Don't Talk to Zombies:

Mitigating DDoS Attacks via Attestation

Bryan Parno, Zongwei Zhou, Adrian Perrig

June 23, 2009

CMU-CyLab-09-009

CyLab

Carnegie Mellon University

Pittsburgh, PA 15213

Don't Talk to Zombies: Mitigating DDoS Attacks via Attestation

Bryan Parno, Zongwei Zhou, Adrian Perrig
Carnegie Mellon University

ABSTRACT

Distributed Denial-of-Service (DDoS) attacks typically originate from exploited endhosts controlled by a remote attacker. Current network-based DDoS defenses can only filter out malicious traffic based on the traffic's inherent properties; they cannot filter based on properties of the endhost that generated the traffic. We observe that the identity of the code that has generated a packet offers powerful predicates for filtering, and we develop a secure, general architecture, Assayer, for in-network filtering based on endhost properties.

Our proposed Assayer architecture leverages hardware-based attestation mechanisms to enable legitimate endhosts to embed secure proofs of code identity in packets. Receivers can specify traffic policies, which are enforced by on-path prioritizers. We design Assayer to achieve scalability, efficiency, and incremental deployability.

We implement and evaluate a basic Assayer prototype and find that the perceived application overhead, felt only during periods of significant network congestion, is less than 12%. Our simulations indicate that our architecture, even when deployed only at the victim's ISP, provides excellent protection against a botnet of 100,000 attacking hosts.

1. INTRODUCTION

Distributed Denial-of-Service (DDoS) attacks are an unfortunate reality on the Internet. Few of these attacks are launched by wealthy attackers who physically possess millions of machines. Instead, a typical attacker exploits software vulnerabilities in remote hosts, creating a "botnet" of tens or hundreds of thousands of machines [15, 24].

Previous proposals to combat DDoS attacks typically treat endhosts as black boxes [19, 27, 33–35]; i.e., they attempt to distinguish between attack traffic and legitimate traffic based solely on the traffic's inherent characteristics. Developing heuristics to make this distinction is difficult, and attackers respond by improving their bots' mimicry of real users [21]. Thus, the best the network can hope to do today is to offer equal service to all endhosts [27, 33], regardless of whether the endhost is acting legitimately or as part of a botnet.

To avoid the arms race in network-based DDoS-defense systems, we intend to provide the network with the powerful ability to identify, on a per-packet basis, the code on the endhost that generated or approved the packet. By enabling receivers to specify desirable properties a sender's code should provide, we enable the network to perform fine-grained receiver-controlled packet filtering.

Our primary contribution is the design and implementation of Assayer, a general architecture that allows a traffic recipient (e.g., a web server) to scalably filter traffic within the network based on properties of the sender. Thus, rather than ask adversaries to set an "evil" bit [5], Assayer enables legitimate hosts to set an unforgeable "good" bit that indicates to the network that the sender possesses some property desirable to the receiver. By prioritizing these packets, network elements ensure that during DDoS attacks, legitimate traffic will be more likely to reach the server. For example, a software-update server might give priority to traffic from the corresponding update agent, rather than a known attack script. From the client's perspective, the client proves that it possesses a server-approved property in exchange for elevated service from the network and server.

Assayer is agnostic to the policies receivers choose to employ. We leave it to each receiver to define what what properties a "good" sender should possess, and we provide the technical mechanisms to enforce these policies. While a receiver could insist on a specific, full-blown software stack, we show that it is much simpler, more secure, and more profitable for the receiver to ask senders to use small code modules that provide basic safety properties, such as limiting the rate or number of packets a particular client will generate.

To securely convey the identity of the sender's code to the receiver, we leverage attestation mechanisms offered by trusted computing hardware in commodity platforms. To circumvent Assayer's trusted computing mechanisms, an attacker must perform a local hardware attack on endhosts, making remotely-controlled bots far less useful for network attacks. Many commercial vendors already ship TPM-enabled platforms [14] (e.g., for use with Microsoft's Bitlocker), and Intel's upcoming chipsets are likely to integrate TPM functionality [16]. As we show in this paper, verifying attestations in the network would be too expensive. To achieve the properties mentioned above, we break up the attestation process; a distributed set of *verifier* hosts check sender attestations and issue tokens that on-path *prioritizer* devices can check efficiently.

Assayer is designed to be incrementally deployable; it provides significant properties even if only the victim's ISP deploys verifier and prioritizer nodes. Also, Assayer does not drop legacy packets unless network congestion occurs. Obviously, packet dropping during congestion happens even without Assayer; Assayer simply allows the network and the receiver to bias packet dropping to preserve packets that are more likely to be desirable to the receiver.

While Assayer could take many forms, we consider an instantiation to defend against DDoS attacks and have implemented a set of prototype components (including three possible sender configurations) to gain insight into the various performance bottlenecks. We also design, implement and evaluate symmetric and asymmetric authentication systems to show how various trade-offs can provide better performance or stronger guarantees. Our evaluation indicates that with judicious configuration, the overhead from Assayer is minimal, despite the strong properties it provides.

2. TRUSTED COMPUTING BACKGROUND

Many commodity computers sold today come equipped with a Trusted Platform Module (TPM) chip [31]. Over 200 million TPMs have already been deployed [14], and Intel plans to include an integrated TPM in future chipsets [16]. The TPM is passive; it does not actively monitor the platform nor can it actively prevent software from running. Instead, it is designed to summarize the software state of a platform in a series of Platform Configuration Registers (PCRs) that form an append-only record and cannot be reset without rebooting the platform. The TPM can be used to digitally sign these values for a third party via an *attestation*, or it can bind secrets to a software configuration via *sealed storage*.

Measurement. When the platform first boots, platform hardware takes a measurement (a SHA-1 hash) of the BIOS and records the measurement in a 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 it can be used to attest to a tiny piece of security-sensitive software [23].

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 ensures 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 [12] (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 [31] includes a provision for using group signatures to generate attestations, but as of yet, we are not aware of any TPMs that implement this functionality.

Sealed Storage. Finally, the TPM can bind data to a particular platform configuration using a technique called sealed storage. Essentially, software can request that the TPM seal a blob of binary data to a particular set of PCR values. In the future, the TPM will only unseal 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.

Late Launch. Recent CPUs from AMD and Intel support a new instruction that performs a late launch operation, which takes a region of code as an argument. During a late launch, the CPU atomically resets its state, enables hardware protections, measures the code region, records the measurement in the TPM, records the fact that a late launch took place, and finally begins to execute the code region. This allows a TPM-equipped platform to attest to less software, since any software executed before the late launch operation can be securely excluded from the attestation [18, 23].

3. PROBLEM DEFINITION

Even during a DDoS attack, a legitimate client should obtain service with high probability after a short time delay. We do not attempt to solve congestion problems created by too many legitimate clients visiting the server.

3.1 Assayer Goals

We aim to design an architecture to allow servers to specify preferred endhost properties that can be efficiently verified on a per-packet basis *within the network*, before congestion can occur. Such a scheme requires key properties.

Unforgeable. Malicious endhosts or network elements should be unable to claim benign properties they do not possess.

Stateless-In-Network Processing. To ensure the scalability of our new prioritizing middleboxes, we aim to avoid keeping per-host or per-flow state on these middleboxes. If per-flow state becomes feasible, we can use it to cache authentication information currently carried in packets.

Privacy preserving. We aim to leak no more user information than is already leaked in present systems. In other words, we do not aim to protect the privacy of a user who visits a website and enters personal information.

Incrementally Deployable. While we believe that Assayer would be useful in future networks, we strive for a system that can bring immediate benefit to those who deploy it.

Efficient. To be adopted, Assayer must not unduly degrade client-server network performance. Furthermore, to prevent DDoS attacks on Assayer's components, they must be capable of acting efficiently.

3.2 Assumptions

We assume that our trusted software and hardware components behave correctly. To increase the accuracy of this assumption, we aim to minimize the size and complexity of our trusted components, since software vulnerabilities are correlated with code size [26], and smaller code is amenable to formal analysis. We assume the verifiers and clients can perform hardware-based attestations; in this work, we focus on TCG-based attestations, since TPMs are becoming ubiquitous in commodity PCs [14]. We also assume that networked devices can be loosely time synchronized (e.g., within 10 seconds); this is readily achievable [25]. Finally, we make the standard TCG assumption [31] that TPM-based protections can only be violated with local, sophisticated hardware attacks. We assume remote attackers cannot induce users to perform physical attacks on their own hardware.

4. THE ASSAYER ARCHITECTURE

Below, we first give an overview of how Assayer's components interact. We then discuss the high-level design of each component (Section 4.2), followed by the concrete protocol details (Section 4.3). We also show how the protocol can be implemented using symmetric cryptographic primitives for high efficiency, or asymmetric primitives for additional security properties. Finally, in Section 4.4, we show how to preserve user privacy while maintaining a server's ability to revoke misbehaving clients.

4.1 Overview

Ideally, to convey endhost properties to the network, we would simply have the client include an attestation of its "good" properties in each packet it sends. Forwarding nodes in the network could check these attestations and give priority to packets with valid proofs. Unfortunately, checking attestations is time consuming, and requires interaction with the client (recall Section 2 – the verifier must send the client a nonce to ensure the attestation's "freshness"). Similarly, if the server is prone to DDoS attacks, it will not have the extra capacity to perform these verifications. Indeed, if the server is subject to a network-level DDoS, legitimate packets may never reach the server in the first place.

To address these issues, Assayer lets the server delegate the task of verifying clients to one or more distributed *verifiers* (Figure 1) operated by or trusted by the server's operator. The server has the complex job of serving application-specific content, and thus is difficult to protect from DDoS attacks, whereas the verifiers, as we show in Section 4.2.2, are much easier to protect. To ensure the integrity of these verifiers, the server requires each verifier to periodically prove its correctness via an attestation. The server then responds by provisioning the verifier with a limited-duration verifier token that is bound to the verifier's platform and code. The server also provides the verifier with a server-specific policy defining properties of "good" clients and assigning relative priorities to those properties.

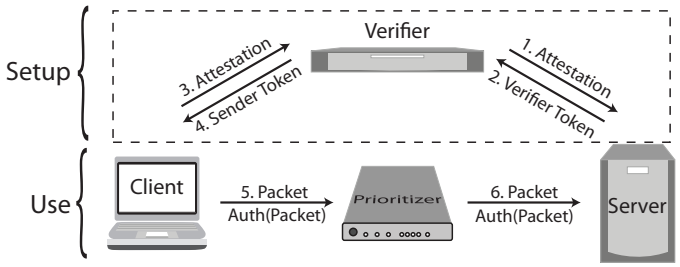

Figure 1: System Components. ***Periodically (e.g., once a week), each verifier generates an attestation for the server. The server checks that the verifier is using approved software and issues a Verifier Token. The token allows the Verifier to issue Sender Tokens after it has verified a client's attestation. The client can use the Sender Token to authorize its packets. The Prioritizer verifies the client's authorization and grants the client's traffic priority over other traffic destined to the same server.***

Periodically, a Assayer-enabled client attests to the verifier that it adheres to a server-defined "good" property. If the attestation is correct, the verifier uses its verifier token to issue a sender token. The token's validity is bound to the client's platform such that it only remains valid while the platform remains in a "good" state. Its validity is also bounded in time based on server policy, but we expect typical durations might be on the order of a day or a week. During that time, client may use the sender token to authorize all of the packets it sends to the server, or it may use authorized packets only in response to network congestion. These packets can be efficiently checked by on-path, server-deployed *prioritizer* middleboxes, which give authorized packets elevated priority as they travel to the server. This approach provides generic, network-level verification of server-defined client properties.

4.2 Assayer Components

Below, we present the high-level design of each Assayer component, saving the protocol details for Section 4.3.

4.2.1 Clients

At its core, Assayer uses a hardware-based *attestation* from the client to securely describe the client's software to the verifier. *Sealed storage* binds the client's token to the attested software state. Any change in the client's security-critical software configuration will make the sender token inaccessible.

Traditionally, software attestations have included every piece of code executed since boot [29]. However, this approach is undesirable for numerous reasons. It is not scalable, since the verifier is forced to track the versions and configurations of hundreds of pieces of software. It also offers dubious security, since it is difficult to determine all of the possible security interactions between these executables. Finally, this approach reveals considerable information about the client's platform, harming user privacy.

Instead, we propose to have the client attest to only two items: (1) the presence of a module that acts as a reference monitor for authorized packets, and (2) adequate protections (i.e., secrecy and execution integrity) for that module. Untrusted code can submit a packet to a module in order to obtain a limited validity (e.g., valid for 10 seconds) authorization for that packet.

The Assayer approach offers multiple advantages. The client attests to a much smaller set of software (potentially fewer than 1,000 lines of code – see Section 6). Smaller code generally contains fewer bugs and is more amendable to formal analysis [26]. It is also easier to standardize on a handful of reference monitors than to standardize on a handful of complete software stacks. Generic modules allow clients a much wider selection of software; conversely, they allow servers to support a more extensive clientele.

Assayer's modular approach also makes it easy for a client to communicate with multiple servers without having to reboot or otherwise reconfigure its software. As long as the servers agree that the client's module protection meets (or exceeds) their requirements, each server can require a separate module. For example, one server can require a module that limits the number of authorized packets generated, while another server can require a module that only authorizes packets containing HTTP requests.

Below, we describe protection options and generic, modular approaches for obtaining useful client properties.

Module Protection.

For the modules described below to provide secure information about the client, they must be protected from untrusted code. Furthermore, the client must convince the verifier that these protections are in place.

Fortunately, these protections are relatively easy to provide, since the modules have little interaction with other software on the system. For example, the modules do not need to maintain exclusive control over the client's network card, since they provide a positive effect (improve the client's packets' priority in the network) on outgoing packets. Thus, untrusted code simply provides the module with the contents of outgoing packets. If the module decides to authorize a packet, it returns the appropriate authorization data. If untrusted code does not invoke the module, outgoing packets simply will not contain authorization tokens, and hence will not receive priority.

Thus, to provide the required module protection, servers may utilize the isolation provided by an operating system, by a hypervisor, or by the hardware itself. Ultimately, it is up to the server to decide (and inform the verifier) what level of protection it desires. For example, some servers may decide that as long as the modules execute within the user's kernel, they will be sufficiently protected from attacks. Alternately, the server may insist that the client use a virtual machine monitor (VMM), with client applications running in one VM, and the module running in a separate VM.

Client Modules.

In theory, a client module can base its packet authorizations on almost any arbitrary endhost property. For example, the module might issue an authorization if a packet was generated by a program run by a specific user known to the server, or if it does not contain "sensitive" data. The module can also convey additional properties about the program that generated the packets. For instance, by applying control-flow enforcement techniques [10], the module can guarantee that outgoing packets originated from a program that never deviated from its expected control flow.

Nonetheless, to combat DDoS attacks, we believe the following two client modules offer a useful combination of protection and simplicity. Each module can be standardized and integrated into endhost software (e.g., as a standard OS component). They are also parameterizable, so each server can customize the guarantees it expects. These values can be provided to the client when it contacts the verifier.

A Limiter Module. A limiter imposes a simple, coarse-grained rate-limit on authorized packets. Its interface accepts a packet and returns an authorization token or a rejection. For example, the limiter might be guaranteed to authorize a limited number of packets or to authorize packets at a limited rate. The exact policy can be specified by the verifier at the same time it checks to see that an appropriate Limiter is in place. The authorization contains a unique nonce and a timestamp that allows prioritizers to perform duplicate detection (see Section 4.3 for details). While offering only a simple level of protection, the Limiter suffices to mitigate many network-level DDoS attacks.

An Attributor Module. An Attributor tracks the origin of packets and only authorizes packets that originate from approved code sources. The definition of approved sources is specified by the server's policy and enforced at the verifier. The verifier must also ensure that the attributor can accurately and securely perform the attribution.

For example, the server might prefer to entrust the attribution task to a Virtual Machine Monitor (VMM), e.g., Xen or VMware. The VMM would authorize packets originating from an approved virtual machine, while the user could run arbitrary programs in a second VM. The client would then attest to the verifier that it had installed an approved VMM capable of enforcing this attribution policy. The attestation would only include the VMM and its policy (and not all applications in the user's VM), since the VMM is trusted to maintain its isolation from applications.

Attributors are likely to be more complex that Limiters, but they offer much more detailed and nuanced information. Rather than simply limit the flow of packets to a fixed rate, they can guarantee that packets are generated by a particular trusted application. This can, for example, allow Intrusion Detection Systems (IDSs) to perform more accurate profiling, since the traffic from a host is already identified as coming from a specific application (e.g., Firefox 3.0.1) rather than simply a class of applications (e.g., a web client).

In addition, much of the information needed for attributing packets may already be present in the protection mechanisms needed to isolate these modules. For example, if the modules are implemented as kernel modules, then the server already trusts the kernel to protect the modules, and hence it can also trust it to provide attribution data without significantly increasing the trusted computing base.

4.2.2 Verifiers

Verifiers are responsible for checking that clients possess server-specified properties and issuing sender tokens. The primary challenge in designing verifiers is to ensure that they do not become a DDoS vulnerability. Fortunately, this is much easier than it is for servers. Servers must constantly serve proprietary, potentially difficult to replicate content using complex software stacks. In contrast, the verifiers perform a single, generic task using a tiny software stack, making them easy to replicate. Since clients only obtain sender tokens infrequently, an occasional outage among a few verifiers can be tolerated. A DDoS attacker would need to flood many verifiers over an extended time (e.g., a week) to prevent clients from obtaining tokens.

To enhance verification robustness, we envision the server employing numerous verifier machines distributed around the Internet. Some servers might choose to contract with various ISPs to host simple server-administered verifier machines (as many web content firms do today). Others might outsource verification to a third party, such as Akamai. To reduce costs, a coalition of associated organizations, such a group of universities might create a federation of verifiers. Each organization would host a single verifier that would act as a verifier for all of organizations in the federation.

In each of these scenarios, the verifier operates outside of the server's direct administrative domain, so we can again use attestation to improve security. Periodically (e.g., once a week), the server can request an attestation from the verifier. Assuming the attestation is correct, the server can issue a limited-duration verifier token bound to the correct verifier software configuration. The server also provides the verifier with the server's policy file, specifying the properties the verifier should expect from clients, and the relative priorities those properties should be given by the prioritizers. The limited duration of verifier tokens bounds the length of the revocation list that must be maintained to track misbehaving verifiers.

Any distributed and well-provisioned set of servers could enable clients to locate the verifiers for a given server. 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 record 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. While the DNS servers may themselves become victims of DDoS attacks, the relatively static listing of verifier machines is still much easier to replicate and serve than an arbitrary server's content. Furthermore, if Assayer becomes ubiquitous, the global top-level domain (gTLD) servers could be extended to store a verifier record (in addition to the standard name server record) for each domain. The gTLD servers are already well-provisioned, since a successful attack on them would make many services unavailable.

4.2.3 Prioritizers

To combat network-level DDoS attacks, we need to prioritize approved packets as early as possible, before they reach the server's bottleneck. The prioritizers must also be able to verify packet authorizations efficiently to prevent the prioritizers themselves from becoming bottlenecks. Finally, the prioritizers must suppress duplicated authorized packets. Section 7 shows that prioritizers can perform all of these duties at reasonable speeds.

We envision prioritizer deployment occurring in phases (Figure 2), dictated by the server operator's needs and business relationships. Initially, to combat application-level attacks, the server' operator may simply deploy a single prioritizer 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 prioritizers 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, prioritizers will likely become standardized, shared infrastructure that is deployed ubiquitously. However, as we show in Section 7.4, partial deployment can provide significant protection from attacks.

To avoid hurting flows destined to other servers, prioritizers give preference to server-approved packets *relative* to other packets destined to that same server. This could be implemented via per-destination fair-queueing, with the Assayer-enabled server's queue configured to give priority to approved packets.

To enable prioritizers to perform duplicate detection, the authorization modules on the client include a unique nonce and a timestamp in each packet authorization. These authorizations only remain valid for a limited duration (e.g., less than 10 seconds), so each prioritizer maintains a limited duration Bloom filter to check for duplicates. Section 5 shows how this approach defeats replay attacks.

4.2.4 Server

The server must arrange for the appropriate deployment of verifiers and prioritizers. It must also periodically verify the correctness of its verifiers and issue fresh verifier tokens. If it detects that a verifier is misbehaving, the server can refuse to renew its token. In Section 4.3, we also describe additional provisions to allow the server to actively revoke rogue verifiers. Finally, the server is responsible for conveying its policy preferences to the verifier. This policy describes the

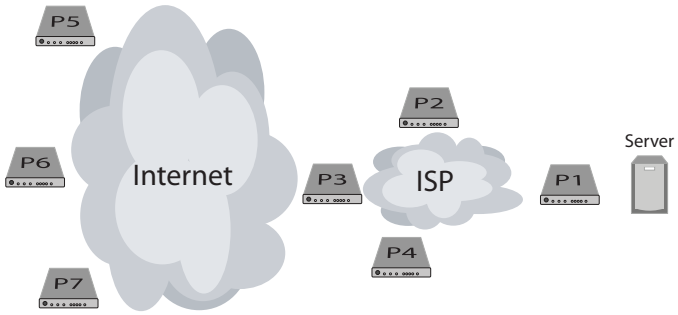

Figure 2: Prioritizer Deployment. ***Initially, a server is likely to deploy a single prioritizer (P1) on its access link. For a fee, an ISP may decide to deploy prioritizers (P2-P4) at its access links. The server may also be able to leverage prioritizers scattered around the Internet (P5-P7) if it has business relations with these entities.***

client configurations the server prefers, as well as the relative priorities these configurations should receive.

4.3 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 authorize their outbound packets. 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 7, we quantify the performance trade-offs entailed.

4.3.1 Desirable Properties

1. **Limited Token Validity.** Verifier Tokens are only valid for a limited time period and are accessible only to valid verifier software. Sender Tokens are only valid for a limited time period and are accessible only to valid client software.
2. **Verifier Accountability.** Verifiers should be held accountable for the clients they approve. Thus one verifier should not be able to generate tokens that appear to originate from another verifier.
3. **Scalability in Prioritizer Count.** The verifier's work, as well as the size of the Sender Token, should be independent of the number of prioritizers.
4. **Topology Independence.** Neither the verifier nor the sender should need to know which prioritizer(s) will see the client's packets. As Figure 2 illustrates, there may be more than one prioritizer on the path to the server, and the number may change over time. Thus, the sender's token must be valid at any prioritizer. We feel the benefits of this approach outweigh the potential for an adversary to use a single sender token on multiple disparate paths to the server. Such duplicated packets will be detected when the paths converge.
5. **Prioritizer Independence.** A prioritizer should not be able to generate Sender Tokens that will be accepted at other prioritizers. This prevents one rogue prioritizer from subverting other prioritizers.
6. **Client and Prioritizer Accountability.** The server should be able to distinguish between traffic generated by a malicious client and that generated by a malicious prioritizer. Otherwise, a rogue prioritizer can impersonate a sender.

V	Knows K_S
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.
$S \rightarrow V$	Attestation request and a random nonce n
$V \rightarrow S$	K_V, TPM_Quote = $PCRs, Sign_{K_{AIK}^{-1}}(PCRs\|\|n)$, C_{AIK}
S	Check cert, sig, n, $PCRs$ represent $Code_V$ and K_V
$S \rightarrow V$	$Policy, Sign_{K_S^{-1}}(Policy)$
$S \xrightarrow{*} P_i$	$K_V, Sign_{K_S^{-1}}(K_V)$

Table 1: Verifier Attestation. ***V is the verifier, S the server, P_i the prioritizers, and C_{AIK} is a certificate for the verifier's AIK. Section 2 has additional background on attestation.***

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 = $PCRs, Sign_{K_{AIK}^{-1}}(PCRs\|\|n)$, C_{AIK}
V	Check cert, sig, n, $PCRs$ represent $Code_C$ and K_C
$V \rightarrow C$	$Token_C = [ID_V, K_C, prio, time, H(C_{AIK}),$ $Sign_{K_V^{-1}}(V\|\|K_C\|\|prio\|\|time\|\|H(C_{AIK}))]$

Table 2: Client Attestation. ***C is the client, $Code_C$ is a module from Section 4.2.1 combined with appropriate protections, V is the verifier, $prio$ is the client configuration's priority (as specified by the server), $time$ is a timestamp, and H is a cryptographic hash function.***

$C \rightarrow Code_C$	Packet contents p.
$Code_C$	Verifies p satisfies module's requirements.
$Code_C$	Generates a random nonce m.
$Code_C \rightarrow C$	$Auth_C = (m, time, Sign_{K_C^{-1}}(p\|\|m\|\|time))$
$C \rightarrow S$	$p, Token_C, Auth_C$

Table 3: Packet Authorization. ***C is the client, $Code_C$ is a module from Section 4.2.1 combined with appropriate protections, and S is the server. The client sends the packet to the server, but it will be processed along the way by one or more prioritizers.***

4.3.2 Protocol Specifications

Below, we describe each interaction shown in Figure 1.

Verifier Attestation. Before giving a verifier the power to prioritize client packets, the server must ascertain that the verifier is in a correct, trusted state (Table 1). It does so via an attestation (see Section 2 for details on attestation).

Algorithm 1 *Processing of packet p at the prioritizer.*

```
1: if p contains Token_C, Auth_C then
2:     (ID_V, K_C, prio, time, H, Sig_V) ← Token_C
3:     Verify Sig_V using K_V.
4:     Use time to check that Token_C has not expired.
5:     (m, timestamp, Sig_C) ← Auth_C
6:     Verify Sig_C using K_C.
7:     Check that timestamp ∈ now ± ε.
8:     Check that pair (K_C, m) is unique in last 2ε time intervals.
9:     Insert (K_C, m) into Bloom Filter.
10:    if All verifications succeed then
11:        p receives priority prio
12:    else
13:        Drop p
14: else
15:    p receives normal priority
```

The attestation convinces the server that the verifier is running trusted code, that only the trusted code has access to the verifier's private key, and that the private key in question is freshly generated. Since the verifier token's validity is limited, the server periodically rechecks the verifier's correctness by rerunning the attestation protocol.

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 6), 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. Thus, any change in the verifier's software will make the private key inaccessible. Finally, the verifier code records a measurement of its new public key in a PCR.

When the server requests an attestation from the verifier, it provides the verifier with a fresh nonce. The verifier uses the TPM to generate a *quote* – essentially a signature over the nonce and the state of the TPM. The verifier replies to the server with its new public key, the quote, and a certificate proving the quote originated with a real TPM. Assuming all of these items verify correctly, the server provides the verifier with its signed policy file. Rather than give the verifier an explicit token, the server informs its prioritizers that the verifier's new public key should be accepted when prioritizing packets. Since the prioritizer is run by (or acts on behalf of) the server, it can be configured with the server'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 (Table 2). Trusted code on the client generates a keypair and attests to the verifier that the private key is bound to the trusted software and was generated recently. If the client's attestation verifies correctly, the server checks to see that the client's trusted software is acceptable, based on the server-provided policy. If it is, the verifier returns a token consisting of the verifier's ID, the client's public key, the server-assigned priority, a timestamp, and the verifier's signature.

Packet Authorization. To authorize a packet, untrusted code on the client asks the trusted module to produce an authorization (Table 3). The untrusted code passes the packet's contents to the trusted code module. The code module verifies that the packet satisfies the module's property. For example, a limiter module checks that it has not already sent too many packets. Finally, the module produces an authorization that consists of a unique nonce, a timestamp, and the client's signature. Untrusted code can then add the client's token and authorization to the packet and send it to the server.

Packet Prioritization. Prioritizers along an approved packet's path process such packets using Algorithm 1. The prioritizer uses the verifier's ID to lookup the corresponding public key provided by the server. It uses the key to verify the authenticity and freshness of the client's token. The prioritizer may optionally decide to cache these results to speed future processing. It then checks the authenticity, freshness, and uniqueness of the packet's authorization. It stores a record of the packet's nonce for a short time (e.g., 4 seconds) to prevent duplication and gives the packet the specified priority if it passes all verification checks. However, if a packet's verification checks fail, the prioritizer drops the packet. Legitimately generated packets will only fail to verify if an on-path adversary modifies the packets. Such an adversary can also drop or alter the packets, so dropping the packets does not increase the adversary's ability to harm the sender.

4.3.3 A Symmetric Alternative

As we show in Section 5, the protocols shown above possess all of the properties described in Section 4.3.1. Unfortunately, they require the client to compute a public-key signature for each packet sent and the prioritizer to verify two public-key signatures per packet. The challenge is to improve the efficiency of the scheme while retaining as many of the properties from Section 4.3.1 as possible. Below, we show how to alter the protocols to use more efficient symmetric primitives.

Verifier Attestation. The last step of the protocol shown in Table 1 is the only one that changes. Instead of sending the verifier's public key to all of the prioritizers, the server generates a new symmetric key K_{VP}. The server encrypts the key using the verifier's newly generated public key and sends the verifier the resulting ciphertext ($Encrypt_{K_V}(K_{VP})$). Since the corresponding private key is sealed to the verifier's trusted code, the server guarantees that only the trusted code can obtain the symmetric key. The server also encrypts the key and sends it to each of the prioritizers.

Client Attestation. The protocol shown in Table 2 remains the same, except for one minor change and one major change. First, when the client sends its token request, it includes a randomly chosen client identifier ID_C. The biggest difference is in the token returned by the verifier.

To compute the new token, the verifier first computes a symmetric key that the client will use to authorize packets:

$$K_{CP} = PRF_{K_{VP}}(V||ID_C||prio||time), \qquad (1)$$

where *PRF* is a secure pseudo-random function. The verifier then sends the client: $Encrypt_{K_C}(K_{CP})$, $Token = (V, ID_C, prio, time)$. Again, the attestation convinces the verifier that K_C^{-1} is bound to trusted code, so it knows that only trusted code can obtain K_{CP}. Furthermore, without knowing K_{VP}, no one can produce K_{CP}.

Packet Authorization. Packet authorization is the same as before, except that instead of producing a signature over the packet contents, the code module produces a Message Authentication Code (MAC) using K_{CP}, an operation that is orders of magnitude faster.

Packet Prioritization. The packet prioritization algorithm remains similar. Instead of checking the verifier's signature, the prioritizer regenerates K_{CP} using Equation 1 and its knowledge of K_{VP}. Instead of verifying the client's signature on the packet, the prioritizer uses K_{CP} to verify the MAC. As a result, instead of verifying two public key signatures, the prioritizer 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 4.3.1, but it does not provide properties 5 and 6. Since each verifier shares a single symmetric key with all prioritizers, a rogue prioritizer can convince other prioritizers to elevate traffic incorrectly. We could prevent this attack by having the server establish a unique key for each verifier-prioritizer pair, but this would violate another property. Either the verifier would have to MAC the packet using all of the keys it shares with the prioritizers (violating the fourth property), or the verifier would have to guess which prioritizers would see the client's packet, violating our topology independence property.

Similarly, since the client and the prioritizer share a symmetric key, the server cannot distinguish between malicious prioritizers and malicious clients. Nonetheless, since the server's operator directly or indirectly controls the prioritizers, such risks should be acceptable to many servers, given the dramatic performance benefits offered by the symmetric scheme.

4.4 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 server to revoke misbehaving clients. Assayer achieves this using structured AIK certificates.

Recall from Section 2 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 server. Using a different AIK for each server prevents the servers from tracking the client across sites. However, similar to a DNS lookup, this approach allows the Privacy CA to learn that *some* client intends to visit a particular server. Fortunately, since the EK contains no user-specific information, the Privacy CA only learns that some TPM-enabled machine intends to visit a particular server.

To preserve user privacy across sessions with a single server, the client can generate a new AIK and request a new certificate from the Privacy CA. However, we require Privacy CAs to only simultaneously issue one AIK certificate per server 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 server until the first certificate expires. This prevents a client from generating multiple *simultaneous* identities for communicating with a particular server.

Since each client token contains a hash of the client's AIK certificate, if the server decides a client is misbehaving, it can provide the hash to the Privacy CA and request that the Privacy CA cease providing AIK certificates to the EK associated with that particular AIK. Similarly, the server can instruct its verifiers and prioritizers to cease accepting attestations and packets from that AIK.

5. POTENTIAL ATTACKS

In this section, we analyze potential attacks and show how Assayer's design defends against them.

5.1 Exploited Endhosts

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

Code Exploits. An adversary that finds an exploit in trusted code can violate Assayer's security, potentially sending unlimited approved packets from exploited endhosts. This supports our argument that servers should trust small code modules instead of large software stacks. Exploits of untrusted code are less problematic. The server trusts the attested client code to protect the client code module, and it trusts the code module to accurately perform its duties (whether they be limiting packet authorizations, or attributing packets to a particular piece of code). Thus, the trusted module will continue to function, regardless of how the adversary exploits the untrusted code.

Flooding Attacks. Since an attacker cannot subvert the authorized packet guarantees, she might instead choose to use an exploited machine to simply flood the server with legacy packets; however, the impact of these packets floods will be mitigated, since all authorized packets will have priority over these legacy packets (Section 7.4 illustrates this). As we explained in Section 4.2.2, the verifiers are designed to withstand DDoS attacks, so flooding them will be unproductive. Finally, since prioritizers check packets at line speed, flooding the prioritizers (with missing, invalid, or even valid authorizations) will not hurt legitimate traffic throughput.

Authorized Packet Duplication. Since the trusted code does not maintain exclusive control of the network interface, the exploit code could ask the trusted module to authorize a packet and then repeatedly send the same packet, either along the same network path or across diverse network paths. Because each authorized packet contains a unique nonce, duplicate packets on the same path will be dropped at the first prioritizer. Similarly, duplicates sent down different paths will be dropped as soon as the paths converge at a prioritizer. Duplicates are more likely to cause congestion close to the victim (since those links tend to be smaller), but paths also tend to converge close to the victim, minimizing the advantage of sending duplicate packets. Thus, a duplicated packet can only be used to attack a particular link once.

5.2 Malicious Endhosts

Beyond the attacks discussed above, an attacker can potentially use hardware-based attacks to subvert the trusted 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, essentially convincing the verifier that the adversary's machine is running trusted 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 their sender tokens. As discussed in Section 4.4, at any moment, each TPM key corresponds to exactly one AIK for a given server. Furthermore, to obtain a sender token from the verifier, the attacker must commit to a specific server-approved property. If the attacker's traffic violates the property, it can be detected, and the attacker's TPM key will be revoked. For example, if the attacker claims to have a limiter module that only permits X packets to be sent, and the server detects that the attacker has sent more than X packets, then the server knows that the client is misbehaving and will revoke its TPM key (see Section 4.4). This key can only be replaced by purchasing a new TPM-equipped machine, making this an expensive and unsustainable attack.

5.3 Rogue Verifiers

A rogue verifier can authorize arbitrary prioritization of arbitrary traffic. However, the verifier's relatively simple task makes its code small and easy to analyze. The attestation protocol shown in Table 1 guarantees that the server only approves verifiers running the correct code. Since verifiers are owned by the server's operator or by someone with whom the operator has a contractual relationship, local hardware exploits should not be a concern.

Furthermore, since verifiers cannot imitate each other (even in the symmetric authentication scheme), a server that detects unusual or incorrect traffic coming from clients approved by a verifier can revoke that verifier. Revocation can be performed by refusing to renew the verifier's token, or by actively informing the prioritizers that they should discard the rogue verifier's public key.

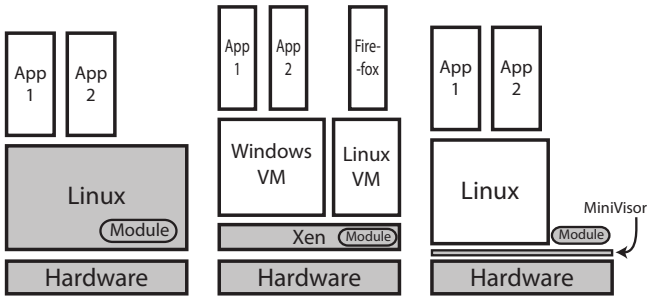

Figure 3: Client Configuration. ***Three client configurations we implemented, based on Linux, Xen, and MiniVisor. Shading indicates the components that must be trusted.***

5.4 Rogue Prioritizers

A rogue prioritizer can discard prioritized packets, or give priority to attack traffic. However, since it sits on the path from the client to the server, a rogue prioritizer can already drop or alter packets arbitrarily. In the asymmetric scheme (Section 4.3.2), a rogue prioritizer cannot convince correct prioritizers to elevate attack traffic, since it cannot generate correct verifier signatures necessary for the sender tokens. Similarly, a rogue prioritizer cannot frame a client, since it cannot sign packets using the client's private key. The symmetric scheme trades off these properties in favor of greater efficiency. Fortunately, since the prioritizers are directly administered by the server's operator and perform a relatively simple task, rogue prioritizers should be rare.

6. 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 implemented three types of client configurations that provide increasing levels of protection for the client modules. The clients create attestations that our checked by our verifier prototype. Finally, we have implemented a basic prioritizer using the Click router [20]. We present our performance results in Section 7, as well as our Internet-scale simulations.

6.1 Client Modules

We evaluated a number of possible client configurations that offer varying degrees of module protection (see Figure 3). We implemented one client that uses Linux to protect the client module, one that uses the Xen Virtual Machine Monitor (VMM) [4], and one that uses a tiny hypervisor called MiniVisor that we developed using hardware-virtualization support. We also evaluated using the Flicker

architecture [23] to protect the client module, but we found that it increased packet transmission time by 1-2 orders of magnitude, and hence we decided it is not yet practical for performance critical applications. Our three client implementations allow the server to choose between a TCB of millions (Linux), thousands (Xen), or hundreds (MiniVisor) of lines of code. MiniVisor is a particularly attractive choice, since (as we show in Section 7), it offers excellent performance while adding only 841 lines of code to the TCB.

With all three systems, we employ a late launch operation (recall Section 2) to simplify client attestations by removing the early boot code (e.g., the BIOS and bootloader) from the set of trusted code. Thus, our attestations consist of the protection layer (Linux, Xen, or MiniVisor), a client module, and the fact that the protection layer is configured to properly isolate the module.

All three systems implement both the asymmetric and the symmetric protocols described in Section 4.3, to allow us to evaluate the performance trade-offs between the two. Since both schemes add bytes to outgoing packets, they could potentially cause considerable packet fragmentation. To prevent this, we reduce the MTU on the interface facing untrusted code by the same number of bytes that our module adds. Of course, untrusted code can increase the MTU, but that will merely hurt performance, not security.

All three systems use Linux's TUN/TAP interface[1] to reroute outbound packets to a user-space program. The user-space program hands the packet contents to a trusted module that decides whether or not to authorize the packet and then routes the packet back to the physical interface. This configuration simplified development, but is it less than optimal from a performance standpoint, since packets are passed across the user-kernel space divide multiple times. Intercepting packets inside of a network driver or kernel module would improve our performance.

In Linux, our code module restricts the rate of packets sent to a particular destination IP address by refusing to authorize packets any faster than the specified rate. In our Xen configuration, the client runs two VMs. One VM runs Windows, and the user can install arbitrary programs in that VM. The second VM runs Firefox on top of a minimal Linux installation. We implement an attributor module that only approves outbound packets from the Linux VM.

With MiniVisor, we implement a simple limiter module that will only approve a fixed number of packets. When MiniVisor is late launched, it uses shadow page tables to isolate its own private memory area and then boots the Linux kernel. Untrusted code running on Linux can request an authorization by invoking MiniVisor's single hypercall. MiniVisor hands the packet contents to the module, which checks that it has not exceed its approval limit, approves the packet provided, and then increments its count of approved packets. MiniVisor then returns to the untrusted code, providing it with the packet authorization data.

[1] http://vtun.sourceforge.net/tun/

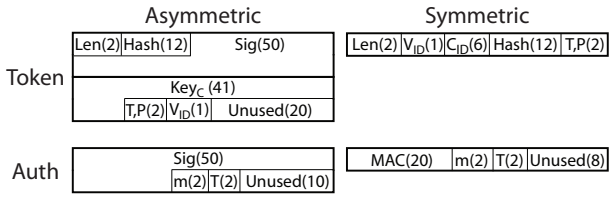

Figure 4: Packet Layout. ***Byte-level layout for sender tokens and packet authorizations. The two are shown separately for clarity, but in practice, would be packed together.*** T ***is a timestamp,*** P ***is the packet's priority, and*** m ***is a randomly-chosen nonce.***

6.2 Client Verification

All of the client modules described above 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 Table 2. Since the code that allows the server to check verifier attestations (Table 1) 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 [12]. 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 server-approved software. 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)[2], 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.

6.3 Packet Authorization

As discussed in Section 4.3, we can generate tokens using either asymmetric or symmetric cryptographic primitives. We implemented both systems to evaluate their relative performance. Figure 4 illustrates the low-level layout of the tokens and authorizations for each scheme.

[2] http://code.google.com/p/spprocpool/

With both schemes, we add the token and the authorization to the packet payload itself, and then adjust the appropriate header fields (length, checksum, etc.). This provides compatibility with legacy network devices. The server must strip this information out before handing the packet contents to its applications, but this was quite simple to implement.

With the asymmetric scheme, we use elliptic curve cryptography to minimize the size of the client's public key, since it is included in the client's sender token and hence requires space in every packet. We use the secp160k1 curve, which provides approximately 80 bits of cryptographic strength. This should provide more than enough strength for the short lifetime (e.g., a day or a week) of client keys. The verifier uses the elliptic curve version of the digital signature algorithm (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 requires 108 bytes, and the client's authorization requires 54 bytes.

With the symmetric scheme, if we use a 160-bit key with SHA1-HMAC (which remains secure, despite recent collision attacks on SHA1), then the client's token only requires 23 bytes, and the authorization requires 24 bytes.

6.4 Prioritizer

We implement the prioritizer on top of the Click router [20]. We designed a custom Click element that examines all packets destined for a particular destination IP address. If a packet contains a Assayer flag in the header, we check the token and the authorization data, following Algorithm 1. If all checks succeed, the packet is added to a priority queue.

With the asymmetric scheme, the prioritizer needs to verify both the verifier's ECDSA signature in the client's token and the client's ECDSA signature in the packet authorization. With the symmetric scheme, the prioritizer needs to verify the client's SHA1-HMAC in the packet authorization.

To detect duplicate packets, we use a Bloom Filter [6]. We only insert a packet into the Bloom Filter after verifying the sender token and the packet authorization. The Bloom Filter ensures that a valid packet is unique in a given time period t with a bounded false positive probability γ.

To illustrate this, suppose that the prioritizer has a 1Gbps inbound link, and the time period t is 1 second. In the worst case, the prioritizer would receive n packets/sec, where n is the link's capacity divided by the minimum packet size, and all of these packets carry correct tokens and valid packet nonces. In the asymmetric scheme, the minimum TCP packet size is 214 bytes, so $n = 584,112$ packets/second. As for symmetric scheme, the minimum TCP packet size is 99 bytes, so $n = 1,262,626$ packets/second. If we use k different hash functions, and n different packets are added into a Bloom Filter of m bits, then γ is approximately $(1 - e^{\frac{-kn}{m}})^k$ [7]. Because the optimal k value is $\frac{mln2}{n}$, γ can be estimated as $(0.6185)^{\frac{m}{n}}$. Thus, to limit the false positive probability to less than $\frac{1}{10^6}$ per packet, we need a 2MB Bloom Filter with 20 hash functions.

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 packets 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 prioritizer to randomize the input to the Bloom Filter before applying the hash functions. Without compromising the prioritizer's secret key, attackers cannot pollute the Bloom Filter with chosen input.

7. EVALUATION

To identify potential performance bottlenecks in the Assayer architecture, we evaluated the performance of each prototype component and compared our two authentication schemes. We also developed an Internet-scale simulator to evaluate how Assayer performs against large botnets.

We find that, as expected, the symmetric scheme outperforms the asymmetric scheme by 1–2 orders of magnitude. Using the symmetric scheme, all three client configurations perform close to the native configurations, with network overheads ranging from 0-11%. Our verifier can sustain about 3300 verifications/second, and the prioritizer can validate Assayer traffic with only a 9.3% decrease in throughput. Finally, our simulations indicate that even sparse deployments (e.g., at the victim's ISP) of Assayer offer strong protection 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 prioritizer has one 2.4 GHz Intel(R) Pentium(R) 4 with 512 MB of memory. All hosts are connected via 100 Mbps links.

7.1 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.

7.1.1 Client Latency

Since clients request new sender tokens infrequently (e.g., once a day, or 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 primary bottleneck for the operation is the time it takes the client to obtain a quote from its TPM, since the quote requires the calculation of a 2048-bit RSA signature on a resource-impoverished TPM processor. On our STMicroelectronics TPM, the quote takes 793.4 ms and constitutes 99.7% of the attestation time. 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.

7.1.2 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 also 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.

7.2 Client Packet Authorization

With Assayer, clients must compute a signature or MAC for each packet they choose to authorize. While some clients may choose to authorize all outbound packets, we expect most clients will only authorize packets when they notice network degradation. Authorizing packets adds computational latency and reduces bandwidth, since each packet carries fewer bytes of application data. To quantify these effects, we experiment with multiple client configurations.

Microbenchmarks indicate that the symmetric scheme adds an average of 12.4 μs to the user-space packet handling routines (i.e., not including the time to route the packet to and from user-space), whereas the asymmetric scheme adds an average of 676.5 μs of per-packet latency. For both schemes, computing the authorization accounts for most of the time: the MAC requires an average of 8 μs, while the elliptic curve signature requires 671 μs.

For macrobenchmarks, 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. These tests require our client module to authorize 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 also significantly increases the number of packets the client module must authorize.

We performed the above experiments using both the asymmetric and the symmetric schemes described in Section 4.3 for all three configurations. Since the relationship between the asymmetric and symmetric schemes was similar across configurations, we present only the comparative results for the Linux configuration (Table 5). All results are the average of 20 trials and include the standard deviations. These results confirm our suspicion that the symmetric scheme offers significantly better performance than the asymmetric scheme. The asymmetric 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 prioritizers.

Table 4 presents our results for the symmetric scheme our various clients. All results are the average of 20 trials and include the standard deviations. As discussed above, the Linux-based client performs quite well. MiniVisor also performs very well, thanks in large part to the hardware support for virtualization that makes context switches extremely fast (approximately 0.5 μs). Both the Linux-based client and the MiniVisor-based client are virtually equivalent to native on several tests and add less than 12% overhead in the worst cases. Xen added surprisingly little overhead to our tests (compared with Linux), and the Xen-based client also performed quite well, with overheads ranging from 0-8%. Its percentage increase is smaller due to Xen's slightly slower performance than native Linux.

7.3 Prioritizer Throughput

In order to evaluate the prioritizer's throughput, we use the Netperf [1] tools running on a client machine to saturate the prioritizer's inbound link with authorized packets. In all tests, we launch the Netperf TCP_STREAM test using 512-byte packets, which is close to the average packet size of the Internet [30].

In our experiments (Table 5), 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 prioritizer implementing our symmetric authorization scheme has about 87 Mbps throughput, while a prioritizer using the asymmetric scheme can only sustain approximately 2 Mbps. As expected, the symmetric scheme is orders of magnitude faster than the asymmetric scheme.

By implementing the prioritizer as a Click kernel module, we improve the performance of the prioritizer using the symmetric scheme to about 154 Mbps, while the performance of a kernel-level basic Click router is approximately 225 Mbps. The prioritizer using the symmetric scheme performs two major operations: verifying packet authorization and detecting duplicate packets (see Section 4.3.3). Eliminating the duplicate detection operation only slightly improves the prioritizer's throughput (up to 169 Mbps), which indicates that verifying the packet authorization is the significant performance bottleneck.

	Unmodified Linux	Assayer Linux Symmetric	Assayer Linux Asymmetric
Ping L	0.817± 0.32	0.814± 0.08 (-0.4%)	2.040± 0.30 (+149.7%)
Ping R	11.91 ± 1.90	12.86 ± 2.88 (+8.1%)	14.07 ± 3.96 (+18.2%)
Req L	3.129± 0.03	3.351± 0.58 (+7.1%)	10.74 ± 4.50 (+243.3%)
Req R	45.83 ± 12.3	45.91 ± 17.9 (+0.2%)	52.53 ± 12.8 (+14.6%)
Down L	1339. ± 348.	1347. ±142.(+0.6%)	2643. ±111.(+97.4%)
Down R	5874. ±1000.	5954. ±525.(+1.4%)	6613. ±720.(+12.6%)
Up L	706.5 ± 61.4	786.5 ±206.(+11.3%)	5194. ±178.(+635.2%)
Up R	3040. ± 568.	3375. ±657.(+11.0%)	6423. ±950.(+111.3%)

Figure 5: Client Authorization: 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 the native configuration.***

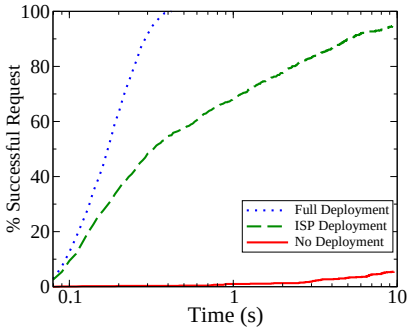

Figure 6: Simulations. ***Time for 1,000 senders to contact the server in the presence of 100,000 attackers. Note that the X-axis is on a logarithmic scale.***

	Native Linux	Assayer Linux	Assayer MiniVisor	Native Xen	Assayer Xen
Ping L	0.817±0.32	0.814±0.08 (-0.4%)	0.811±0.13 (-0.1%)	0.816±0.07	0.838±0.07 (+7.6%)
Ping R	11.91 ±1.90	12.86 ±2.88 (+8.1%)	11.99 ±3.24 (+0.1%)	12.00 ±2.97	11.73 ±2.27 (-2.2%)
Req L	3.129±0.03	3.351±0.58 (+7.1%)	3.48 ±0.26 (+11.3%)	3.984±0.16	4.027±0.18 (+1.1%)
Req R	45.83 ±12.3	45.91 ±17.9 (+0.2%)	44.07 ±6.93 (-0.4%)	45.00 ±10.2	45.71 ±14.0 (+1.6%)
Down L	1339. ±348	1347. ±142 (+0.6%)	1427. ±382 (+6.6%)	1319.0 ±185	1348.0 ±314 (+2.2%)
Down R	5874. ±1000	5954. ±525 (+1.4%)	5884. ±990 (+0.2%)	5900.0 ±1210	5871.0 ±711 (-0.5%)
Up L	706.5 ±61.4	786.5 ±206 (+11.3%)	777.4 ±153 (+10.0%)	711.1 ±64.6	715.5 ±58.4 (+0.6%)
Up R	3040. ±568	3375. ±657 (+11.0%)	3078. ±1001 (+0.1%)	3130.0 ±393	3254.0 ±561 (+3.4%)

Table 4: Client Authorizations (Symmetric). ***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 percentage change versus the native configuration.***

To confirm this, we modify our packet authorization implementation to use UMAC [22] instead of SHA1-HMAC. UMAC is much faster than SHA1-HMAC if the key has been set up, but with Assayer, a key is generated from the client's token and set up for every packet. This would make UMAC slower than SHA1-HMAC. To improve UMAC 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 token. Measurements indicate that the average Internet flow consists of approximately 20 packets [30]. Using this measurement as a rough estimate of our key cache's effectiveness, our prioritizer's performance improves to 204 Mbps. This represents a 9.3% performance loss relative to a kernel-level basic Click router. As a result, it seems plausible that a prioritizer could check authorizations at near line rates.

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

Table 5: Packet Prioritization 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.***

7.4 Internet-Scale Simulation

Finally, to evaluate Assayer's effectiveness against large botnets, we developed an Internet-scale simulator. The simulation's topology was developed from the CAIDA Skitter probes of router-level topology [8]. 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 authorized tokens will result in the revocation of the attacker's keys – see Section 4.4).

Since the Skitter map does not include bandwidth measurements, we use a simple bandwidth model in which endhost uplinks have one tenth the capacity of the victim's network connection, while the rest of the links have ten times that capacity. Thus, each endhost has a small uplink that connects it to a well-provisioned core that narrows down when it reaches the victim. To make these values concrete, 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 connection request every 10 ms, while attackers flood the victim with requests at their maximum uplink capacity. Attackers start sending until the network is saturated (so that legitimate senders face the full brunt of the DDoS attack), and then measure how long it takes legitimate senders to contact the server.

We run our simulations with no Assayer deployment, with Assayer prioritizers deployed at the victim's ISP, and with ubiquitous (full) Assayer deployment. Figure 6 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 only the victim's ISP, we see that more than 68% legitimate clients succeed in less than a second, and 95% succeed within 10 seconds, even in the face of a DDoS attack by 100,000 endhosts.

8. DISCUSSION

8.1 Defeating Additional Network Attacks

We have focused on DDoS attacks, but Assayer can potentially help address other network-based attacks. For example, in an enterprise environment, a client module could authorize packets that are not tainted with sensitive data, allowing firewalls to easily filter outbound traffic. Developing such a client module is challenging, but Assayer provides a generic mechanism to convey such a module's information to the network.

Other attacks, such as spam and click-fraud, also rely on automated endhosts behaving badly. Using a client module that detects human activity (via keyboard or mouse events) or the presence of approved software (such a particular unaltered email client or web browser) would allow mail servers to give preference to emails that are less likely to be spam, and advertisers to reward sites for clicks more likely to originate from humans.

8.2 Legacy and Mobile Devices

While many modern commodity computers are equipped with a TPM [14], a large number of older computers are not so equipped. One approach would be to have ISPs attest to properties that they impose on outbound client packets. For example, an ISP could attest to code on one or more proxies that would prevent the ISPs customers from sending traffic to a particular server at more than X packets/second. After obtaining a sender token, the ISP could authorize traffic to the server on behalf of all of its clients. As long as X is set to a reasonable rate, legitimate legacy clients will not notice a difference, and the server will still be protected.

A similar proxy-based approach would work for mobile devices. There are also specifications for TPMs adapted to the mobile domain [32], as well as a few phones that already contain a TPM [9].

9. RELATED WORK

Closely Related Work. Kandula et al. propose the use of CAPTCHAs to distinguish human-driven web requests from bot-driven requests [17]. Unfortunately, CAPTCHAs are difficult to bind to the specific client request, and they can be outsourced to other humans, whereas Assayer binds sender tokens to the client's platform.

Gummadi et al. propose the Not-A-Bot system [13] 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. Through trace-driven experiments, the authors show that the system can significantly reduce malicious traffic. 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 combatting network-level DDoS attacks.

Both of these systems work well for human-driven application-specific scenarios, but it is difficult to adapt them to services that are not primarily human-driven. For example, Assayer could protect DNS, NTP, transaction processing, network backup, or software update servers, while neither of the above systems cover these cases.

Finally, Ramachandran et al. propose imbuing packets with the provenance of the hosts and applications that generated them [28]. Unfortunately, these packet markings are not secured, so the system must assume that the entire network is trusted and that all hosts have deployed the system in a tamper-proof fashion. By using commodity trusted hardware, Assayer circumvents these assumptions and can operate in networks with hostile elements, with partial deployment, and under attack by malicious software on the hosts.

We note that the last two systems could be implemented securely and efficiently as client modules within Assayer, if those particular properties prove to be desirable to servers.

Conveying Host 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.

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. This system can be viewed as one possible instantiation of Assayer; however, Assayer does not require SELinux or Diffserv, nor does it require universal deployment.

DDoS Prevention. Numerous systems have been proposed to fight DDoS. Because they act solely at the network level, without any reliable host-based information, resource-based systems, such as Portcullis [27] or Speak-up [33], can (provably) achieve fairness amongst endhosts, but they cannot prioritize packets more likely to originate from trustworthy hosts. Similarly, packet capability systems, such as SIFF [34] and TVA [35] assume that the endhost can somehow determine whether a host can be trusted based solely on its packets. This becomes increasingly difficult as adversaries adapt to mimic legitimate hosts.

The recently proposed AIP architecture [2] assigns hosts IP addresses based on a hash of their public key. This provides a convenient mechanism to securely determine that statements signed by a private key "speak for" a particular IP address. Such a technique is orthogonal to Assayer, though it could potentially simplify the process by which a client learns about a server's policy. The AIP authors also propose equipping clients with "smart" network cards that will obey signed shut-off requests from servers. This functionality, could easily be instantiated as a Assayer client module.

10. CONCLUSION

Treating endhosts as black boxes limits the effectiveness of DDoS countermeasures. Using hardware-based attestation to allow legitimate hosts to attest to their benign nature opens the black box and greatly mitigates the effects of botnets, since remotely exploited machines can no longer effectively send attack traffic. As a result, both the network and the application server can spend resources on packets more likely to originate from legitimate sources. Our implementation suggests that Assayer can be implemented in an efficient manner and even partial deployments can be effective against botnets of 100,000 hosts.

11. REFERENCES

[1] Netperf. `http://www.netperf.org`.

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

[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] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[5] S. M. Bellovin. The security flag in the IPv4 header, RFC 3514, April 1st, 2003.

[6] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. In *Internet Mathematics*, 2002.

[8] CAIDA. Skitter. `http://www.caida.org/tools/measurement/skitter/`.

[9] J. Durand. DRM, TPM in the mobile domain. Presentation from the DRM Workshop I, Oct. 2002.

[10] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *SOSP*, Nov. 2006.

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

[12] H. Finney. PrivacyCA. `http://privacyca.com`.

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

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

[15] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets. In *USENIX LEET*, 2008.

[16] Intel Corporation. Intel I/O controller hub 9 (ICH9) family datasheet, Aug. 2008.

[17] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *NSDI*, 2005.

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

[21] O. Kolesnikov and W. Lee. Advanced polymorphic worms: Evading IDS by blending in with normal traffic. TR GIT-CC-05-09, Georgia Tech, 2005.

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

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

[24] E. Messmer. Downadup/conflicker worm: When will the next shoe fall? Network World, Jan. 2009.

[25] D. Mills, A. Thyagarajan, and B. Huffman. Internet timekeeping around the globe. In *Precision Time and Time Interval Applications and Planning*, 1997.

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

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

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

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

[30] K. Thompson, G. J. Miller, and R. Wilder. Wide-area Internet traffic patterns and characteristics. *IEEE Network*, 11:10–23, 1997.
[31] Trusted Computing Group. Trusted platform module main specification. Version 1.2, Revision 103, 2007.
[32] Trusted Computing Group. TCG mobile trusted module specification. Version 1.0, Revision 6, 2008.
[33] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *ACM SIGCOMM*, 2006.
[34] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless internet flow filter to mitigate DDoS flooding attacks. In *IEEE Symposium on Security and Privacy*, 2004.
[35] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. *SIGCOMM*, 2005.