three bytes of HMAC value with either current or old K_s (depending on the key generation timestamp), the verification process fails and the application client is notified of a failure.

After secure keycode validation, the attester extracts only the scan code and the timestamp from secure keycode for human content attestation. We call the array of (scan code, base time offset) the CodeTimeMap of the content. To reduce the size of the original timestamp, we use a base time and its offset to represent the timestamp. The base time is the smallest timestamp in the content keycode and the base time offset is the difference between the base time and its original timestamp. The granularity of the base time offset is 100 ms. The attester rearranges the offsets so that no two same characters belong to the same 100 ms time bin. For example, if a user types a word, 'cool', and in the rare case that the first 'o' and the second 'o' belong to the same 100 ms bin, the base time offset of the second 'o' is adjusted to

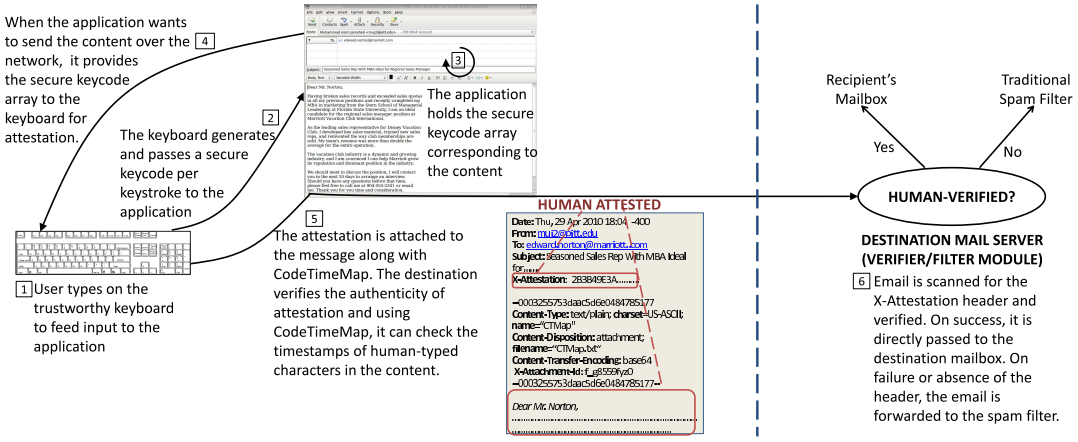

Figure 1: Human Content Attestation Process

the next 100 ms bin. This prevents the bots from reusing the same character. The content CodeTimeMap is signed by the TPM and the resulting signature attests that the content is typed by a human using the trustworthy keyboard that hosts the TPM. The return value for successful attestation is (base time, base time offset size, CodeTimeMap, TPM signature, signature timestamp), where the base time offset size is the number of bytes required to represent the largest base offset in the CodeTimeMap. Two to three bytes should be enough for the offset in practice (1.8 hours to 19.4 days).

Fig. 1 illustrates how a typical network application attests to the human-typed content. The application receives the secure keycode from the trustworthy keyboard as a human user types the content. When the application needs to send the message over the network, it asks the keyboard for attestation by providing the array of secure keycode for the message. When the application obtains the digital signature of the content, it sends the signature and the content CodeTimeMap along with the original content. In case the content for transfer includes invalid secure keycode (e.g., copy & paste from other documents, characters typed long time ago, automatically-generated content by application, etc.), the application requests a signature only for the portion of valid secure code. It tags other part as unattested, and lets the remote verifier decide whether to accept it or not using her own policy. Another option is to get the full content attestation by engaging the human user. If the attester fails to verify some keycode in the content, it can generate a CAPTCHA test to the human user. For example, the attester can create a CAPTCHA image consisting of n characters chosen at random. The attester returns this image with a TPM signature that signs (CodeTimeMap | n-character phrase) along with other information. The application shows the CAPTCHA image to the user, and if the user's answer matches the phrase, the application appends the phrase to the CodeTimeMap. Note that the CAPTCHA test is a configuration option, and a human user chooses this path only when she wants full guarantee to pass human verification for the entire content.

Malicious bots can attempt to mix and match the secure keycode, but the chance of producing a meaningful message should be negligible. Because the keycode becomes invalid after two rounds of HMAC secret key rotation, the bots cannot indefinitely wait for any missing characters they want. Our framework does not prevent replaying the entire message that a human user creates, but we doubt if spamming bots have an incentive to replay a regular human-typed email. In case of interactive applications such as SSH, the receiving end can enforce a policy to accept only the commands whose last-typed character is within a threshold period (e.g., 10 seconds).

Function	Latency
TPM_Extend	14 ms
TPM_PCR_Reset	11 ms
TPM_Quote ({1024,2048}-bit RSA)	{145, 746} ms
TPM_LoadAIK_Key (one-Time Operation)	2.43 s

Table 2: Function Latency (STMicroelectronics TPM)

3.3 Remote Verification

The remote verification process is straightforward. The verifier for the content recipient first checks whether the key (known as attestation identity key (AIK)) used to sign the CodeTimeMap actually comes from a valid TPM. This can be done by contacting a trusted third-party Certifying Authority (CA) that vouches for the integrity of the TPM via a certificate. Next, it verifies the signature itself with the public AIK. After signature verification, the remote verifier can apply her own policy based on the keycode timing information. For example, if an email is created beyond an acceptable time interval, the verifier can ignore human attestation, and run a regular spam filter on the content. Or if many words in the content are created in random order, the verifier may suspect the content was mixed and matched by a bot. We believe there is large space to explore in the filtering policy based on the human attestation signature and the CodeTimeMap.

4. EXPERIMENTAL EVALUATION

For a proof of concept, we have implemented the attester as a user-level process (856 lines of C code) and have applications get the secure input and content attestation via a socket interface. We have updated the USB keyboard driver to expose the key events through the `sysfs` file system in-

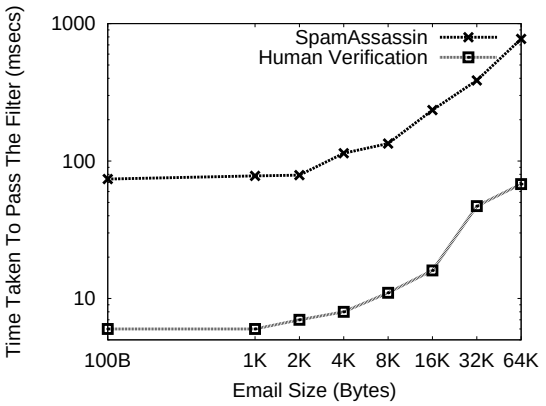

Figure 2: Time to verify human-attested emails. On 2.50 GHz Intel Pentium single-core machine with 6 GB RAM.

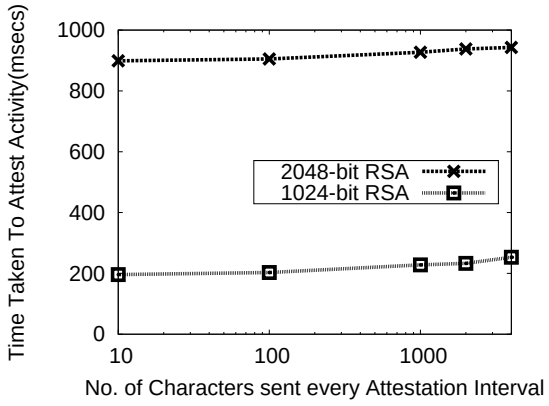

Figure 3: Time taken to attest a group of characters on a Dropbear ssh session. On a 1.20 GHz Intel Pentium single core machine with 3GB RAM.

terface. We use Linux 2.6.21 and an on-board TPM from STMicroelectronics that ships with Dell Optiplex 780, an 2.70 GHz Intel dual-core machine with 3 GB memory.

We first measure the TPM function latency and show the results in Table 2. TPM_Extend loads the data to be signed to Platform Configuration Register (PCR) and TPM_PCR_Reset flushes the data. TPM_Quote signs the PCR value using the private AIK loaded by TPM_LoadAIK_Key. TPM_Quote (2048-bit RSA) takes 746 ms, which is the slowest operation in getting the attestation. A typical 2048-bit TPM_Quote size is 580 bytes long consisting of an AIK (256B), a TPM-constructed data blob containing essential PCR attributes (48B), the RSA signature (256B) of that data blob and the SHA1 hash (20B) of the actual data. The current TPM specification do not allow AIK certification for key sizes smaller than 2048 bits. This is due to the concern that smaller RSA keys can be broken in the future.

4.1 Human-typed Email Verification

We have implemented a Mozilla Thunderbird 2.0.0.23 extension that attaches Base64-encoded human attestation signature to every human-typed email. We also implement a verification filter that works with Postfix 2.5.5-1 [9]. The filter scans the human attestation signature attached to emails and flags them as human-generated or unattested.

Figure 3 shows our Postfix filter performance. All emails in the graph take less than 70 ms for human verification. We compare it with the filtering latency of SpamAssassin 3.2.5, a popular content-based spam filter, with the default configuration. Our verification performance is comparable to that of SpamAssassin for small content, but as the size of the email increases, the performance gap widens. Our filter is 12 times faster than SpamAssassin at 64 KB email.

4.2 SSH Command Attestation

We have added human attestation to the Dropbear 0.52 [1] SSH client/server suite so that each ssh command carries an attestation signature and the server verifies the signature before launching the command. That is, for every enter key, we send the signature to the server for human verification.

Figure 3 shows the attestation signature generation time as the command size increases. We downgraded the CPU clock frequency of Dell Optiplex 780 to 1.20 GHz to simulate a low-power processor that the attester is likely to be using. We see relatively high latency for 2048-bit RSA signature generation, reflecting the slow operation of TPM_Quote. In contrast, 1024-bit RSA signatures show much smaller latency, which makes it practical for interactive environments. Given that 1024-bit RSA is widely used in the SSL protocol, we believe 1024-bit RSA can be a reasonable choice for the current day use if the the TPM specification allows it. One nice trend is that the response time stays more or less the same regardless of the command size. We hope that the performance of TPM_Quote improves as the demand for faster TPM grows.

5. RELATED WORK

Not-a-Bot (NAB) [3] motivated our work. NAB proposes a simple human attestation framework that can be used to reduce spam, click frauds, DDoS attacks, etc. They infer the human existence from input device events, and allow human content attestation with a TPM within an acceptable time window (they use one second) from the last key click. However, their scheme can be abused by smart attackers, which is noted in their work. We share the same goal of suppressing the bot traffic, but provide a stronger attestation framework that eliminates the bot abuse by tightly binding the key events to the content. Our earlier work explored bot detection in the Web environment by having obfuscated Javascript code catch the input events of the users [8]. It has shown to be effective to block most Web abuse on CoDeeN content distribution networks [18], though it does not prevent sophisticated bots that generate software interrupt-based input events.

Sailer *et al.* developed the first framework that employs a TPM for static software stack-based remote attestation [13]. McCune *et al.* built a *trusted path* between sandboxed software and the remote application by using dynamic root of trust [6] and reduced the trusted computing base to a single secure hypervisor having a small code footprint. Our framework adds human attestation and can further improve the level of trust in human-interactive applications as it eliminates the dependence of a trusted software stack from the scheme.

Modern spam filters [2, 15] typically rely on content-based filtering coupled with IP-based [14] and URL-based [16] black-

listings. There has been significant amount of research on devising accurate detection techniques to defend against spammers primarily originating from botnets [4, 5, 12, 17, 20]. They usually add a set of features to spam-detecting classifiers that capture the behavior of botnets. While the spam detection accuracy has improved greatly over time, the inherent probabilistic nature of learning-based filtering often creates false positives, which makes the legitimate email delivery unreliable. Human attestation can potentially bring the false positive rate to zero.

6. CONCLUSION

We have developed an accurate human attestation framework that draws the trust from the input device. Without any dependency on the software stack integrity beyond trustworthy input devices, remote applications can verify if the content is actually created by humans. We have also shown that it is straightforward to employ our scheme into existing applications that deal with human-typed content.

The immediate next step would require adding other input devices such as mouse. We are working on capturing the sequence of mouse events and connecting them to the content creation context. Remote verifier policy is another interesting area to explore. With the timestamp of each character, one can calculate the fraction of the words created from left-to-right and use the information to block any content that is mixed and matched by sophisticated bots. User privacy can be a concern, and minimizing the exposure of the timing information while maintaining a good filtering quality should be further studied.

7. ACKNOWLEDGEMENT

We thank Daniel Mosse and Sunghwan Ihm for valuable discussion. We also thank anonymous APSys reviewers for their insightful comments. This research was funded by KAIST award G04100004.

8. REFERENCES

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

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

[3] 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 USENIX NSDI*, 2009.

[4] 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.

[5] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying spamming botnets using botlab. In *Proceedings of USENIX NSDI*, 2009.

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

[7] MessageLabs Intelligence: 2009 Annual Security Report. `http://www.messagelabs.com/mlireport/2009MLIAnnualReport_Final_PrintResolution.pdf`.

[8] 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.

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

[10] Private conversation with Ben Carter, IT staff member for the Univerity of Pittsburgh, 2009.

[11] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proceedings of ACM SIGCOMM*, 2006.

[12] 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.

[13] R. Sailer, X. Zhang, T. Jaeger, and L. v. Doom. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the USENIX Security Symposium*, 2004.

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

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

[16] URL Blacklist. `http://urlblacklist.com/`.

[17] S. Venkataraman, S. Sen, O. Spatscheck, P. Haffner, and D. Song. Exploiting network structure for proactive spam mitigation. In *Proceedings of the USENIX Security Symposium*, 2007.

[18] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[19] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. In *Proceedings of ACM SIGCOMM*, 2008.

[20] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. BotGraph: Large scale spamming botnet detection. In *Proceedings of USENIX NSDI*, 2009.

[21] 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.