Help Me Help You:
Using Trustworthy Host-Based Information in the Network

Bryan Parno, Zongwei Zhou, Adrian Perrig

November 18, 2009

CMU-CyLab-09-016

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Help Me Help You: Using Trustworthy Host-Based Information in the Network

Bryan Parno, Zongwei Zhou, Adrian Perrig
Carnegie Mellon University

***Abstract*—As hardware support for improved endhost security becomes ubiquitous, it is important to consider how network security and performance can benefit from these improvements. If endhosts (or at least portions of each endhost) can be trusted, then network infrastructure no longer needs to arduously and imprecisely reconstruct data already known by the endhosts. Through the design of a general-purpose architecture we call Assayer, we explore the issues in providing trusted host-based data, including the balance between useful information and user privacy, and the tradeoffs between security and efficiency. We also evaluate the usefulness of such information in three case studies.**

To gain insight into the performance we could expect from such a system, we implement and evaluate a basic Assayer prototype. Our prototype requires fewer than 1,000 lines of code on the endhost. Endhosts can annotate their outbound traffic in a few microseconds, and these annotations can be checked efficiently; even packet-level annotations on a gigabit link can be checked with a loss in throughput of only 3.7-18.3%.

I. Introduction

Why is it difficult to improve network security? One possible culprit is the fact that network elements cannot trust information provided by the endhosts. Indeed, network elements often waste significant resources painstakingly reconstructing information that *endhosts already know*. For example, a network-level Denial-of-Service (DoS) filter must keep track of how many packets each host recently sent, in order to throttle excessive sending. Researchers have developed many sophisticated algorithms to trade accuracy for reduced storage overhead [10,34,36], but they all amount to *approximating* information that can be *precisely and cheaply tracked* by the sender! In other words, the filter's task could be greatly simplified if each sender could be trusted to include its current outbound bandwidth usage in each packet it sent.

Of course, the question remains: Is it possible to trust endhosts? We observe that the current widespread deployment of commodity computers equipped with hardware-based security enhancements may in fact allow the network to trust *some* host-based information. In recent years, over 200 million computers equipped with a Trusted Platform Module have been deployed [16], and most smartphones include processors and software with far more security features than a standard desktop [2, 31]. As such security features become ubiquitous, it is natural to ask if we can leverage them to improve network security and efficiency.

As an initial exploration of how endhost hardware security features can be used to improve the network, we have designed a general architecture named Assayer. While Assayer may not represent the optimal way to convey this information, we see it as a valuable first step to highlight the various issues involved. For example, can we provide useful host-based information while also protecting user privacy? Which cryptographic primitives are needed to verify this information in a secure and efficient manner? Our initial findings suggest that improved endhost security can improve the security and efficiency of the network, while simultaneously reducing the complexity of in-network elements.

In the Assayer architecture, senders employ secure hardware to convince an off-path *verifier* that they have installed a small code module that maintains network-relevant information. A small protection layer enforces mutual isolation between the code module and the rest of the sender's software, ensuring both security and privacy. Once authorized by a verifier, the code module can insert cryptographically-secured information into outbound traffic. This information is checked and acted on by in-path *filters*.

To evaluate the usefulness of trustworthy host-based information, we consider the application of Assayer to three case studies: Spam Identification, Distributed Denial-of-Service (DDoS) Mitigation, and Super-Spreader Worm Detection.

To better understand the performance implications of conveying host-based information to the network, we implement a full Assayer prototype, including multiple protocol implementations. The size of the protection layer on the client that protects code modules from the endhost (and vice versa) is minuscule, and our verifier prototype can sustain 3300 client verifications per second. Generating and verifying annotations can be done efficiently using our performance-optimized scheme.

Contributions: **(1)** An exploration of how we can design network security mechanisms to leverage endhost-based knowledge and state, **(2)** The design of an architecture for providing such information securely and efficiently, and **(3)** An implementation and evaluation of the architecture.

II. Secure Hardware Background

The prevalence of malware on endhosts makes the use of hardware-based security crucial. Otherwise, there is little to distinguish legitimate software from malicious software pretending to be legitimate. Such hardware could take the form of a security coprocessor, such as the IBM 4758 [30] or a USB-attached smart card [17].

However, in our prototype implementation we chose to use a Trusted Platform Module (TPM) [33] due in large part to their ubiquity: over 200 million TPMs have already been deployed [16]. Below, we highlight the properties we require from our secure hardware, and summarize how the TPM provides those properties. We emphasize, however, that many other types of secure hardware offer similar properties.

Measurement. When a TPM-equipped platform first boots, platform hardware takes a measurement (a SHA-1 hash) of the BIOS and records the measurement in one of the TPM's Platform Configuration Register (PCR). The BIOS is then responsible for measuring the next piece of software (e.g., the bootloader) and any associated data files. The BIOS records the measurement in a PCR before executing the software. As long as each subsequent piece of software performs these steps (measure, record, execute), the TPM

serves as an accurate repository of measurements of code executed on the platform [29]. While initial work assumed that the TPM would be used to attest to an entire software stack, subsequent work has demonstrated that a *late launch* (new functionality added to CPUs by AMD and Intel) can be used to attest to a small piece of security-sensitive software [18].

Attestation. To securely convey measurements to an external verifier, the TPM creates attestations. Given a verifier-supplied nonce, the TPM will use a private key that is never accessible outside the TPM to generate a TPM_Quote by computing a digital signature over the nonce and the contents of the PCRs. The nonce assures the verifier that the attestation is fresh and not from a previous boot cycle.

To ensure the attestation comes from a real hardware TPM (rather than a software emulation), the TPM comes with an endorsement keypair $\{K_{EK}, K_{EK}^{-1}\}$ and an endorsement certificate for the public key from the platform's manufacturer declaring that K_{EK} does indeed belong to a real TPM.

User Privacy. To preserve the user's privacy, the TPM does not sign attestations with K_{EK}^{-1}. Instead, the TPM generates a public key pair $\{K_{AIK}, K_{AIK}^{-1}\}$ called an Attestation Identity Key (AIK) pair. Using K_{EK}^{-1} and the endorsement certificate, the TPM convinces a Privacy Certificate Authority (Privacy CA) that K_{AIK} belongs to a legitimate TPM and thus obtains a certificate for the public AIK. The TPM then uses K_{AIK}^{-1} to sign attestations. By using multiple AIKs, the client can preserve her privacy, as long as she trusts the Privacy CA not to collude with the services she visits. The latest TPM specification [33] includes a provision for Direct Anonymous Attestation [6], which uses group signatures to generate attestations, but as of yet, we are not aware of any TPMs that implement this functionality.

Sealed Storage. The TPM can bind data to a particular platform configuration using a technique called sealed storage. Essentially, software can request that the TPM seal (encrypt) a blob of binary data to a particular set of PCR values. The TPM will only unseal (decrypt) the data if the current values in the PCRs match the values specified during the seal operation. Thus, if the platform boots different software, the new software will be unable to access previously sealed data.

III. Problem Definition

A. Architectural Goals

We aim to design an architecture to allow endhosts to share information with the network in a trustworthy and efficient manner. This requires the following key properties:

Annotation Integrity. Malicious endhosts or network elements should be unable to alter or forge the data contained in message annotations.

Stateless In-Network Processing. To ensure the scalability of network elements that rely on endhost information, we seek to avoid keeping per-host or per-flow state on these devices. If per-flow state becomes feasible, we can use it to cache authentication information carried in packets.

Privacy Preservation. We aim to leak no more user information than is already leaked in present systems. However, some applications may require small losses of user privacy. For example, annotating outbound emails with the average length of emails the user sent in the last 24 hours leaks a small amount of personal information, but it can significantly decrease the probability a legitimate sender's email is marked as spam [15]. We can provide additional privacy by only specifying this information at a coarse granularity, e.g., "short", "medium", and "long". Further research will be necessary to determine whether people accept this tradeoff.

Incremental Deployability. While we believe that trustworthy endhost information would be useful in future networks, we strive for a system that can bring immediate benefit to those who deploy it.

Efficiency. To be adopted, the architecture must not unduly degrade client-server network performance.

B. Assumptions

Since we assume that our trusted software and hardware components behave correctly, we aim to minimize the size and complexity of our trusted components, since software vulnerabilities are correlated with code size [25], and smaller code is more amenable to formal analysis. We assume that clients can perform hardware-based attestations. In this work, we focus on TCG-based attestations, but other types of secure hardware are also viable (see Section II). Finally, we make the assumption that secure-hardware-based protections can only be violated with local hardware attacks. We assume remote attackers cannot induce users to perform physical attacks on their own hardware.

IV. The Assayer Architecture

With Assayer, we hope to explore the intriguing possibilities offered by the advent of improved hardware security in endhosts. If endhosts can be trusted, how can we simplify and improve the network? What techniques are needed to extend host-based hardware assurance into the network? Can trust be verified without significantly reducing network performance? We examine these issues and more below.

Initially, we focus on the qualities needed to build a generic architecture for conveying host-based information to the network, and hence our discussion is necessarily quite general. However, we explore application-specific details, including deployment incentives in Section VI.

A. Overview

Suppose a mail server wants to improve the accuracy of its spam identification using host-based information. For example, a recent study indicates that the average and standard deviation of the size of emails sent in the last 24 hours are two of the best indicators of whether any given email is spam [15]. These statistics are easy for an endhost to collect, but hard for any single mail recipient to obtain.

However, the mail server is faced with the question: how can it decide whether host-provided information is

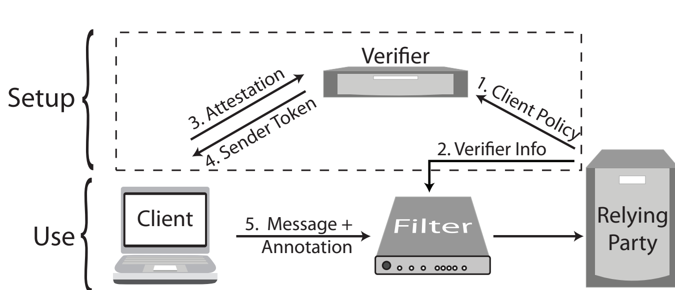

Figure 1. **System Components.** *The relying party (e.g., a mail server or an ISP) delegates the task of inspecting clients to one or more verifiers. It also configures one or more filters with information about the verifiers. Every T days, the client convinces a verifier via an attestation that its network measurement modules satisfy the relying party's policy. The verifier issues a Sender Token that remains valid for the next T days. The client can use the Sender Token to annotate its outbound messages (e.g., an annotation for each email, flow, or packet). The filter verifies the client's annotation and acts on the information in the annotation. For example, the filter might drop the message or forward it at a higher priority to the relying party.*

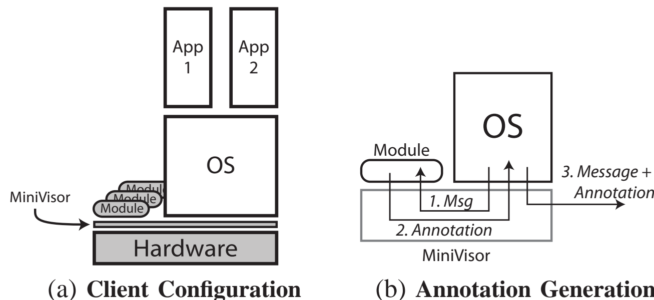

Figure 2. **Client Operations. (a)** *The client attests to the presence of a protective layer (MiniVisor) that isolates modules from untrusted code (and vice versa) and from each other. This allows the client to attest to multiple relying parties without a reboot.* **(b)** *Untrusted code submits messages (e.g., email or packets) to the modules to obtain authenticated annotations.*

trustworthy? Naively, the mail server might ask each client to include a hardware-based attestation (see Section II) of its information in every email. The mail server's spam filter could verify the attestation and then incorporate the host-provided information into its classification algorithm. Any legacy traffic arriving without an attestation could simply be processed by the existing algorithms. Unfortunately, checking attestations is time-consuming and requires interaction with the client. Even if this were feasible for an email filter, it would be unacceptable for other applications, such as DDoS mitigation, which require per-packet checks at line rates.

Thus, the question becomes: how can we make the average case fast and non-interactive? The natural approach is to cryptographically extend the trust established during a single hardware-based attestation over multiple outbound messages. Thus, the cost of the initial verification is amortized over subsequent messages.

As a result, the Assayer architecture employs two distinct phases: an infrequent setup phase in which the *relying party* (e.g., the mail server) establishes trust in the client, and the more frequent usage phase in which the client generates authenticated annotations on outbound messages (Figure 1).

The relying party delegates the task of inspecting clients to one or more off-path *verifier* machines. Every T days, the client convinces a verifier that it has securely installed a trustworthy code *module* that will keep track of network-relevant information (Figure 2), such as the number of emails recently sent, or the amount of bandwidth recently used. Section IV-B1 considers how we can secure this information while still allowing the user to employ a commodity OS and preserving user privacy. Having established the trustworthiness of the client, the verifier issues a limited-duration *Sender Token* that is bound to the client's code module.

During the usage phase, the client submits outbound messages to its code module, which uses the Sender Token to authenticate the message annotations it generates. These annotations are then checked by one or more fast-path *filter* middleboxes, which verify the annotations and react accordingly. For instance, a relying party trying to identify spam might feed the authenticated information from the filter into its existing spam classification algorithms. Alternatively, a web service might contract with its ISP to deploy filters on its ingress links to mitigate DDoS attacks by prioritizing legitimate traffic. If the traffic does not contain annotations, then the filter treats it as legacy traffic (e.g., DDoS filters give annotated traffic priority over legacy traffic).

B. Assayer Components

We present the design decisions for each of Assayer's components, saving the protocol details for Section IV-C.

1) Clients: In this section, we consider the generic requirements for allowing clients to transmit trustworthy information to the network. We explore application-specific functionality and client deployment incentives in our case studies (Section VI).

Client Architecture. At a high-level, we aim to collect trustworthy data on the client, despite the presence of (potentially compromised) commodity software. To accomplish this, a client who wishes to obtain the benefits of Assayer can install a protective layer that isolates the application-specific *client modules* from the rest of the client's software (Figure 2(a)). These client modules could be simple counters (e.g., tracking the number or size of emails sent) or more complex algorithms, such as Bayesian spam filters. The protective layer preserves the secrecy and integrity of the module's state, as well as its execution integrity. It also protects the client's other software from a malicious or malfunctioning module. Untrusted code can submit outbound traffic to a module in order to obtain an authenticated annotation (see Figure 2(b)).

How can a client convince the verifier that it has installed an appropriate protective layer and client module? With Assayer, the client can employ hardware-based *attestation*

to prove exactly that. When the verifier returns a Sender Token, the protective layer invokes *sealed storage* to bind the Sender Token to the attested software state. This can be combined with the TPM's monotonic counters to prevent state-replay attacks. Thus, any change in the protective layer or client module will make the Sender Token inaccessible and require a new attestation.

Designing the protective layer raises several questions. How much functionality should it provide? Should the client modules be able to learn about the client's other software? Should the protective layer control the entire platform, or only enough to provide basic isolation?

With Assayer, we chose to implement the protective layer as a minimal hypervisor (dubbed MiniVisor) that contains only the functionality needed to protect the client modules from the client's other software (and vice versa). This approach sacrifices visibility into the client's software state (e.g., a client module for web browsing cannot determine which web browser the client is using), but protects user privacy from overly inquisitive client modules. Using MiniVisor makes the Trusted Computing Base tiny and reduces the performance impact on the client's other software.

In developing MiniVisor, we reject the use of a full-fledged OS [29] or even a VMM [12] as protective layers. Such an approach would leak an excessive amount of information about the client's platform, and assuring the security of these larger code bases would be difficult.

Initially, it is tempting to give MiniVisor full control over the client's network card, in order to ensure that all traffic can be examined by the client modules. However, this approach would significantly increase MiniVisor's complexity, and it would be difficult to ensure full control over *all* outbound traffic on *all* interfaces. Instead, we advocate the use of application-specific incentives to convince the commodity software to submit outbound traffic to the client modules. Since the resulting annotations are cryptographically protected for network transmission, these protections will also suffice while the annotations are handled by the untrusted software. In Section VI, we explore whether sufficient client incentives exist for a variety of applications.

Client Modules. As mentioned above (and explored in more detail in Section VI), we expect Assayer to support a wide variety of client modules. While we have initially focused on relatively simple modules, e.g., modules to track the size and frequency of emails sent, they could potentially expand to perform more complex operations (e.g., Bayesian spam filtering) of interest to relying parties.

For *global vantage* applications, such as recording statistics on email volume, we expect a single client module would be standardized and accepted by multiple relying parties. MiniVisor's configuration would ensure that the client only runs a single copy of the module to prevent state-splitting attacks. The module could be obtained and installed along with the client's application software (e.g., email client). The application (or a plugin to the application) would then know to submit outbound emails to the module via MiniVisor.

For *single relying party* applications, such as DDoS prevention, a relying party may only care about tracking statistics specific to itself. In this case, the relying party may itself provide an appropriate client module, for example, when the client first contacts the verifier. This model highlights the importance of preserving user privacy, as we emphasized above. The client's OS can submit packets to this module if and only if they are destined to the particular relying party that supplied the client module.

2) Verifiers: Verifiers are responsible for checking that clients have installed a suitable version of MiniVisor and client module and for issuing Sender Tokens. The exact deployment of verifiers is application and relying-party specific. We envision three primary deployment strategies. First, a relying party could deploy its own verifiers within its domain. Second, a trusted third party, such as VeriSign or Akamai could offer a verification service to many relying parties. Finally, a coalition of associated relying parties, such as a group of ISPs, might create a federation of verifiers, such that each relying party deploys a verifier and trusts the verifiers deployed by the other relying parties.

In the last two of these scenarios, the verifiers operate outside of the relying party's direct administrative domain. Even in the first scenario, the relying party may worry about the security of its verifier. To assuage these concerns, the relying party periodically requests a hardware-based attestation from the verifier. Assuming the attestation is correct, the relying party establishes the key material necessary to create an authentic channel between the verifiers and the filters. On the verifier, this key material is bound (using sealed storage) to the correct verifier software configuration.

The use of multiple verifiers, as well as the fact that clients only renew their client tokens infrequently (every T days), makes the verifiers unlikely targets for Denial-of-Service (DoS) attacks, since a DoS attacker would need to flood many verifiers over an extended time (e.g., a week or a month) to prevent clients from obtaining tokens.

Any distributed and well-provisioned set of servers could enable clients to locate the verifiers for a given relying party. While a content distribution network is a viable choice, we propose a simpler, DNS-based approach to ease adoption. Initially, each domain can configure a well-known subdomain to point to the appropriate verifiers. For example, the DNS records for company.com would include a pointer to a verifier domain name, e.g., verifier.company.com. That domain name would then resolve to a distributed set of IP addresses representing the server's verifier machines.

3) Filters: Filters are middleboxes deployed on behalf of the relying party to act on the annotations provided by the client. For instance, a spam filter might give a lower spam score to an email from a sender who has generated very little email recently. These filters must be able to verify client annotations efficiently to prevent the filters themselves from

becoming bottlenecks. In addition, to prevent an attacker from reusing old annotations, each filter must only accept a given annotation once.

Filter deployment will be dictated by the application (discussed in more detail in Section VI), as well as by the relying party's needs and business relationships. For example, a mail server might simply deploy a single filter as part of an existing spam classification tool chain, whereas a web hosting company may contract with its ISP to deploy DDoS filters at the ISP's ingress links.

To enable filters to perform duplicate detection, the client modules include a unique nonce as part of the authenticated information in each annotation. Filters can insert these unique values into a rotating Bloom Filter [4] to avoid duplication. Section V discusses the effectiveness of this approach against replay attacks.

C. Protocol Details

Below, we enumerate desirable properties for the authorization scheme used to delegate verifying power to verifiers, as well as that used by clients to annotate their outbound traffic. We then describe a scheme based on asymmetric cryptographic operations that achieves all of these properties. Since asymmetric primitives often prove inefficient, we show how to modify the protocols to use efficient symmetric cryptography, though at the cost of two properties. Hybrid approaches of these two schemes are possible, but we focus on these two to explore the extremes of the design space. In Section VIII, we quantify their performance trade-offs.

1) Desirable Properties:

1) **Limited Token Validity.** Verifier key material is only valid for a limited time period and is accessible only to valid verifier software. Sender Tokens should have similar restrictions.
2) **Verifier Accountability.** Verifiers should be held accountable for the clients they approve. Thus one verifier should not be able to generate Sender Tokens that appear to originate from another verifier.
3) **Scalability in Filter Count.** The verifier's work, as well as the size of the Sender Token, should be independent of the number of filters.
4) **Topology Independence.** Neither the verifier nor the sender should need to know which filter(s) will see the client's traffic. In many applications, more than one filter may handle the client's traffic, and the number may change over time. Thus, the sender's token must be valid at any filter operated by the same relying party.
5) **Filter Independence.** A filter should not be able to generate Sender Tokens that are valid at other filters. This prevents a rogue filter from subverting other filters.
6) **Client and Filter Accountability.** The relying party should be able to distinguish between a malicious client and a malicious filter. Otherwise, a rogue filter can impersonate a sender.

2) Protocol Specifications: At a high-level, after verifying the trustworthiness of a verifier, the relying party installs the verifier's public key in each of the filters. The verifier, in turn, assesses the trustworthiness of clients. If a verification is successful, the verifier signs the client's public key to create a Sender Token. The client includes this token in each annotated message, and the client module generates annotations by signing its information (e.g., count of emails sent) using the client's private key. Below, we describe these interactions in more detail.

Verifier Attestation. Before giving a verifier the power to authorize client annotations, the relying party must ascertain that the verifier is in a correct, trusted state (Figure 3). It does so via an attestation (Section II). The attestation convinces the relying party that the verifier is running trusted code, that only the trusted code has access to the verifier's private key, and that the keypair is freshly generated.

To prepare for an attestation, the verifier launches trusted verifier code. This code is measured by the platform, and the measurement is stored in the TPM's Platform Configuration Registers (PCRs). In practice (see Section VII), we use a late launch operation to measure and execute a minimal kernel and the code necessary to implement the verifier. The verifier code generates a new public/private keypair and uses the TPM to seal the private key to the current software configuration.

After checking the verifier's attestation, the relying party instructs its filters to accept the verifier's new public key when processing annotated traffic. Since the filter is run by (or acts on behalf of) the relying party, it can be configured with the relying party's public key, and thus verify the authenticity of such updates.

Client Attestation. A similar process takes place when a client requests a Sender Token from a verifier (Figure 4). The client's MiniVisor generates a keypair and attests to the verifier that the private key is bound to the client module and was generated recently. If the client's attestation verifies correctly, the verifier returns a Sender Token consisting of the verifier's ID, the client's public key, an expiration date, and the verifier's signature.

Traffic Annotation. To annotate outbound traffic (e.g., an email or a packet), untrusted code on the client asks the client module to produce an annotation (Figure 5). The untrusted code passes the traffic's contents to the client module. The client module uses its internal state to generate a digest d containing network relevant information about the traffic and/or client; i.e., it may indicate the average bandwidth used by the host, or the number of emails sent. Note that for most applications, the digest will include a hash of the traffic's contents to bind the annotation to a particular piece of traffic. Finally, the module produces an annotation that consists of a unique nonce, the digest, and the client's signature. Untrusted code can then add the client's Sender Token and annotation to the outbound traffic and send it to the relying party.

V	Knows K_{RP}
V	Launches software $Code_V$.
	$Code_V$ recorded in PCRs.
$Code_V$	Generates $\{K_V, K_V^{-1}\}$. Seals K_V^{-1} to $Code_V$.
V	Extends K_V into a PCR.
$RP \rightarrow V$	Attestation request and a random nonce n
$V \rightarrow RP$	K_V, TPM_Quote = $PCRs, Sign_{K_{AIK}^{-1}}(PCRs \Vert n)$, C_{AIK}
RP	Check cert, sig, n, $PCRs$ represent $Code_V$ and K_V
$RP \xrightarrow{*} F_i$	K_V, $Sign_{K_{RP}^{-1}}(K_V)$

Figure 3: **Verifier Attestation.** *V is the verifier, RP the relying party, F_i the filters, and C_{AIK} a certificate for the verifier's AIK.*

C	Launches software $Code_C$.
	$Code_C$ recorded in PCRs.
$Code_C$	Generates $\{K_C, K_C^{-1}\}$. Seals K_C^{-1} to $Code_C$.
C	Extends K_C into a PCR.
$C \rightarrow V$	Token request
$V \rightarrow C$	Attestation request and a random nonce n
$C \rightarrow V$	K_c, TPM_Quote = $PCRs, Sign_{K_{AIK}^{-1}}(PCRs \Vert n)$, C_{AIK}
V	Check cert, sig, n, $PCRs$ represent $Code_C$ and K_C
$V \rightarrow C$	$Token_C = \big[ID_V, K_C, expire_C, H(C_{AIK}),$ $Sign_{K_V^{-1}}(V \Vert K_C \Vert expire_C \Vert H(C_{AIK}))\big]$

Figure 4: **Client Attestation.** *C is the client, $Code_C$ is MiniVisor, V is the verifier, $expire_C$ is an expiration date for the sender's token, and H is a cryptographic hash function.*

$C \rightarrow Code_C$	Traffic contents p.
$Code_C$	Processes p to produce digest d.
$Code_C$	Generates a random nonce m.
$Code_C \rightarrow C$	$Annote_C = (m, d, Sign_{K_C^{-1}}(m \Vert d))$
$C \rightarrow RP$	$p, Token_C, Annote_C$

Figure 5: **Traffic Annotation.** *C is the client, $Code_C$ is the client module from Section IV-B1 and RP is the relying party. The digest d represents the module's summary of network-relevant information about the client and/or traffic. The client sends the traffic to the relying party, but it will be processed along the way by one or more filters.*

1: **if** p contains $Token_C, Annote_C$ **then**
2: $(ID_V, K_C, expire_C, H, Sig_V) \leftarrow Token_C$
3: Verify Sig_V using K_V.
4: Use $expire_C$ to check that $Token_C$ has not expired.
5: $(m, d, Sig_C) \leftarrow Annote_C$
6: Verify Sig_C using K_C.
7: Check that pair (K_C, m) is unique.
8: Insert (K_C, m) into Bloom Filter.
9: **if** All verifications succeed **then**
10: Accept d as an authentic annotation of p
11: **else**
12: Drop p

Figure 6: **Filtering Annotations.** *Processing traffic p*

Annotation Checking. Filters that receive annotated traffic can verify its validity using the filtering algorithm shown in Figure 6. The filter uses the verifier's ID to look up the corresponding public key provided by the relying party. It uses the key to verify the authenticity and freshness of the client's Sender Token. The filter may optionally decide to cache these results to speed future processing. It then checks the authenticity and uniqueness of the annotation. It stores a record of the nonce to prevent duplication and accepts the validity of the annotation's digest if it passes all verification checks. However, if an annotation's verification checks fail, the filter drops the traffic. Legitimately generated traffic will only fail to verify if an on-path adversary modifies the traffic. Such an adversary can also drop or alter the traffic, so dropping malformed traffic does not increase the adversary's ability to harm the client.

3) A Symmetric Alternative: The protocols shown above possess all of the properties described in Section IV-C1. Unfortunately, they require the client to compute a public-key signature for each item of traffic sent and the filter to verify two public-key signatures per annotation. The challenge is to improve the efficiency while retaining as many of the properties from Section IV-C1 as possible.

At a high-level, instead of giving the verifier's public key to the filters, we establish a shared symmetric key between each verifier and all of the filters. Similarly, the client uses a symmetric, rather than a public, key to authenticate its annotations. The verifier provides the client with this key, which is calculated based on the symmetric key the verifier shares with the filters, as well as the information in the client's Sender Token. This makes it unnecessary for the verifier to MAC the client's token, since any changes to the token will cause the filters to generate an incorrect symmetric key, and hence to reject client's annotations. We describe these changes in more detail below.

Verifier Attestation. The last step of the protocol in Figure 3 is the only one that changes. Instead of sending the verifier's public key to all of the filters, the relying party generates a new symmetric key K_{VF}. The relying party encrypts the key using the verifier's newly generated public key and sends the verifier the resulting ciphertext ($Encrypt_{K_V}(K_{VF})$). Since the corresponding private key is sealed to the verifier's trusted code, the relying party guarantees that the symmetric key is protected. The relying party also encrypts the key and sends it to each of the filters, establishing a shared secret between the verifier and the filters.

Client Attestation. The protocol shown in Figure 4 remains the same, except for two changes. First, when the client sends its token request, it includes a randomly chosen client identifier ID_C. The larger change is in the Sender Token returned by the verifier. To compute the new token, the verifier first computes a symmetric key that the client uses to authorize annotations:

$$K_{CF} = PRF_{K_{VF}}(V \| ID_C \| expire_C), \tag{1}$$

where PRF is a secure pseudo-random function. The verifier then sends the client: $Encrypt_{K_C}(K_{CF})$, $Token = (V, ID_C, expire_C)$. The attestation convinces the verifier that K_C^{-1} is bound to trusted code, i.e., only trusted code can obtain K_{CF}. Without knowing K_{VF}, no one can produce K_{CF}.

Traffic Annotation. Traffic annotation is the same as before, except that instead of producing a signature over the traffic's contents, the code module produces a Message Authentication Code (MAC) using K_{CF}, an operation that is orders of magnitude faster.

Annotation Checking. The algorithm for checking annotations remains similar. Instead of checking the verifier's signature, the filter regenerates K_{CF} using Equation 1 and its knowledge of K_{VF}. Instead of verifying the client's signature on the annotation, the filter uses K_{CF} to verify the MAC. As a result, instead of verifying two public key signatures, the filter calculates one PRF application and one MAC, operations that are three orders of magnitude faster.

This scheme achieves the first four properties listed in Section IV-C1, but it does not provide properties 5 and 6. Since each verifier shares a single symmetric key with all filters, a rogue filter can convince other filters to accept bogus annotations. We could prevent this attack by having the relying party establish a unique key for each verifier-filter pair, but this would violate another property. Either the verifier would have to MAC the client's Sender Token using all of the keys it shares with the filters (violating the fourth property), or the verifier would have to guess which filters would see the client's traffic, violating our topology-independence property.

Similarly, since the client and the filter share a symmetric key, the relying party cannot distinguish between malicious filters and malicious clients. Nonetheless, since the relying party's operator controls the filters, such risks should be acceptable in many applications, given the dramatic performance benefits offered by the symmetric scheme.

D. User Privacy and Client Revocation

To encourage adoption, Assayer must preserve user privacy, while still limiting clients to one identity per machine and allowing the relying party to revoke misbehaving clients. The Direct Anonymous Attestation (DAA) protocol [6] was designed to provide exactly these properties. However, as mentioned in Section II, available TPMs do not yet implement this protocol, so until DAA becomes available on TPMs (or whatever secure hardware forms the basis for Assayer), Assayer must imperfectly approximate it using structured AIK certificates. We emphasize that this is a *temporary* engineering hack, not a fundamental limitation of Assayer, since DAA demonstrates that we can achieve both privacy and accountability.

Recall from Section II that TPM-equipped clients sign attestations using randomly generated attestation identity keys (AIKs). A Privacy CA issues a limited-duration certificate that vouches for the binding between an AIK and the original TPM Endorsement Key (EK). With Assayer, clients obtain AIK certificates that specify that the AIK is intended for communicating with a specific relying party. Using a different AIK for each relying party prevents the relying parties from tracking the client across sites. However, since all of the AIKs are certified by the same EK, they can all be bound to a single installation of MiniVisor, preventing an attacker from using a separate MiniVisor for each destination.

Of course, similar to issuing multiple DNS lookups using the same source IP address, this approach allows the Privacy CA to learn that *some* client intends to visit a particular set of relying parties. The DAA protocol eliminates both this linkage and the reliance on the Privacy CA.

To preserve user privacy with respect to a single relying party, the client can generate a new AIK and request a new certificate from the Privacy CA. However, Privacy CAs may only simultaneously issue one AIK certificate per relying party per TPM EK. Thus, a client could obtain a 1-day certificate for an AIK, but it could not obtain another certificate for the same relying party until the first certificate expires. This prevents a client from generating multiple *simultaneous* identities for communicating with a particular relying party.

Since each client token contains a hash of the client's AIK certificate, if the relying party decides a client is misbehaving, it can provide the hash to the Privacy CA and request that the Privacy CA cease providing relying party-specific AIK certificates to the EK associated with that particular AIK. This would prevent the client from obtaining new AIKs for communication with this particular relying party, though not for other relying parties. Similarly, the relying party can instruct its verifiers and filters to cease accepting attestations and annotations from that AIK.

V. Potential Attacks

We analyze potential attacks on the generic Assayer architecture and show how Assayer defends against them.

A. Exploited Clients

Code Replacement. An attacker may exploit code on remote, legitimate client machines. If the attacker replaces MiniVisor or the client module with malware, the TPM will refuse to unseal the client's private key, and hence the malware cannot produce authentic annotations. Without physical access to the client's machine, the attacker cannot violate these hardware-based guarantees.

Code Exploits. An adversary who finds an exploit in trusted code (i.e., in MiniVisor or a client module) can violate Assayer's security. This supports our argument that trusted client code should be minimized as much as possible. Exploits of untrusted code are less problematic. The relying party trusts MiniVisor to protect the client module, and it trusts the client module to provide accurate annotations. Thus, the trusted client module will continue to function, regardless of how the adversary exploits the untrusted code.

Flooding Attacks. Since an attacker cannot subvert the annotations, she might instead choose to flood Assayer components with traffic. As we explained in Section IV-B2, the verifiers are designed to withstand DoS attacks, so flooding

them will be unproductive. Since filters must already check annotations efficiently to prevent bottlenecks, flooding the filters (with missing, invalid, or even valid annotations) will not hurt legitimate traffic throughput.

Annotation Duplication. Since MiniVisor does not maintain control of the client's network interface, an attacker could ask the client module to generate an annotation and then repeatedly send the same annotation, either to the same filter or to multiple filters. Because each authorized annotation contains a unique nonce, duplicate annotations sent to the same filter will be dropped. Duplicates sent to different filters will be dropped as soon as the traffic converges at a single filter downstream. Section VII-D discusses our Bloom Filter implementation for duplicate detection.

B. Malicious Clients

Beyond the above attacks, an attacker might use hardware-based attacks to subvert the secure hardware on machines she physically controls. For example, the adversary could physically attack the TPM in her machine and extract its private keys. This would allow her to create a fake attestation, i.e., convince the verifier that the adversary's machine is running trusted Assayer code, when it is not.

However, the adversary can only extract N TPM keys, where N is the number of machines in her physical possession. This limits the attacker to N unique identities. Contacting multiple verifiers does not help, since sender identities are tracked based on their AIKs, not on their Sender Tokens. As discussed in Section IV-D, at any moment, each TPM key corresponds to exactly one AIK for a given relying party. Furthermore, to obtain a Sender Token from the verifier, the attacker must commit to a specific relying-party-approved client module. If the attacker's traffic deviates from the annotations it contains, it can be detected, and the attacker's AIK for communicating with that relying party will be revoked. For example, if the attacker's annotations claim she has only sent X packets, and the relying party detects that the attacker has sent more than X packets, then the relying party knows that the client is misbehaving and will revoke the AIK the client uses to communicate with this relying party (see Section IV-D). Since the Privacy CA will not give the attacker a second AIK for the same relying party, this AIK can only be replaced by purchasing a new TPM-equipped machine, making this an expensive and unsustainable attack.

C. Rogue Verifiers

A rogue verifier can authorize arbitrary clients to create arbitrary annotations. However, the verifier's relatively simple task makes its code small and easy to analyze. The attestation protocol shown in Figure 3 guarantees that the relying party only approves verifiers running the correct code. Since verifiers are owned by the relying party or by someone with whom the relying party has a contractual relationship, local hardware exploits should not be a concern. Revocation can be performed by refusing to renew the verifier's key material, or by actively informing the filters that they should discard the rogue verifier's public key.

Focus		Policy Creator: Recipient	Policy Creator: Network
	Concentrated	Spam	DDoS
	Diffuse	Spam	Super Spreaders

Figure 7. **Case Studies.** *Our case studies can be divided based on who determines acceptable policies and how focused the attack traffic is. For example, spammers can send a large amount of spam to one recipient or a few spam messages to many recipients.*

D. Rogue Filters

A rogue filter can discard or mangle annotated traffic, or give priority to attack traffic. However, since it sits on the path from the client to the relying party, a rogue filter can already drop or alter traffic arbitrarily. Fortunately, since the filters are directly administered by the relying party and perform a relatively simple task, rogue filters should be rare.

VI. Case Studies

To evaluate the power of trustworthy host-based information, we consider the usefulness of Assayer in three diverse applications (see Figure 7). For each application, we motivate why Assayer can help, explain how to instantiate and deploy Assayer's components, and consider the motivations for clients to deploy the system. While we present evidence that Assayer-provided information can help in each application, it is beyond the scope of this work to determine the optimal statistics Assayer should provide or the optimal thresholds that relying parties should set.

A. Spam Identification

Motivation. Numerous studies [7, 15, 28] suggest that spam can be distinguished from legitimate email based on the sender's behavior. For example, one study found that the time between sending two emails was typically on the order of minutes [38], whereas the average email virus or spammer generates mail at a much higher rate (e.g., in the Storm botnet's spam campaigns, the average sending rate was 152 spam messages per minute, per bot [22]). Another study [15] found that the average and standard deviation of the size of emails sent over the last 24 hours were two of the best indicators of whether any individual email was spam.

These statistics can be difficult to collect on an Internet-scale, especially since spammers may adopt stealthy strategies that send only a few spam messages to each domain [28], but still send a large amount of spam in aggregate. However, the host generating the email is in an ideal position to collect these statistics.

Of course, host-based statistics are not sufficient to definitively identify spam, and they may falsely suggest that legitimate bulk email senders are spammers. However, by combining these statistics with existing spam classification techniques [20, 28], we expect that spam identification can be significantly improved for most senders.

Instantiation. To aid in spam identification, we can design a client module for annotating outbound email with relevant statistics, e.g., the average size of all emails generated during the last 24 hours. Each time the client generates a new email, it submits the email to the client module. The client module creates an authenticated annotation for the email with the relevant statistics, which the untrusted client email software can add as an additional email header.

The relying party in this case would be the email recipient's mail provider, which would designate a set of verifiers. The filter(s) could simply be added as an additional stage in the existing spam identification infrastructure. In other words, the filter verifies the email's annotation and confirms the statistics it contains. These statistics can then be used to assess the probability that the message is spam.

Client Incentives. Legitimate clients with normal sending behavior will have an incentive to deploy this system, since it will decrease the chance their email is marked as spam. Some mail domains may even require endhosts to employ Assayer before accepting email from them.

Malicious clients may deploy Assayer, but either they will continue to send email at a high rate, which will be indicated by the emails' annotations, or they will be forced to reduce their sending behavior to that of legitimate senders. While not ideal, this may represent a substantial reduction in spam volume. Finally, malicious clients may send non-annotated spam, but as more legitimate senders adopt Assayer, this will make spam more likely to be identified.

B. *Distributed Denial-of-Service (DDoS) Mitigation*

Motivation. Like spam, DDoS is an attack in which adversaries typically behave quite differently from benign users. To maximize the effectiveness of a network-level attack, malicious clients need to generate as much traffic as possible, whereas a recent study indicates that legitimate clients generate much less traffic [5]. To confirm this, we analyzed eight days (135 GB of data representing about 1.27 billion different connections) of flow-level network traces from a university serving approximately 9,000 users. If we focus, for example, on web traffic, we find that 99.08% of source IP addresses never open more than 6 simultaneous connections to a given destination, and 99.64% never open more than 10. Similarly, 99.56% of source IP addresses send less than 10 KBps of aggregate traffic to any destination. This is far less than the client links permit (10-1000 Mbps). Since the traces only contain flow-level information, it is difficult to determine whether the outliers represent legitimate or malicious traffic. However, other application traffic, such as for SSL or email, shows similar trends; the vast majority of users generate very little outbound traffic. This suggests that it is indeed possible to set relatively low thresholds to mitigate DDoS activity while leaving virtually all legitimate users unaffected.

Assayer enables legitimate hosts to annotate their traffic with additional information to indicate that their traffic is benign. For example, the client module might annotate each packet to show the rate at which the client is generating traffic. By prioritizing packets with low-rate annotations, filters ensure that, *during DDoS attacks*, legitimate traffic will be more likely to reach the server. A few non-standard legitimate clients may be hurt by this policy, but the vast majority will benefit.

Of course, this approach will only prevent attackers from sending large floods of traffic from each machine they control. They can still have each machine send a low rate of traffic, and, if they control enough machines, the aggregate may be enough to overwhelm the victim. Nonetheless, this will reduce the amount of traffic existing botnets can use for attack purposes and/or require attackers to build much larger botnets to have the same level of effect on the victim.

Numerous other systems have been proposed to fight DDoS. Packet capability systems, such as TVA [39] could use Assayer to decide whether to provide a client with a capability. Resource-based systems, such as Portcullis [26] or speak-up [35], attempt to rate-limit clients or enforce equal behavior amongst clients using checkable resource consumption, such as CPU or bandwidth. Assayer provides a more direct view into endhost behavior, but relies on secure hardware to bootstrap its guarantees. Overlay systems such as SOS [19] and Phalanx [8] use a large system of nodes to absorb and redirect attack traffic. This approach is largely orthogonal to Assayer, though combining these two approaches could be promising. For example, Assayer could keep track of overlay-relevant statistics on the host, or help provide per-client fair queuing within the overlay.

Instantiation. On the client, we modify the untrusted network stack to submit outbound packets to the Assayer client module. The client module generates an annotation indicating the number or size of packets generated recently. In this case, the relying party is the server the client is attempting to access (e.g., a website or software update server). The filters can prioritize annotated packets with "legitimate-looking" statistics over other traffic. To avoid hurting flows destined to other servers, filters give preference to annotated packets *relative* to other packets destined to that same server. This could be implemented via per-destination fair-queuing, with the Assayer-enabled server's queue configured to give priority to approved packets.

To combat network-level DDoS attacks, we need to prioritize annotated packets as early as possible, before they reach the server's bottleneck. The filters must also be able to verify packet annotations efficiently at line rates. We envision filter deployment occurring in phases, dictated by the server operator's needs and business relationships. Initially, the server operator may simply deploy a single filter in front of (or as a part of) the server. However, to combat network-level attacks, the server's operator may contract with its ISP to deploy filters at the ISP's ingress links. Similar arrangements could be made with other organizations around the network, depending on the business relationships available. In the

long run, filters will likely become standardized, shared infrastructure that is deployed ubiquitously. However, as we show in Section VIII-D, partial deployment can provide significant protection from attacks.

Client Incentives. The client proves that it is generating traffic at a moderate rate in exchange for elevated service from the network and server. This makes it more likely that the client can access the web services it desires, even during DDoS attacks.

C. Super-Spreader Worm Detection

Motivation. A super-spreader worm exploits a host and then begins rapidly scanning hundreds or even thousands of additional hosts for potential vulnerabilities. Again, studies show that rapidly contacting multiple hosts, as such worms do, is quite unlike typical user behavior [37]. While detecting such traffic is relatively straightforward on an enterprise scale (using tools such as NetFlow), detecting this behavior on an Internet scale is much more challenging, since any individual monitor has only a limited view of an endhost's behavior, and the amount of traffic sent to each host is so small (often no more than a single packet) that it can be missed by sampling algorithms.

These observations have lead to the development of multiple algorithms [34, 36] for trading detection accuracy for reduced state and processing overhead. However, ultimately, the host generating the scanning traffic is in the best position to detect this behavior, since it can keep a perfectly accurate local count of how many hosts it has contacted, assuming that the network can indeed trust the host's count. While middleboxes might perform this service in enterprise networks, home networks and small businesses are less likely to install such additional hardware. An Assayer approach would also allow remote destinations to verify that incoming packets do not constitute worm propagation. Nonetheless, deployment issues remain, as we discuss below.

Instantiation. For this application, the client software would again submit packets to the client module, which would produce annotations indicating the number of destinations contacted in the last X minutes. The relying party could be a backbone ISP hoping to avoid worm-based congestion, or a stub ISP protecting its clients from worm infections. In-network filters can verify the authenticity of these annotations and drop or delay packets from hosts that have contacted too many destinations recently. Like in the DDoS application, filters would need to be deployed at the edges of the relying party's administrative domain and would need to be able to verify annotations at line rate.

Client Incentives. While it is interesting to see that Assayer enables super-spreader worm detection from a technical perspective, there are several challenges in incentivizing clients and in dealing with non-annotated traffic. These challenges suggest that, despite its technical feasibility, Assayer may not be an optimal choice for this application.

While a next-generation clean-slate network could simply mandate the use of Assayer to detect super-spreader worms, deploying Assayer in a legacy environment faces a bootstrapping challenge. Unlike in the DDoS application, it is not sufficient to simply prioritize annotated traffic over non-annotated traffic, since lower-priority traffic will still spread the worm. Instead, non-annotated traffic must be significantly delayed or dropped to have a credible chance of slowing or stopping worm propagation. However, in a legacy environment, ISPs cannot slow or drop legacy traffic until most users have started annotating their traffic, but users will not annotate their traffic unless motivated to do so by the ISPs. A non-technical approach would be to hold users liable for any damage done by non-annotated packets, thus incentivizing legitimate users to annotate their packets. This obviously raises both legal and technical issues.

VII. Implementation

To evaluate the effectiveness and performance of Assayer, we have developed a basic prototype system. Because these are prototypes, they give rough upper-bounds on Assayer's performance impact, but considerable room for optimization remains. We present our performance results in Section VIII.

A. Client Architecture

We implemented the client configuration shown in Figure 2 employing a tiny hypervisor called MiniVisor that we developed using hardware-virtualization support available from both AMD and Intel. Since MiniVisor does not interact with any devices, we were able to implement it in 841 lines of code and still (as shown in Section VIII) offer excellent performance. It supports a single hypercall that allows untrusted code to submit traffic to a client module and receive an annotation in return.

We employ a late launch operation (recall Section II) to simplify client attestations by removing the early boot code (e.g., the BIOS and bootloader) from the set of trusted code. When MiniVisor is late launched, it uses shadow page tables to isolate its own private memory area and then boots the Linux kernel for normal client usage. The client attestations consist of the protection layer (MiniVisor), a client module, and the fact that the protection layer is configured to properly isolate the module.

For packet-based applications (e.g., DDoS mitigation and super-spreader detection), we use Linux's TUN/TAP interface to re-route outbound packets to a user-space program. The user-space program invokes MiniVisor's hypercall to obtain an annotation and then routes the packet back to the physical interface. This configuration simplified development, but it is less than optimal from a performance standpoint. Intercepting packets inside of a network driver or kernel module would improve our performance. For these applications, adding annotations to packets could potentially cause considerable packet fragmentation. To prevent this, we reduce the MTU on the network interface by the number of bytes in a standard annotation.

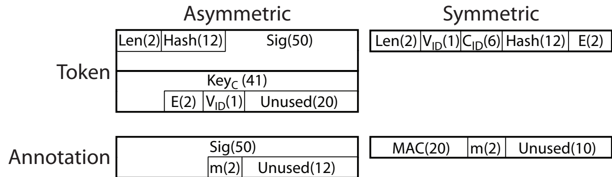

Figure 8. **Token and Annotation Layout.** *Byte-level layout for Sender Tokens and traffic annotations. The two are shown separately for clarity, but in practice, would be packed together. E is an expiration date, and m is a randomly-chosen nonce.*

B. Client Verification

Regardless of the application, the client must be able to create an attestation that can be checked by a verifier. Thus, we developed generic client software to produce the attestations, as well as a verifier server program to check the attestations and produce client tokens. Together, they implement the protocol shown in Figure 4. Since the code that allows the relying party to check verifier attestations (Figure 3) is very similar (and less performance-sensitive), we describe and evaluate only the client attestation and verification implementations.

Client Attestations. Before it can create an attestation, our client code first generates an AIK and obtains an AIK certificate from a Privacy CA. To create an attestation, the client contacts the verifier and requests a nonce. Given the verifier's nonce, the client invokes a TPM_Quote operation. It sends the verifier the public key created by its code module, the contents of the PCRs, the list of the code described by the PCRs, the TPM's signature and the AIK certificate. The verifier checks the validity of the certificate, verifies the TPM's signature, checks that the nonce value is the same one it sent, and finally checks to make sure the PCR values reflect an appropriate version and configuration of MiniVisor. Assuming these checks pass, it returns an appropriate Sender Token.

Verifier Implementation. Our verifier prototype is implemented as a simple user-space server program. The implementation is based on a Unix/Linux preforked server library (spprocpool), and the client and the verifier communicate using UDP. The verifier pre-forks several worker processes and waits for client connections. When it receives a connection, the verifier passes this connection to an idle worker process. The worker process chooses a random nonce for the client and verifies the resulting attestation. A more sophisticated server architecture would undoubtedly improve our system's performance, but this simple prototype gives us an upper-bound on a verifier's potential performance.

C. Traffic Annotation

To evaluate their relative performance, we implemented both the asymmetric and symmetric protocols for generating annotations (Section IV-C). Figure 8 illustrates the layout of the Sender Tokens and traffic annotations for each scheme.

With both schemes, we add the Sender Token and the annotation to the payload itself, and then adjust the appropriate header fields (length, checksum, etc.). This provides compatibility with legacy network devices. The traffic recipient needs to remove this information before handing the payload to applications, but this is simple to implement. Of course, legacy traffic will not contain annotations, and hence is easy to identify and handle in an application-specific manner.

With the asymmetric scheme, we use elliptic curve cryptography to minimize the size of the client's public key. We use the secp160k1 curve, which provides approximately 80 bits of cryptographic strength. The verifier uses ECDSA to sign the client's token, and the client also uses ECDSA to sign the contents of authorized packets. In sum, the client's token takes 108 bytes, and the annotation requires 52 bytes.

With the symmetric scheme, if we use a 160-bit key with SHA1-HMAC, then the client's token only requires 23 bytes, and the annotation requires 22 bytes.

D. Filter

We implemented the filter's functionality (Figure 6) both in userspace (for applications such as spam filtering), and as a module for the Click router [21] (for applications such as DDoS mitigation and super-spreader detection). Both implementations check traffic (either email or packets) for an Assayer flag in the header fields. If present, the filter checks the Sender Token and the annotation, following the algorithm in Figure 6.

To detect duplicate annotations, we use a Bloom Filter [4]. We only insert an annotation into the Bloom Filter after verifying the Sender Token and the annotation. The Bloom Filter ensures that a valid annotation is unique in a given time period t with a bounded false positive probability γ.

To illustrate this, suppose that the filter receives N valid annotations/second. If we use K different hash functions, and N different annotations are added into a Bloom Filter of M bits, then γ is approximately $(1 - e^{\frac{-KN}{M}})^K$ [4]. For network applications, suppose the filter operates on a 1 Gbps link. In the worst case, the filter would receive n packets/sec, where n is the link's capacity divided by the minimum packet size, and all of these packets carry valid annotations. In the symmetric scheme, $n = 1,262,626$ packets/second. Thus, to limit the false positive probability to less than $\frac{1}{10^6}$ per annotation, we need a 2MB Bloom Filter with 20 hash functions. Similar calculations can be applied to less performance-intensive applications, such as spam identification.

If we use public hash functions in our Bloom Filter, an adversary could use carefully chosen inputs to pollute the Bloom Filter, i.e., use a few specially-crafted annotations to set nearly all the bits in the Bloom Filter to 1. This attack would dramatically increase the false positive rate and break the duplicate detection. Thus, we use a pseudorandom function (AES) with a secret key known only to the filter to randomize the input to the Bloom Filter.

VIII. Evaluation

To identify potential performance bottlenecks in the Assayer architecture, we evaluated the performance of each prototype component and compared our two authentication schemes. In the interest of space, and since our spam detection application is far less latency-sensitive than our packet-level applications (DDoS and worm mitigation), we focus our evaluation on Assayer's packet-level performance. We also developed an Internet-scale simulator to evaluate how Assayer performs against DDoS attacks by large botnets.

We find that, as expected, the symmetric authentication scheme outperforms the asymmetric scheme by 1–2 orders of magnitude. Using the symmetric scheme, MiniVisor's performance is quite close to native, with network overheads ranging from 0-11%. Our verifier can sustain about 3300 verifications/second, and the filter can validate Assayer traffic with only a 3.7-18.3% decrease in throughput (depending on packet size). Finally, our simulations indicate that even sparse deployments (e.g., at the victim's ISP) of Assayer offer strong DDoS mitigation during large-scale attacks.

In our experiments, our clients and verifier run on Dell Optiplex 755s, each equipped with a 3 GHz Intel Core2 Duo and 2 GB of RAM. The filter has one 2.4 GHz Intel(R) Pentium(R) 4 with 512 MB of memory. All hosts are connected via 1 Gbps links.

A. Client Verification

We measure the time it takes a single client to generate an attestation and obtain a Sender Token from a verifier. We also evaluate how many simultaneous clients our verifier can support.

1) Client Latency: Since clients request new Sender Tokens infrequently (e.g., once a week), the latency of the request is unlikely to be noticed during normal operation. Nonetheless, for completeness, we measured this time using our prototype client and verifier and found that the client takes an average of 795.3 ms to obtain a Sender Token. The vast majority (99.7%) of the time is spent obtaining a quote from the TPM, since the quote requires the calculation of a 2048-bit RSA signature on a resource-impoverished TPM processor. The verifier only spends a total of 1.75 ms processing the client's request using the symmetric scheme and 3.58 ms using the asymmetric scheme.

2) Verifier Throughput: To test the throughput of the verifier, we developed a minimal client program that requests a nonce and responds with a pre-generated attestation as soon as the verifier responds. The client employs a simple timeout-based retransmission protocol. We launch X clients per second and measure the time it takes each client to receive its Sender Token. In our tests, each of our 50 test machines simulates 20-500 clients.

In 10 trials, we found that a single verifier using the symmetric scheme can serve a burst of up to 5700 clients without any UDP retransmission, and can sustain an average rate of approximately 3300 clients/second. With the asymmetric scheme, a verifier can serve 3800 clients in a burst, and can sustain about 1600 clients/second. This implies that our simple, unoptimized verifier prototype could, in a day, serve approximately 285 million clients with the symmetric scheme and 138 million clients with the asymmetric scheme.

Annotation Size	Symmetric	Asymmetric
10 B	2.11	1166.40
100 B	3.15	1156.75
1,000 B	5.00	1154.20
10,000 B	27.20	1180.45
100,000 B	247.55	1396.40
1,000,000 B	2452.15	3597.95
10,000,000 B	24696.25	25819.75

Figure 9. **Annotation Generation.** *Average time required to generate annotations using our symmetric and asymmetric protocols. All times are in microseconds.*

B. Client Annotations

With Assayer, clients must compute a signature or MAC for each annotation. Annotating traffic adds computational latency and reduces effective bandwidth, since each traffic item (e.g., email or packet) carries fewer bytes of application data.

Figure 9 summarizes the results of our microbenchmarks examining the latency of generating annotations of various sizes. All results are the average of 20 trials. We expect applications such as DDoS mitigation and super-spreader worm detection to require small annotations (since the annotation must fit in a packet), while spam identification may require larger annotations. The symmetric scheme's performance is dominated by the cost of hashing the annotation. With the asymmetric scheme, the performance for smaller annotations is dominated by the time required to generate the ECDSA signature. Thanks to MiniVisor's use of hardware support for virtualization, context switching to the client module is extremely fast (approximately 0.5 μs).

For macrobenchmarks, since email is designed to be delay tolerant, we focus on quantifying Assayer's effect on packet-level traffic. Thus, we evaluate the effect of annotating each outbound packet with the number of packets sent, along with a hash of the packet's contents. We first ping a local host (**Ping L**), as well as a host across the country (**Ping R**). This quantifies the computational latency, since each ping only uses a single packet and bandwidth is not an issue. We then fetch a static web page (8 KB) (**Req L/R**) and download a large (5 MB) file from a local web server and from a web server across the country (**Down L/R**). These tests indicate the performance impact a user would experience during an average web session. They require our client module to annotate the initial TCP handshake packets, the web request, and the outbound acknowledgements. To quantify the impact of Assayer's bandwidth reduction, we also measure the time to upload a large (5 MB) file (**Up L/R**). This test significantly increases the number of packets the client module must annotate.

	Throughput (Mbps)	% of Click
Basic Click (user)	124	-
Sym Filter (user)	87	70.1%
AsymFilter (user)	2	1.7%
Basic Click (kernel)	225	-
Sym Filter (kernel)	154	68.4%
Sym Filter (kernel, no dup)	169	75.1%
Sym Filter (kernel, UMAC)	204	90.7%

Figure 12. **Packet Filtering Performance.** *"User" and "kernel" denote user-level and kernel-level mode. "Sym" and "asym" denote the symmetric scheme and the asymmetric scheme. "Basic Click" is the basic click router which simply forwards each packet. "no dup" means no duplicate detection operations are performed. All tests employ 512-byte packets.*

We performed the above experiments using both the asymmetric and the symmetric schemes described in Section IV-C. Figure 10 summarizes our results. The symmetric scheme adds less than 12% overhead, even in the worst-case tests that involve uploading a large file. In many cases, the difference between the symmetric scheme and native Linux is statistically insignificant. The asymmetric scheme, on the other hand, adds significant overhead, though the effects are mitigated for remote hosts, since round-trip times occupy a large portion of the test. We could reduce the overhead by selecting a scheme that allows more efficient signing, but this would increase the burden on the filters.

C. Filter Throughput

In order to evaluate the filter's throughput inspecting packet-level annotations, we use the Netperf tools running on a client machine to saturate the filter's inbound link with annotated packets. To compare our various schemes, we launch the Netperf TCP_STREAM test using 512-byte packets, which is close to the average packet size on the Internet [32]. We then experiment with varying packet sizes.

In our experiments (Figure 12), we found that a user-level basic Click router, which simply forwards all packets, could sustain a throughput of approximately 124 Mbps. A user-level filter implementing our symmetric annotation scheme has about 87 Mbps throughput, while a filter using the asymmetric scheme can only sustain approximately 2 Mbps.

By implementing the filter as a Click kernel module, we improve the performance of the filter using the symmetric scheme to about 154 Mbps, while the performance of a kernel-level basic Click router is approximately 225 Mbps. The filter using the symmetric scheme performs two major operations: verifying packet annotations and detecting duplicate annotations (see Section IV-C3). Eliminating the duplicate detection operation only slightly improves the filter's throughput (up to 169 Mbps), which indicates that verifying the packet annotation is the significant performance bottleneck.

To confirm this, we modify our packet annotation implementation to use UMAC [23] instead of SHA1-HMAC. To improve UMAC's performance, we implement a key cache mechanism that only generates and sets up a UMAC key for the first packet of every network flow, since all of the packets in a network flow will have the same Sender Token. Measurements indicate that the average Internet flow consists of approximately 20 packets [32]. Using this measurement as a rough estimate of our key cache's effectiveness, our filter's performance improves to 204 Mbps. This represents a 9.3% performance loss relative to a kernel-level basic Click router.

Finally, we vary the packet size used in our experiments. We find that our UMAC-based symmetric filter undergoes a 18.3% performance loss relative to Click when using 100 byte packets, whereas it comes within 3.7% of Click when using 1500 byte packets.

D. Internet-Scale Simulation

Finally, to evaluate Assayer's effectiveness for DDoS mitigation, we developed an Internet-scale simulator. The simulation's topology was developed from the CAIDA Skitter probes of router-level topology. The Skitter map forms a rooted tree at the trace source and spans out to over 174,000 endpoints scattered largely uniformly around the Internet. We make the trace source the victim of the DDoS attack and then randomly select 1,000 endpoints to represent legitimate senders and 100,000 endpoints to represent attackers. We assume that legitimate senders have obtained Sender Tokens, whereas the attackers simply flood (since flooding with Sender Tokens will result in the revocation of the attacker's keys – see Section IV-D).

Since the Skitter map does not include bandwidth measurements, we use a simple bandwidth model in which each endhost has a small uplink that connects it to a well-provisioned core that narrows down when it reaches the victim. More precisely, senders have 10 Mbps connections, the victim has a 100 Mbps link, and the links in the middle of the network operate at 1 Gbps. In our experiments, legitimate senders make one request every 10 ms, while attackers flood the victim with requests at their maximum uplink capacity.

We run our simulations with no Assayer deployment, with Assayer filters deployed at the victim's ISP, and with ubiquitous (full) Assayer deployment. Figure 11 shows the amount of time it takes legitimate senders to contact the server. With no deployment, less than 6% of legitimate clients can contact the server, even after 10 seconds. With a full deployment of Assayer, most clients contact the server within one RTT, which is unsurprising given that legitimate traffic enjoys priority over the attack traffic throughout the network. However, even with partial deployment at the victim's ISP, more than 68% of legitimate clients succeed in less than a second, and 95% succeed within 10 seconds.

IX. Potential Objections

In this section, we consider potential objections to the notion of conveying endhost information to the network.

	Native Linux		Assayer Symmetric		Assayer Asymmetric	
Ping L	0.817	±0.32	0.811	±0.13 (-0.1%)	2.104	±0.31 (+157.5%)
Ping R	11.91	±1.90	11.99	±3.24 (+0.1%)	14.03	±3.67 (+17.8%)
Req L	3.129	±0.03	3.48	±0.26 (+11.3%)	12.27	±4.18 (+292.1%)
Req R	45.83	±12.3	44.07	±6.93 (-0.4%)	51.35	±12.1 (+12.0%)
Down L	1339.	±348	1427.	±382 (+6.6%)	2634.	±114 (+96.7%)
Down R	5874.	±1000	5884.	±990 (+0.2%)	6631.	±721 (+12.8%)
Up L	706.5	±61.4	777.4	±153 (+10.0%)	5147.	±177 (+628.5%)
Up R	3040.	±568	3078.	±1001 (+0.1%)	6234.	±961 (+105.1%)

Figure 10: Client Annotations: Symmetric Vs. Asymmetric. ***L represents a local request, and R represents a remote request. All times are shown in milliseconds rounded to four significant figures. Values in parentheses represent the change versus native.***

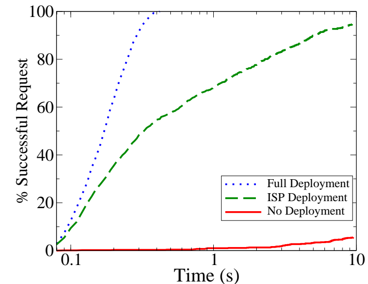

Figure 11: ***Time for 1,000 senders to contact the server in the presence of 100,000 attackers. Note the log scale.***

A. Isn't the TPM Evil?

No. The TPM is a limited security co-processor. Like any technology, it can be used for good or for evil. While it might be used for unsavory purposes, it also holds enormous potential to improve host and network security. This paper explores the potential benefits from a network's perspective.

B. Why Not Collect Information on the Local Router?

One might imagine that instead of collecting statistics on the host, we could instead collect them on the local router, which has almost as good a view on host behavior as the host itself. However, this would misalign resource and incentives. Compared to endhosts, routers tend to be resource impoverished and lack the secure hardware to convey information to remote parties. Even if they had such hardware, it is much harder to decide how much traffic a router should be allowed to send. For example, a spam filter might be able to develop a reasonable model of how much email a normal user (represented by a single TPM-equipped machine) might send [15], but it seems more challenging to model how much email a TPM-equipped router might be expected to forward. Furthermore, a sender's ISP has relatively little incentive to help a remote destination filter out spam, whereas the user has a strong incentive to ensure her email is not marked as spam. Nonetheless, as we note in Section XI, we consider the development of an ISP-based proxy an interesting direction for future work.

C. Is This Really Deployable Incrementally?

No, in the sense that if a single party deploys Assayer, they will not see any benefit; this is true of other useful services such as telephones, email, etc. Yes, in the sense that for many applications, once a single server deploys an Assayer filter, individual senders can upgrade to Assayer and immediately see benefits. For example, a server concerned about DDoS can install an Assayer filter. Any client that upgrades will see their traffic prioritized during an attack, whereas legacy clients (easily distinguished by the lack of annotations) will notice the same degradation they would today, potentially incentivizing them to upgrade as well. As discussed in Section VI-C, not all applications are structured this way; one contribution of this work is to identify which applications will immediately benefit from an Assayer approach, and which face deployment challenges.

X. Related Work

Conveying Generic Host Information to the Network. Baek and Smith describe an architecture for prioritizing traffic from privileged applications [3]. Clients use trusted hardware to attest to the execution of an SELinux kernel equipped with a module that attaches Diffserv labels to outbound packets based on an administrator's network policy. Assayer uses a much smaller TCB (MiniVisor vs. SELinux) and does not require universal deployment to be effective.

Garfinkel et al. observe that trusted hardware could be used to mitigate network-based attacks [13]. However, their design of Terra [12] focused on setting up a secure VMM on the host, not on how to securely and efficiently convey host-based information to the network.

Feng and Schluessler propose, at a high-level, using Intel's Active Management Technology to provide information on the machine's state to network elements by introspecting on the main CPU's activities [11]. Unlike Assayer, they do not focus on conveying this information efficiently, nor do they provide a full system design and implementation.

Levin et al. [24] show that attesting to the value of a counter can be useful in a range of applications. This minimalistic approach works for some applications, but the limited counter functionality will exclude many applications that could be supported by an Assayer client module.

Dixon et al. propose pushing middle-box functionality, such as NAT and QoS to endhosts, using trusted computing as a foundation [9]. This is orthogonal, but potentially complementary, to Assayer's goal of conveying host-based information to network elements.

Conveying Specific Information to the Network. Ramachandran et al. propose imbuing packets with the provenance of the hosts and applications that generated them [27]. Combining this idea with Assayer (e.g., as an Assayer client module) would provide secure provenance data, even in networks with hostile elements or partial deployment.

Gummadi et al. propose the Not-A-Bot system [14] that tries to distinguish human traffic from bot traffic. They attest to a small client module that tags outgoing packets generated within one second of a keystroke or mouse click. However, the system only considers application-level attacks, i.e., the network is assumed to be uncongested. Thus, the server is responsible for verifying client attestations, which is less

practical for applications such as combating network-level DDoS attacks or super-spreader worms. The system works well for human-driven application-specific scenarios, but it is difficult to adapt it to services that are not primarily human-driven. For example, Assayer could protect NTP, transaction processing, network backup, or software update servers.

The recently proposed AIP architecture [1] assigns each host an IP address based on the hash of its public key. This provides an alternate mechanism to bind a key to a network sender. The AIP authors also propose equipping clients with "smart" network cards that will obey signed shut-off requests from servers. As we discussed in Section IV-B1, this functionality could easily be incorporated in MiniVisor by taking control of the client's network card, though we feel a cryptographic approach offers more resilient guarantees.

XI. Conclusion and Future Work

Many interesting and useful host-based properties are difficult or expensive to calculate external to the host, but simple to obtain on the host itself. We find that the growing ubiquity of trusted hardware on endhosts offers a powerful opportunity: a small amount of software on the endhosts can be trusted to provide useful information to receivers and network elements. Even local malware cannot interfere with the information provided. This approach enables fundamentally different network security mechanisms for confronting network attacks, such as spam, DDoS, and worms, since network filters can operate based on the host-supplied data. Developing richer client modules that offload network or server work onto the client offers a promising direction for future work. Additional work is also needed to determine how best to incorporate legacy and mobile devices into an Assayer-based network, perhaps based on secure proxies run by ISPs. Mobile devices should prove easier, since they already incorporate secure hardware (SIM cards).

References

[1] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *ACM SIGCOMM*, 2008.

[2] ARM. TrustZone security foundation. http://arm.com/TrustZone, Accessed Nov. 2009.

[3] K.-H. Baek and S. Smith. Preventing theft of quality of service on open platforms. In *The Workshop on Security and QoS in Communication Networks*, 2005.

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. of the ACM*, 13(7), 1970.

[5] K. Borders and A. Prakash. Web tap: Detecting covert web traffic. In *Proceedings of ACM CCS*, Oct. 2004.

[6] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of ACM CCS*, Oct. 2004.

[7] A. Brodsky and D. Brodsky. A distributed content-independent method for spam detection. In *Proceedings of the Workshop on Hot Topics in Understanding Botnets*, 2007.

[8] C. Dixon, T. Anderson, and A. Krishnamurthy. Phalanx: Withstanding multimillion-node botnets. In *Proceedings of USENIX NSDI*, 2008.

[9] C. Dixon, A. Krishnamurthy, and T. Anderson. An end to the middle. In *Hot Topics in Operating Systems*, 2009.

[10] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. 21(3):270–313, 2003.

[11] W. Feng and T. Schluessler. The case for network witnesses. In *Proceedings of Secure Network Protocols (NPSec)*, 2008.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, 2003.

[13] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS support and applications for trusted computing. In *Workshop on Hot Topics in Operating Systems*, May 2003.

[14] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-bot: Improving service availability in the face of botnet attacks. In *Proceedings of USENIX NSDI*, 2009.

[15] 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 *USENIX Security Symposium*, 2009.

[16] T. Hardjono and G. Kazmierczak. Overview of the TPM key management standard. TCG Presentations: https://www.trustedcomputinggroup.org/news/, Sept. 2008.

[17] N. Itoi, W. A. Arbaugh, S. J. Pollack, and D. M. Reeves. Personal secure booting. In *Australasion Conference on Information Security and Privacy*, 2001.

[18] B. Kauer. OSLO: Improving the security of Trusted Computing. In *USENIX Security Symposium*, 2007.

[19] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *ACM SIGCOMM*, 2002.

[20] A. Khorsi. An overview of content-based spam filtering techniques. *Informatica*, 31:269–277, 2007.

[21] E. Kohler. *The Click modular router*. PhD thesis, MIT, 2000.

[22] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. On the spam campaign trail. In *Proceedings of the Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[23] E. T. Krovetz. UMAC: Message authentication code using universal hashing. RFC 4418, Mar. 2006.

[24] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of USENIX NSDI*, 2009.

[25] S. C. Misra and V. C. Bhavsar. Relationships between selected software measures and latent bug-density. In *CCSIA*, 2003.

[26] B. Parno et al. Portcullis: Protecting connection setup from denial-of-capability attacks. *SIGCOMM*, 2007.

[27] A. Ramachandran, K. Bhandankar, M. B. Tariq, and N. Feamster. Packets with provenance. Technical Report GT-CS-08-02, Georgia Tech, 2008.

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

[29] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*, 2004.

[30] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8), Apr. 1999.

[31] J. Srage and J. Azema. M-Shield mobile security technology. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.

[32] K. Thompson, G. J. Miller, and R. Wilder. Wide-area Internet traffic patterns and characteristics. *IEEE Network*, 11, 1997.

[33] Trusted Computing Group. Trusted platform module main specification. Version 1.2, Revision 103, 2007.

[34] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *Proceedings of NDSS*, 2005.

[35] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *SIGCOMM*, 2006.

[36] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *USENIX Security Symposium*, 2004.

[37] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proceedings of the Computer Security Applications Conference*, 2002.

[38] M. M. Williamson. Design, implementation and test of an email virus throttle. In *Proceedings of the Computer Security Applications Conference*, 2003.

[39] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. *ACM SIGCOMM*, 2005.