Truth In Advertising: Lightweight Verification of Route Integrity

Edmund L. Wong, Praveen Balasubramanian,
Lorenzo Alvisi, Mohamed G. Gouda, Vitaly Shmatikov
Dept. of Computer Sciences, The University of Texas at Austin
Austin, TX, USA
{elwong,praveenb,lorenzo,gouda,shmat}@cs.utexas.edu

Abstract

We design and evaluate a lightweight route verification mechanism that enables a router to discover route failures and inconsistencies between advertised Internet routes and actual paths taken by the data packets. Our mechanism is accurate, incrementally deployable, and secure against malicious intermediary routers. By carefully avoiding any cryptographic operations in the data path, our prototype implementation achieves the overhead of less than 1% on a 1 Gbps link, demonstrating that our method is suitable even for high-performance networks.

Categories and Subject Descriptors:
C.2.2 [**Computer-Communication Networks**] Network Protocols

General Terms: Security

Keywords: Interdomain routing, security, BGP

1. INTRODUCTION

One of the fundamental features of the Internet is the autonomy of the individual domains which comprise it. Each domain, or *autonomous system* (AS), decides independently how to select the routes and how to forward the data traffic it receives from its customers and peer domains. The control plane of today's Internet, *i.e.*, the mechanism for establishing inter-domain routes, is based on the Border Gateway Protocol (BGP). Each BGP route is a chain of autonomous systems, leading to a particular set of destinations (identified by an IP address prefix). BGP is a *path vector* protocol: the routers of each autonomous system keep in their databases the complete AS-level route leading to each reachable prefix. The autonomous system advertises chosen routes to its peer ASes. The advertised routes need not be the cheapest, fastest or shortest; choosing which routes to advertise and use is an internal decision for every AS.

Upon receiving its peers' advertisements, an AS applies its internal policy to select the "best" route to every destination and update its route database. The policies that govern route selection by individual ASes are completely autonomous and, in general, not necessarily based on just the next hop and/or route length. BGP route advertisements are specifically designed to include the *entire* route. The reason for this is to enable selection policies that involve "deep" route inspection beyond the next hop when deciding which route to use. Route selection policies may take into account service-level agreements, business contracts, previously observed performance, reputation, or any other factors. At the network level, however, all of these considerations must be translated into BGP policies that choose a particular route simply because certain ASes are present or absent in it.

Route selection policies are meaningful only if the advertisements on which they are based are truthful, *i.e.*, if ASes forward traffic in the *data plane* using the routes they advertised in the *control plane*. Control-plane verification of AS-level routes may prevent malicious ASes from advertising unauthorized routes (see Section 2), but they can still deviate arbitrarily from stated routes by forwarding an AS's data down routes that the latter would *not* have selected.

In short, there is no truth in advertising. Inconsistencies between the control plane and the data plane can arise not just because of BGP anomalies [15], router misconfiguration [14], or security compromises. Autonomous systems may also have economic incentives to misroute data traffic. Different ASes charge different fees for carrying their peers' traffic. There may be a substantial financial benefit for an AS to advertise a fast route, which is more likely to be selected by its peers and provide more revenue, but forward data packets using a different, cheaper path. Because route selection depends on the assumption that the advertised route is the one that will actually be used, such inconsistencies can not only nullify the route selection policies used by a particular AS, but also have a negative impact on stability and performance of the Internet.

AS-level traceroute experiments indicate a discrepancy between the control and data planes for at least 8% of Internet routes [15]. This is only the lower bound on the true scale of the problem, because none of the existing route verification methods are secure against misbehaving ASes determined to cover their tracks. For example, traceroute-based verification can be easily evaded by a malicious AS who simply forwards traceroute packets down the "correct" advertised route, while diverting data packets.

Our contributions. We present a lightweight protocol that enables a router, acting as the *verifier*, to verify that its data traffic follows a certain route through the Internet.

Our protocol does this by verifying the presence of a given autonomous system (the *prover*) and its predecessor on the data path. By chaining these verifications, the verifier can localize faults to a particular segment of the route.

Our main objective is *high-performance verification.* This is essential for verifying Internet routes, because the provers may be heavily loaded AS-interconnect routers, which potentially process billions of packets per second. We carefully design our verification protocol so that it (i) completely avoids cryptographic operations in the data path, and (ii) does not require the prover to maintain any long-term flow-specific state. Instead, the prover merely maintains a flow-independent table to efficiently participate in multiple concurrent instances of the verification protocol.

The cost of avoiding cryptographic operations in the data path is an offline setup phase. It is executed infrequently via an out-of-band channel (*e.g.*, a TLS-protected website) and enables the prover and the verifier to share a set of $(2k+1)$-bit random secret tuples, where k is the security parameter. We emphasize that the prover and the verifier do not need to maintain a secure channel outside of this setup. Moreover, the prover does not have to manage individual, online cryptographic state for each verifier. It is sufficient for the prover to remember the secrets that it recently generated, but not with whom they are shared.

To check integrity of some route which is supposed to include the prover, the verifier randomly embeds the first element of a shared secret tuple into some of the data packets on the route. These are the probe packets. In the rest of the data packets, the verifier embeds fresh random numbers. The prover checks for these tags in the incoming packets and responds with the second element in the shared secret tuple to those containing the first element. This checking need not be done in the critical path and has a minimal impact on router performance. The packet itself is forwarded normally.

Our verification method is *secure against malicious adversaries* in the data path who may be trying to divert or drop data packets because probe packets are indistinguishable from normal data packets for everyone except the prover and the verifier. Our protocol is completely *decentralized* and assumes no more knowledge at each router about the state of the network than provided by standard control-plane protocols such as BGP. It is also *incrementally deployable* and requires no changes to TCP/IP.

We give a security analysis of our protocol, and show that our prototype implementation, using the Click software router [13], introduces only 0.38 to 6.10% overhead over 1 Gbps links with almost no impact on latency.

Our focus is on the secure detection of faults in the advertised routes, whether they are caused by a discrepancy between the control and data planes or by a failure in an intermediary router. What to do when the verifier detects a problem is a matter of its own BGP policy. For example, the verifying AS may update its local route selection policy to avoid routes that include certain ASes.

We assume that the prover cooperates in the route verification procedure. In typical Internet scenarios, a rational AS would not want to appear in route advertisements from other ASes if they do not use it (and pay for its services) when the route is selected. Therefore, it has an incentive to help verifiers ensure that if they have chosen a route containing this AS, their data traffic indeed passes through it.

We stress that we do not aim to prevent any AS from re-routing its peers' traffic at will *as long as its data-plane behavior is consistent with its control-plane behavior.* An AS may switch routes as long as it informs its peers by propagating the corresponding route advertisements. Upstream ASes can then re-apply their route selection policies and decide if they still wish to continue using the route.

Organization of the paper. We describe related work in Section 2, our protocol in Section 3, and analytic and experimental evaluation in Sections 4 and 5, respectively. In Section 6, we discuss alternative implementations of our protocol, and conclude in Section 7.

2. RELATED WORK

Much research has been devoted to securing the *control plane* of inter-domain routing by authenticating route update messages [9, 12, 23]. We focus instead on verifying the integrity of the *data plane*, *i.e.*, testing that the actual data paths are consistent with the route advertisements.

Other methods for data-plane verification include *secure traceroute* [17] and *stealth probing* [1]. Secure traceroute requires each verifier-prover pair to establish a shared secret predicate, which is used to mark certain packets as probe packets. The prover's responses include the next hop and are authenticated using message authentication codes (MACs). Stealth probing is even more expensive, requiring a complete IPsec tunnel between the prover and the verifier in order to hide probe packets from intermediate routers.

Both solutions are mainly intended for verifying end-to-end or edge-to-edge connectivity rather than full route integrity (*i.e.*, data packets may still be misrouted in the middle of the path). First, the prover must establish and maintain a separate cryptographic state for each verifier. A single AS may be engaged in thousands of concurrent verifications, requiring an expensive state management mechanism.

Second, cryptographic operations in the critical path of the prover's routers impose a significant throughput penalty when the routers are fully loaded. Stealth probing requires the router to decrypt every received packet; secure traceroute requires the router to compute a MAC for each response to a probe packet, in addition to the possible cryptographic operations involved in recognizing probe packets. Our preliminary tests on the current (February 2007) Click source code [13], which incorporates the IPsec implementation used by stealth probing, indicate a 88% drop in throughput, from around 620 Mbps to 71 Mbps, even after increasing the MTU to prevent fragmentation and adding headroom in the driver to prevent expensive packet copies. It is unlikely that approaches using online cryptography can scale to inter-AS traffic levels and be used to verify a full AS route.

Secure traceroute can also use non-cryptographic predicates on any sufficiently random packet field to identify probe packets. To prevent a malicious intermediate router from learning the predicate, all responses must be buffered at the prover until the next predicate is securely established. This neither provides an immediate feedback to the verifier about the state of the route, nor allows fine-grained control over the probing rate. If the predicate is updated frequently, the prover and verifier need an online cryptographic channel, which is problematic if the prover is a core AS.

Goldberg *et al.* [8] present protocols for detecting and localizing the sources of packet loss in the presence of malicious routers. They rely on cryptographic operations in the data path, and, in the absence of performance measurements, it is not clear whether they scale to AS-level verification. The fault localization protocol of [8] can only check the presence of a prover on the route, and it is unclear whether it can be chained to verify the absence of unwanted ASes. (To the best of our knowledge, the research described in [8] was done concurrently and independently of this paper.)

Other techniques include Listen and Whisper [22], which focuses on end-host reachability, and thus cannot detect whether data packets have been diverted to a different AS path as long as they reach their destination, and Fatih [16], which requires routers to collect and share detailed statistics for each route in order to detect routing anomalies. Collecting per-flow statistics within inter-AS routers that process billions of packets per second does not appear feasible.

Single-packet IP traceback [20] requires routers to maintain Bloom filters with the digests of packets that traversed them. Since AS routers cannot distinguish probes from data traffic, they must keep track of all packets and, to support route verification, allow verifiers secure access to their filters. Storage alone requires roughly 12 GB per minute on a core Internet router [20]. In [19], routers mark each packet's 16-bit fragmentation identifier field with a unique identifier. This scheme is not secure if routers are malicious, since they can easily mark packets with other routers' identifiers.

3. ROUTE VERIFICATION PROTOCOL

Our main protocol enables an AS V (the *verifier*) to check, given an advertised route containing some AS P (the *prover*), whether a particular data packet flow indeed traverses P (more precisely, some border router belonging to P) and the AS that precedes P according to the route advertisement.

In general, our protocol allows any AS in the network to act as the verifier, but specific implementations may impose the restriction that only a single verifier can be verifying a given packet flow at any time. In this case, an edge AS would typically act as the verifier. For example, our prototype implementation, described in detail in Section 5, breaks up secret tags into pieces and embeds these pieces into consecutive packets' IP fragmentation identifier fields. Since each IP packet header contains only one such field, only one verifier can use any given packet for route verification. Although we expect that edge ASes will typically act as verifiers, an intermediary AS can act as a verifier as long as the packets used are not already tagged by some upstream AS. We also discuss an alternative implementation method in Section 6 that supports multiple verifiers through IP options at the expense of increased processing overhead.

For the purposes of this paper, we omit the details of the route advertisement protocol (*e.g.*, standard BGP can be used), and assume that the lifetime of the route is much longer than that of individual connections using that route.

3.1 Design principles

The route verification protocol must be *secure in the presence of malicious intermediate ASes*, even if the latter are aware that route verification is taking place, know the verifier's and prover's algorithms, and are actively trying to evade detection. In particular, probe packets must be indistinguishable from regular data packets. Otherwise, a malicious intermediary can forward all probe packets down the expected route, while diverting regular packets.

Naive solutions, such as asking the destination to measure its received traffic rate, do not solve the problem when both the expected and diverted routes lead to the same end host. Asking the prover's routers to measure traffic rates or other flow characteristics and communicate them to the verifier requires the prover to potentially maintain long-term per-flow state and does not scale to Internet-level route verification.

The same AS may appear on hundreds of thousands of routes, and its routers may be engaged as provers in many concurrent route verifications. Even simple operations such as computing a cryptographic hash function on every packet do not scale to the line speeds of modern AS-interconnect routers. To minimize the prover's overhead, our protocol does *not require any online cryptographic operations on data packets.* The prover and the verifier need not maintain pairwise secure channels and/or cryptographic associations.

Our use of cryptography is limited to an *offline* setup phase (see Section 3.4). In Section 6, we describe an alternative that uses two cryptographic hash functions during route verification without online pairwise secure channels.

Algorithm 1 Verifier protocol

```
/* data forwarding loop */
for every packet $m$ heading to $S$ via $P$ do
  if random(0, 1) < Pr(tag) then
    if random(0, 1) < Pr(secret) then
      remove some $(s_1, s_2, b)$ from $\mathbb{S}_{V,P}$
      tag $m$ with $s_1$
      if $b = 0$ then
        $\mathbb{T}_1 = \mathbb{T}_1 \cup \{s_2\}$, $\mathbb{T}_2 = \mathbb{T}_2 \cup \{s_2 \oplus \mathsf{routeid}(m)\}$
      else
        $\mathbb{T}_1 = \mathbb{T}_1 \cup \{s_2 \oplus \mathsf{routeid}(m)\}$, $\mathbb{T}_2 = \mathbb{T}_2 \cup \{s_2\}$
      end if
    else
      tag $m$ with fresh $k$-bit random number $\bar{s}_1$
    end if
  end if
  forward $m$ to next hop
end for

/* response handling loop */
for every response $m$ that contains tag $t$ do
  if $t \in \mathbb{T}_1$ then
    $\mathbb{T}_1 = \mathbb{T}_1 \setminus \{t\}$, $n_{\text{valid}} = n_{\text{valid}} + 1$
  else if $t \in \mathbb{T}_2$ then
    $\mathbb{T}_2 = \mathbb{T}_2 \setminus \{t\}$, $n_{\text{invalid}} = n_{\text{invalid}} + 1$
  end if
end for
```

3.2 Verifying presence of an AS

We now describe our main protocol which allows the verifier to check the presence of a certain prover on a given data path. We extend this protocol to additionally verify the identity of the preceding AS in Section 3.3.

Consider an abstract route shown in Fig. 1. A border router in AS V (the *verifier*) is using this route to send traffic to an edge network S. E and P are intermediate ASes on the route. V wants to verify that P (the *prover*) is indeed present on the data path. We assume that P is cooperating, but E may be malicious. P may be concurrently engaged in other instances of the protocol with other verifiers $\mathbb{V}'$.

In Section 3.4, we describe an *offline* setup phase, after which V and P share a set $\mathbb{S}_{V,P}$ of secret tuples (s_1, s_2, b)

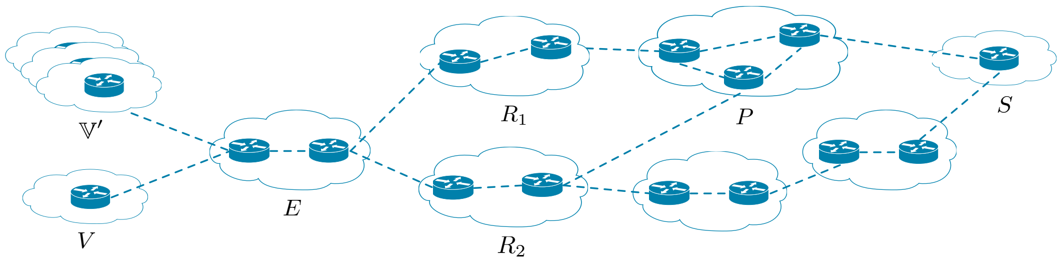

Figure 1: Routes between V and S. The advertised route is $E \rightarrow R_1 \rightarrow P \rightarrow S$.

Algorithm 2 Prover protocol

```
/* data forwarding loop */
for any incoming packet $m$ do
  if $m$ contains tag $t$ and return address $V$ then
    $\mathbb{Q} = \mathbb{Q} \cup \{(t, \mathsf{routeid}(m), V)\}$
  end if
  forward $m$ to next hop
end for

/* tag handling loop */
for $(t, \mathsf{routeid}(m), V)$ in queue $\mathbb{Q}$ do
  $\mathbb{Q} = \mathbb{Q} \setminus \{(t, \mathsf{routeid}(m), V)\}$
  if $\exists(s_1, s_2, b) \in \mathbb{S}$ such that $t = s_1$ then
    if $b = 0$ then
      generate anonymous response $\hat{m} \equiv s_2$
    else
      generate anonymous response $\hat{m} \equiv s_2 \oplus \mathsf{routeid}(m)$
    end if
    send $\hat{m}$ to $V$ after random delay $\mu$
  end if
end for
```

consisting of two random k-bit secrets s_1 and s_2 and one random bit b. All of P's border routers store the set $\mathbb{S}_P$, which contains all secret tuples known to P. This is a single set, not partitioned by verifier.

Once verification begins, V randomly "tags" some of the data packets with the first element s_1 from any secret tuple $(s_1, s_2, b) \in \mathbb{S}_{V,P}$, and the rest with a fresh, random k-bit tag $\bar{s}_1$. Each secret tuple from $\mathbb{S}_{V,P}$ is used at most once. Along with each tag, V also embeds a return address that P can use to contact V. Note that only V and P can tell the difference between s_1 and $\bar{s}_1$. We will refer to the packets containing secret tags as *probe packets*.

When a tagged packet m arrives at one of P's border routers, it extracts the tag t from m. Suppose $\exists(s_1, s_2, b) \in \mathbb{S}_P$ such that $t = s_1$. If $b = 0$, then P replies with s_2. If $b = 1$, P replies with $s_2 \oplus \mathsf{routeid}(m)$, where $\mathsf{routeid}(m)$ is a *route identifier* based on m's source and destination ASes or address prefixes, padded with 0's to k bits. The reply is sent to the return address included with the tag.

A malicious intermediary may attempt to selectively drop replies originating from certain provers, thus "framing" them as the source of packet loss on the route. To prevent this, replies are *anonymized* by setting their source addresses to a constant pre-defined value regardless of the prover's IP address. Even though a malicious intermediary E cannot tell which reply corresponds to which tag, E may attempt to exploit timing information to determine which downstream AS is sending the replies, *e.g.*, by measuring the time elapsed between the first tagged packet in the flow and the first observed response. To prevent this, the prover may delay each reply for a random duration, proportional to the number of hops between the prover and the destination AS.

Regardless of whether the tag is a shared secret, the data packet itself is routed normally by the prover. With the exception of the initial tag extraction, tag analysis can be done outside the critical packet forwarding path. If the prover's router is under a heavy load, it can continue forwarding packets, queuing extracted tags for later processing.

V keeps a list of *valid* replies which initially contains, for every secret tuple (s_1, s_2, b) used on packet m, s_2 if $b = 0$ and $s_2 \oplus \mathsf{routeid}(m)$ if $b = 1$. In addition, V also keeps a list of *invalid* replies, which initially contains $s_2 \oplus \mathsf{routeid}(m)$ if $b = 0$ and s_2 if $b = 1$. V records P's valid and invalid replies. As soon as a valid or invalid reply is received, it is counted and removed from the appropriate set (if k is sufficiently large, even a single invalid reply indicates a route failure; see Section 4.2). Any other reply, *i.e.*, a reply containing a number which is neither s_2, nor $s_2 \oplus \mathsf{routeid}(m)$ for some (s_2, m), is simply discarded. We describe a general statistical method for using the number of valid and invalid replies to determine whether a route is faulty in Section 4.

To further lighten their load, P's routers can check only a random fraction of the packets for tags and forward the rest unchecked. This has the effect of increasing the drop rate of the route as observed by the verifier since some of the normally routed packets will not produce a valid reply, and therefore will appear to have been dropped and/or diverted. We discuss the implications of this in Section 4.

V may opt to tag only some of the data packets along the route with random tags. This decreases both the verifier's and the prover's overheads, but a malicious intermediary can divert or drop the untagged packets with impunity since they are easily distinguishable from probe packets.

3.3 Verifying predecessor AS

We now describe an extension to the basic protocol of Section 3.2 that enables the verifier to check whether the AS *preceding* the prover on the data path is consistent with the route advertisement. This is essential for chaining verifications together to check integrity of the entire route.

The main idea is to use a separate set of secret tuples $\mathbb{S}_P^{R_i}$ per each interface of P's border routers instead of the same $\mathbb{S}_P$ for all interfaces. As before, V and P share a set of use-once $(2k+1)$-bit secret tuples $\mathbb{S}_{V,P}$. During the setup phase (see Section 3.4), V specifies, for each secret tuple it shares with P, what the expected previous AS R_i will be. The tuple is then added to the appropriate $\mathbb{S}_P^{R_i}$ set, which is later installed at border routers that connect P to AS R_i.

Consider again the abstract route depicted in Fig. 1, where P is preceded by R_1 in the advertised route. In this example, for each secret tuple that V obtains from P during the setup phase, V informs P that this secret will arrive from R_1. These tuples are then added only to the set $\mathbb{S}_P^{R_1}$, which is installed only at the interfaces connecting P to R_1. If one of these secrets then later arrives to P via some other AS R_2, the secret will be checked against a *different* set $\mathbb{S}_P^{R_2}$. Since the secret does not appear in this set, the border router will not generate a valid reply, enabling the verifier to detect a problem with the route as described in Section 4.

This method of verification can be *chained*. If all provers on the path cooperate, V can check that P_1 is the predecessor of P_2, P_2 is the predecessor of P_3, and so on. V simply runs multiple instances of the protocol, maintaining a separate set of secrets and separate statistics for every prover P_j on the route. When secrets are inserted into packets, the appropriate sets of expected and unexpected responses are updated. When a response comes back, V checks whether this is an expected or unexpected response for any prover, and, if so, updates the sets accordingly.

Completely checking a route consisting of n ASes requires n executions of the verification protocol. Verifications can be interleaved, *i.e.*, consecutive data packets may contain tags destined to different provers. If the verifier is only interested in checking the presence of each AS, $\lceil n/2 \rceil$ executions are sufficient, since a single instance of the protocol verifies the presence of both an AS and its immediate predecessor.

By chaining verifications, the verifier can check the integrity of an entire route and determine that no additional ASes—those not mentioned in the route advertisement—have been inserted into the route. We describe how the cause of faults can be localized in Section 4.

3.4 Offline setup and secret exchange

Our protocols require a setup phase, in which V securely obtains from P a set $\mathbb{S}_{V,P}$ of secret tuples (s_1, s_2, b), consisting of two random k-bit secrets s_1 and s_2 and one random bit b. This is done *offline*, independent of route usage, and only requires P to remember what secrets have been distributed, not who obtained them. Thus, P does not need to manage separate cryptographic state for each verifier.

One simple method by which P can provide secrets to potential verifiers is through a TLS-protected website, from which an authenticated verifier V_j can securely download a fresh random seed z_j. Whenever P wants to provide a new set of tuples to its verifiers, P publishes a long random number r and the counter t, representing the current "generation" of shared secrets. When V_j wants to acquire a new set of secret tuples for a particular generation t, V_j contacts P's server, supplying the identity of the AS R_i that precedes P on V_j's route (see Section 3.3) and the number of tuples n_j that V_j would like to generate. Applying a cryptographic pseudo-random number generator (PRNG) to r, t and z_j, V_j and P generate a new set of shared $(2k+1)$-bit secret tuples $\mathbb{S}_{V_j,P}(t)$ corresponding to a particular generation t.

P generates the entire set $\mathbb{S}_{V_j,P}(t)$, and adds it to the current set $\mathbb{S}_P^{R_i}(t)$ corresponding to the interface connecting P with AS R_i (specified by V_j as P's predecessor). V_j does not have to generate all n_j tuples at once; they can be generated later from r, t, and z_j, provided V_j remembers the state of the PRNG. The secret sets must be completely replaced at the prover and verifier at the beginning of a new generation.

This setup is done completely off the critical data path. The next generation of shared secrets $\mathbb{S}_P^{R_i}(t+1)$ can be set up while the current generation $\mathbb{S}_P^{R_i}(t)$ is still in use by P's border routers. Verifiers are expected to check whether their set of shared secrets is current, which can be done offline.

Even though the prover's border routers have to store a large set of secret tuples, we contend that the space overhead of our protocol is feasible for modern core routers. As described in Section 5, our protocol uses a 4-way cuckoo hash table [7]. A 2 GB 4-way cuckoo hash table can hold roughly 130 million 129-bit secret tuples (consisting of two 64-bit secrets), which is enough for approximately 4,500 ASes to send 20 secrets per minute for a day to a single incoming interface on a prover's router. Considering that a state-of-the-art core router such as Cisco's CRS-1 has several gigabytes of memory per interface card [4], our space overhead appears practical for core Internet routers.

3.5 Route selection policies

Our focus in this paper is on developing a mechanism for detecting and localizing route faults, not on what an AS may do once a fault has been detected. Like other considerations governing BGP route selection, this is an internal policy decision that is made independently by each AS. There may be legitimate reasons for discrepancies between the control plane and data plane, *e.g.*, during BGP routing changes [15]. In general, each AS must decide individually how to translate the fact that a particular data path deviates from the advertised route into a specific BGP policy.

4. SECURITY ANALYSIS

The main purpose of our protocol is to enable verifier V to check whether a given data route passes through a certain AS P. In this setting, there is no conceptual difference between packets that are "naturally" dropped due to network imperfections, those dropped by a malicious intermediate AS E, and those diverted by E to a different route that does not include both P and its predecessor. In all of these cases, the packet's actual data path is different from the advertised route, indicating an inconsistency or route fault.

The problem of detecting route faults is effectively that of *secure network measurement*. More precisely, given any threshold δ, V must be able to tell, in the presence of a malicious E, whether the drop rate of V's route to P is above δ or not. The specific value of δ is determined by V's internal policy, *i.e.*, it is up to V to decide what drop rate is acceptable. Our protocol does not depend on a specific δ.

We do not aim to verify complete packet integrity, which is a much more difficult problem (arguably, packet integrity, as opposed to *route* integrity, should be addressed by end-to-end mechanisms such as IPsec). Similarly, we do not aim to detect adversaries who, for example, create a copy of each data packet, route the copy correctly and the original packet incorrectly, or vice versa. We do not view this as a significant limitation. It is not clear whether a malicious intermediary AS E has any incentive to stage such an attack, as it increases the amount of traffic E has to route. Our protocol does ensure that if only some of the routes passing through E are being verified, E cannot cheat by switching tags between packets on different routes.

4.1 Tracking valid and invalid responses

V measures the connection's drop rate δ by keeping track of the valid and invalid responses it receives. Recall that if V

sends a packet m containing a known secret s_1, P responds with another secret s_2, XORed with $\mathsf{routeid}(m)$ if $b = 1$. Consequently, there are three different types of responses that V may receive: a *valid* response (s_2 if $b = 0$, $s_2 \oplus \mathsf{routeid}(m)$ if $b = 1$), an *invalid* response ($s_2 \oplus \mathsf{routeid}(m)$ if $b = 0$, s_2 if $b = 1$), or a *fake* response (any other value).

V ignores replies that are neither s_2, nor $s_2 \oplus \mathsf{routeid}(m)$. If V counted the number of other responses, an adversary who is not even present on the route could confuse V into believing that the route is faulty by sending a large number of fake responses. In our protocol, an AS must be actually present on the data route in order to send either a valid or invalid response to V's probes.

It is important, however, that V track the invalid responses it receives. Otherwise, a malicious intermediary E could "launder" V's probes through a flow originating from a different AS, one that is not performing verification. We describe this attack in further detail in Section 4.3.

4.2 Fault detection and localization

Suppose V tags n packets with a secret tag and receives $\hat{n}$ valid responses. Each test is executed for some time $\tau(\epsilon)$, which is set to be much longer than the round-trip latency of the connection. More specifically, if n' represented the number of responses generated by P in response to those of V's packets that reach it, we assume that $(1-\epsilon)n'$ of P's responses will reach V within time τ.

If the route does not include malicious routers, V should expect to receive

$$(1-\epsilon)(1-\delta)^2 n \equiv \theta_n n \qquad (1)$$

responses to probe packets tagged with shared secrets. Here $(1-\delta)^2$ is the probability that neither the probe packet, nor P's response to it has been dropped in the network.

It is possible that V receives a "legitimate" invalid response if one of V's random tags collides with one of P's actual secrets s_1, shared with either V or another verifier, and the corresponding response happens to collide with some value in the invalid reply set. This probability is negligible for large k, so we assume that V should expect to receive no invalid replies. Thus, even a single invalid reply indicates the presence of a malicious intermediary on the route.

We now describe the valid-response test in detail. Let $\theta = \hat{n}/n$ be the fraction of probe packets for which the verifier receives responses. We then have two statistical hypotheses:

H_1: $\theta \geq \theta_n$. The response rate to probe packets is at least as expected.

H_a: $\theta < \theta_n$. The response rate is lower than expected.

We will say that the route is *faulty* if the verifier can reject the hypothesis H_1 with high confidence. The problem of fault detection now reduces to standard statistical hypothesis testing: what is the threshold n_1 such that if verifier V sends n probe packets and receives at most n_1 responses, V can reject hypothesis H_1? Note, however, that attempting to minimize the false positive ratio (declaring a normal route faulty) increases the false negative ratio (declaring a faulty route normal), and vice versa.

To solve the problem, we use a uniformly most powerful (UMP) test [5]. For any significance level α, a UMP test caps the false positive rate at α while guaranteeing the lowest probability of a false negative for any statistical test

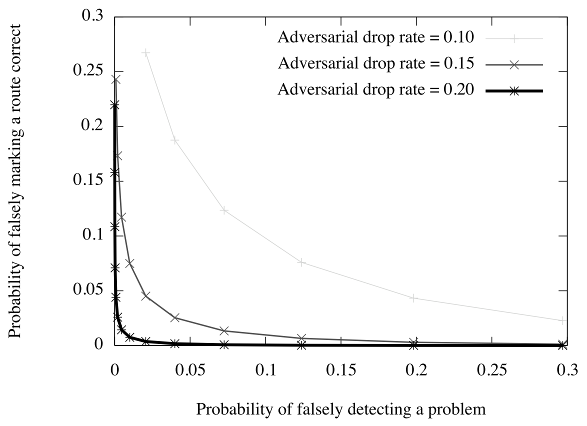

Figure 2: The tradeoff between incorrectly detecting a fault and marking a faulty route correct for a UMP test over the number of valid responses given 100 secret tags ($n = 100$), an expected response rate of 0.9 ($\theta_n = 0.9$), and varying adversarial drop rates (η).

with significance α. In our setting, α is a parameter of the verifier's algorithm; therefore, for any given α, our UMP test maximizes the probability of correctly detecting a fault, while capping the probability of falsely marking a normal route faulty to be at most α for any θ_n.

From the viewpoint of the verifier, the network is a "black box." The verifier sends a probe packet, and with some probability a response comes back. Therefore, we model the receipt of responses from the network as a Bernoulli process, where the number of valid responses to n probe packets is a random variable that follows a binomial distribution with n trials and probability θ_n of success. The question "is the route faulty?" is then equivalent to the question "is θ less than θ_n?", which we answer with the following UMP test for binomial distributions [5]:

For a given level of statistical significance α, find n_1 such that

$$\Pr(\hat{n} \leq n_1 \mid \theta = \theta_n) = \alpha \qquad (2)$$

is satisfied for a sample of size n drawn from a binomial distribution with parameter θ_n.

Accept H_a (mark route as faulty) if the number of valid responses $\hat{n}$ is at most n_1 ($\hat{n} \leq n_1$). Otherwise, if $\hat{n} > n_1$, accept H_1 (consider the route to be normal).

In addition, given a particular UMP test of significance α and threshold n_1, we can calculate the probability of a false negative given a certain adversarial drop rate. If we assume that there is an adversary who is dropping or diverting η of the packets, the probability that our test will falsely mark the route as being correct is

$$1 - F(n_1; n, \theta_n(1-\eta)) \qquad (3)$$

where $F(x; n, p)$ is the binomial cumulative density function with parameters n and p given x successes.

In summary, for any false positive rate α, we use equation (2) to calculate a threshold n_1 that guarantees this rate and minimizes the false negative rate, which we calculate for any adversarial drop rate using equation (3).

Fig. 2 illustrates the tradeoff for $n = 100$, $\theta_n = 0.9$, and various adversarial drop rates η. For a test that has significance level $\alpha \leq 0.01$, (2) dictates that $n_1 = 81$ ($\alpha = 0.0046$). Thus, receiving at most 81 responses occurs about 0.46% of the time given our choice of θ_n and n, and we can conclude that there is a greater than 99% chance that the route is faulty. Given this test, if there is an adversary along the path dropping $\eta = 0.15$ of the tags, then the probability of correctly detecting the problem given our test ($n_1 = 81$) is approximately 0.883. A test with significance $\alpha = 0.05$ is slightly more aggressive: receiving fewer than $n_1 = 84$ is enough to mark the route faulty. However, this test will detect an adversary dropping $\eta = 0.15$ of the tags with probability of approximately 0.975.

As θ_n decreases, either due to an increase in the network drop rate or optimizations that throw away some of the tags (as mentioned in Section 3.2), the number of trials necessary to ensure statistical significance of our test increases.

Our test detects faults on the route from the verifier to the prover *or* from the prover to the verifier (because of asymmetric routing, the two routes may be different). For example, even if the actual data path is correct, but all replies from the prover are misrouted or dropped, the verifier will declare the route faulty.

The verifier can test the route at discrete times, or continuously in overlapping "sliding windows." The latter requires the verifier to maintain a sliding list of n most recent packets, with which to compare received responses.

In addition to detecting route faults, our protocol also helps V localize the source of the problems on the route. Because a malicious intermediary E cannot tell which downstream prover sent a given response, it cannot selectively drop responses from a particular prover. Therefore, whenever V observes that a route is faulty, blame for the fault can be assigned to some AS between the last prover whose responses arrive correctly and the first prover who does not return the expected responses.

If chaining is done as described in Section 3.3 and there is a single malicious AS on the route, this localizes the fault to a pair of consecutive ASes (our protocol cannot tell the difference between a malicious AS preceded by a good AS, and a good AS preceded by a malicious AS who is dropping all of its replies). In Section 4.3, we discuss the scenario when the route contains multiple colluding malicious ASes.

4.3 Attacks

Suppose the route from verifier V to prover P contains a malicious router E. The primary attack we are concerned with is E dropping data packets or diverting them to a different route that does not include P. E could also potentially drop or reroute responses. We view these attacks as equivalent in the sense that both attacks result in data not traversing P and thus no response returning to V.

Other possible attacks include E diverting packets to a route that includes not only P, but also additional ASes that were not part of the route advertisement; E "framing" a well-behaved AS by making it appear that it is misrouting packets; E switching tags between flows; and attacks by malicious routers who are *not* present on the route, but try to make the route appear faulty by generating bogus responses.

A malicious E on the route may also abuse our protocol to stage a "reflection" attack on some victim by replacing the verifier's return address in tags with the victim's address, thereby directing the prover's responses to the victim. However, the probe rate in our protocol is too low for this to be an effective denial-of-service attack, and it will result in E being detected as the cause of route fault.

Security of our protocol relies on four properties:

E cannot tell the difference between probe packets and data packets. If E drops or reroutes a packet containing some shared secret s_1, V will not receive the expected response. Since E cannot differentiate s_1 from a random tag $\bar{s}_1$, E must route packets containing s_1 down the advertised path to avoid detection.

Because the prover's replies are not authenticated, E may attempt to generate a fake response to any given tag. Since V expects responses only to probe packets, E must guess correctly whether a given k-bit tag is a shared secret s_1 and, if so, what are the values of s_2 and b in the corresponding tuple. E's probability of guessing correctly is $\frac{|\mathbb{S}_V|}{2^k} \cdot \frac{1}{2^k} \cdot \frac{1}{2}$, which is negligibly small.

E cannot determine which responses contain an encoded route identifier. Although E itself cannot determine which tags are secrets, E could potentially still misroute V's packets. Consider a malicious intermediary AS E which is processing two concurrent flows from ASes V and W, respectively, both of which are supposed to traverse some downstream AS P according to the advertised routes. Suppose that only V is performing verification and thus only packets in V's flow are tagged.

In this case, for every packet m_V from V's flow, E may attempt to extract the tag, insert it into a packet m_W in W's flow, route m_W correctly and misroute m_V. By forwarding any resulting responses back to V, E can misroute V's traffic yet avoid detection by V.

To prevent this, prover's replies generated according to the protocol of Section 3.2 depend on the value of a secret bit b. If $b = 0$, the expected reply is the second element s_2 of the secret tuple. Otherwise, the expected reply is $s_2 \oplus \mathsf{routeid}(m_V)$. Thus, for each tuple whose bit $b = 1$, P's response to m_W will be different from its response to m_V. Since E does not know the value of b for any given response $\hat{m}$, E must guess correctly whether $\hat{m} = s_2$ (in which case E must forward $\hat{m}$ unmodified), or $\hat{m} = s_2 \oplus \mathsf{routeid}(m_W)$ (in which case E must forward $\hat{m} \oplus \mathsf{routeid}(m_W) \oplus \mathsf{routeid}(m_V)$ to avoid detection).

Because E knows that $\hat{m}_W$ is either s_2 or $s_2 \oplus \mathsf{routeid}(m_W)$, E could send *both* $\hat{m}_W$ and $\hat{m}_W \oplus \mathsf{routeid}(m_W) \oplus \mathsf{routeid}(m_V)$ to V, one of which (E does not know which) is the expected valid response. To prevent this attack, V tracks invalid responses and declares the route faulty if it receives any.

E cannot tell which prover sent a given response. An adversarial AS E on the route cannot tell the difference between a random tag and a shared secret when it observes a tagged packet. E may attempt to attack the protocol by selectively dropping responses from some downstream prover in an attempt to *frame* it as being faulty. However, by setting the source IP address of all replies to a default value and randomly delaying them, our protocol prevents E from determining which downstream prover sent a given reply. Therefore, a single adversary E cannot selectively drop tags destined to or responses originating from a particular downstream AS in an attempt to frame it.

If there are *multiple* colluding adversaries on a given route, they can distinguish responses generated by provers who lie between any two adversaries on the route from those generated by provers who are downstream from the last adversary. For example, consider some route that contains $E_1 \to \mathbb{P} \to E_2$, where E_1 and E_2 are colluding adversaries and $\mathbb{P}$ represents some set of provers lying between E_1 and E_2. If a response is observed only by E_1, but not by E_2, they can determine that it must have originated from some prover in $\mathbb{P}$. Nevertheless, E_1 and E_2 still cannot tell *which* prover in $\mathbb{P}$ produced the response, and cannot selectively "frame" a single prover within $\mathbb{P}$. Thus, V can localize the fault to somewhere between the last successfully verified AS and the first one for which verification failed.

E cannot change the predecessor specified by V. Even if the data path passes through all ASes listed in the advertised route, a malicious intermediate router may attempt to *insert additional ASes* into the path. For example, even though the advertised route goes directly from E to P, E may divert the traffic to another malicious AS, who then forwards it to P. As long as the packets eventually pass through P, the basic protocol of Section 3.2 will not detect the deviation. However, in the extended protocol described in Section 3.3, V specifies which incoming interface will be used and thus where the secrets will be stored. Packets then must arrive to P through the specified predecessor. Secrets arriving to P through the wrong incoming interface will not generate the expected responses.

The only exception is the case when the route contains multiple *consecutive* malicious ASes who are willing to lie to the verifier. They can insert additional ASes between themselves without being detected. Moreover, a malicious router can always copy all passing packets and share them with the adversary. It is not clear whether such eavesdropping attacks can be detected using *any* verification method that does not give the verifier complete access to the internal state of all routers on the data path, including those controlled by malicious entities.

5. EVALUATION

Experimental setup. We built a prototype implementation of the prover and verifier within the Click 1.5.0 software router [13] on top of a modified Linux 2.6.16.13 kernel. Our verifier and prover run on Pentium III/933 machines with 896 MB of RAM with two 64-bit AceNIC Gigabit Ethernet cards in 64-bit 33 MHz PCI slots. Each end host had one AceNIC Ethernet card and 512 MB of RAM but were otherwise the same. The machines are all connected as shown in Fig. 3 via crossover cables. All experiments were run using both Fast (100 Mbps) and Gigabit (1 Gbps) Ethernet links. Verification was performed only on the traffic heading from the client to the server. We vary the probe rate, or percentage of secret-tagged packets, and measure throughput, latency, and CPU utilization.

Implementation. In our protocol, we chose to use 8-byte tags and 4-byte return IP addresses, which are broken up into six 16-bit pieces. We insert each piece into the 16-bit fragmentation field of the header of each non-fragmented IP packet. Since packets are rarely fragmented [21], this field should almost always be available for tagging.

Our prover maintains a two-level sparse table of the most recent tag fragments it has received from a particular verifier indexed by the route identifier, which it uses to reconstruct the tag and the return address, which is then looked up in the secret table. This requires 12-byte route-specific state at the prover to temporarily store fragments until the tag and the address are reconstructed. This short-lived state is an implementation artifact and not a necessary part of the protocol. We discuss alternative implementations in Section 6.

Since generating responses to shared secrets is the main bottleneck at the prover, splitting each secret over six data packets reduces the prover's workload by decreasing the average number of responses per packet on the route that is being verified. It also makes a given set of shared secrets last six times longer before a new secret set has to be generated.

Because tags are broken up into multiple fragments which are inserted into consecutive packets, the loss or misrouting of any secret fragment is equivalent to losing or misrouting the entire secret. Moreover, this implementation is sensitive to packet reordering. If packets tagged with secret fragments are reordered, the reassembled secret will likely be invalid and thus generate no reply. Although reordering is rather infrequent [18], our statistical analysis must still take it into account. Assuming that any packet drop or reorder causes a secret to be invalid, we redefine θ_n from (1) to

$$(1-\epsilon)(1-\delta)^6(1-\delta)(1-\phi)^6 n \equiv \theta_n n$$

where ϕ is the probability of packet reordering between the verifier and prover and $(1-\delta)^6$ represents the probability that all tag fragments reach the prover. In Section 6, we discuss an alternative that is not sensitive to packet reordering.

The prover and verifier consist of a kernel-level Click component to handle packet modification and preliminary checks, and a user-level process to transmit secrets to and from the kernel-level component, track tag usage, and reply to secret tags. The prover and verifier start off by sharing a set of 312,500 secret tuples. The prover then loads the entire secret set, while the verifier buffers approximately 20,000 secret tuples. As the verifier tags packets and uses up secret tuples during route verification, the user-level process pushes more to the verifier's buffer via the `/proc` filesystem interface. The verifier uses RC4-based frandom [2] after discarding the first 256 bytes to randomly decide which packets to tag with secrets and to generate random secrets.

When a packet arrives at the prover's router, it is checked to see if the fragmentation offset is zero and the "more fragments" flag is disabled. A tag fragment is extracted from the fragmentation identifier field and inserted into the appropriate sliding window buffer, which is then looked up in a 8 MB 4-way cuckoo hash table [7] using four hash functions [11]. A cuckoo hash table guarantees constant-time lookups at the cost of linear space overhead. Since such a table could potentially be extremely large, we allocate it within a user-level process. However, every tag then must be copied from kernel memory to the user-level process's memory in order to be checked against the table of secrets. Our implementation avoids many of these copies through a small 1 MB Bloom filter [3] with four hash function within our kernel-level prover. Through an extra set of lookups, it can filter out many random tags, thus significantly reducing our memory-copying overhead. Tags that pass the smaller filter are copied via `/proc` to the user-level process, which then checks the hash table. For every tag that the hash table classifies as a secret, the user-level prover generates a UDP packet with the appropriate response as a payload and sends it to the listed IP address.

The verifier listens for incoming UDP responses and checks them against the expected response set, which we implement using a counting Bloom filter [6], a generalization of

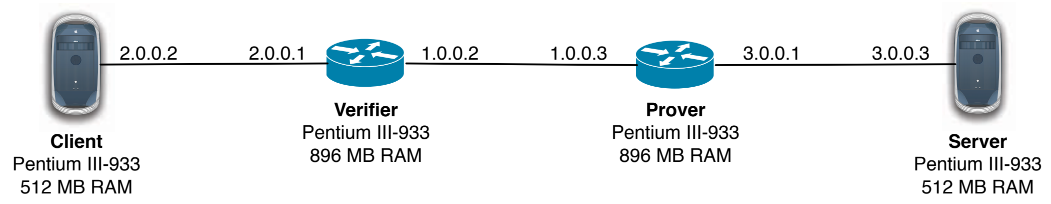

Figure 3: Our testbed.

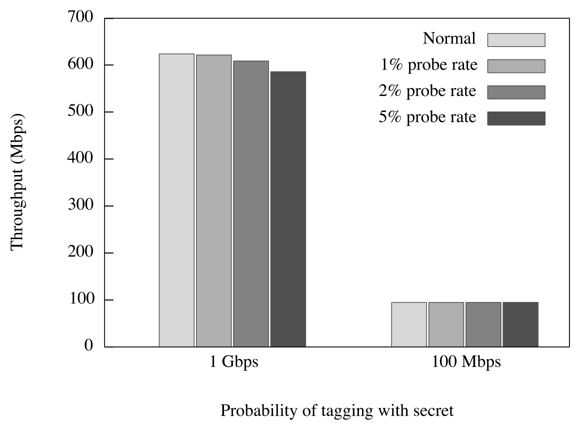

Figure 4: Average throughput of running an unmodified Click router compared to our protocol with a varying proportion of probe packets.

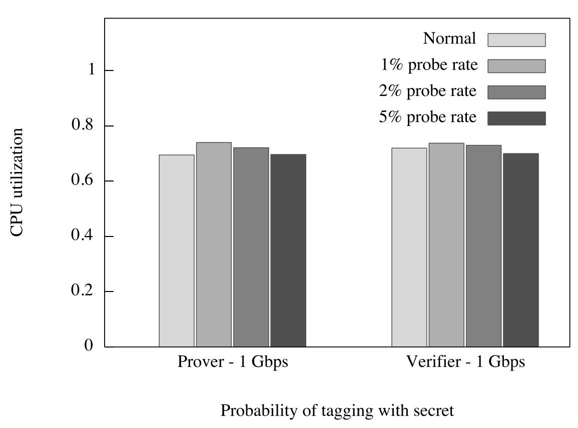

Figure 6: Average CPU utilization of our verifier and prover.

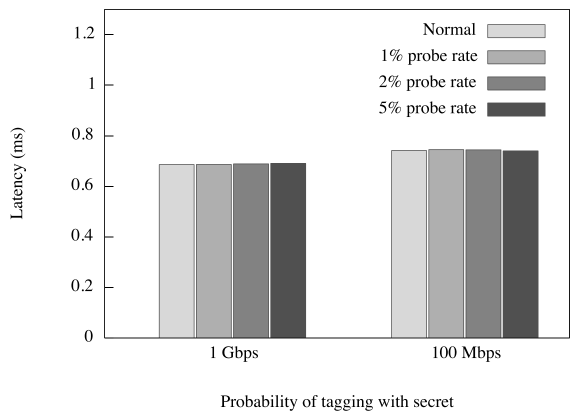

Figure 5: Average round-trip time of a packet.

the Bloom filter that allows probabilistic element removal. Alternatively, we could have used a larger, non-probabilistic hash set to store all secrets.

Results. To estimate the overhead of our protocol, we used Iperf 2.0.2 [10] to measure the average throughput of twenty-one 30-second unidirectional TCP bulk data transfers over both Gigabit (1 Gbps) and Fast Ethernet (100 Mbps) links. Since Iperf was continuously transferring data as quickly as possible, our packets were almost all 1500 bytes in length.

Fig. 4 shows the average throughput achieved with different proportions of probe packets. Using the unmodified Click router, we were able to forward approximately 624.1 and 94.7 Mbps over Gigabit and Fast Ethernet links, respectively. Performing verification on Fast Ethernet resulted in no overhead regardless of the proportion of probe packets. This is likely due to our machines being sufficiently powerful to handle verification at line speed.

We then repeated our experiments on Gigabit Ethernet links. Verification with a 1%, 2% and 5% probe ratio resulted in, respectively, throughputs of 621.7, 608.8, and 586 Mbps, overheads of 0.38%, 2.45%, and 6.10% over the standalone Click router. In our experiments, every packet is tagged and thus must be inspected by the prover, but increasing the probe rate increases the number of UDP responses that the prover must send back. Disabling the prover and running just the verifier with a 5% probe ratio, we achieved an average throughput equivalent to that of running two unmodified Click routers. This indicates that the overhead of random tagging is very small even when all packets are being tagged, and that the overhead is mainly caused by responding to probe packets on the prover side.

Another significant contributor to the overhead is the transfer of tags from kernel to user memory, which we solved with a smaller, kernel-level Bloom filter. In the throughput experiments, we used a 1 MB filter to hold 312,500 secrets. Decreasing the size of the filter to 256 KB dropped the throughput of 1% probing by about 13.7%.

We used 3000 ping requests to measure the average round-trip latency of the packets traveling through routers running our route verification protocol. On 100 Mbps and 1 Gbps links, our protocol had no statistically significant impact on latency, even when we increased the probe rate.

Finally, we measured the system load of running our protocol. We used the `top` utility to check CPU utilization of the Click kernel thread and our user-level verifier and prover every 3 seconds for 180 seconds while running our throughput experiments. Fig. 6 shows the CPU overhead of our protocol. Running the unmodified Click router uses about 70 to 72% of the CPU. Running our modified version of the Click router results in a 70 to 74% CPU utilization on both the prover and verifier when the probe rate is decreased from 5% to 1%, respectively. Although it may seem odd for the CPU utilization of our protocol to drop as the probe rate increases, note that the Click router itself is more CPU-intensive than our protocol; probing 1% of the packets incurs an overhead of at most 6.5% over the standalone Click router even when the throughput is roughly the same. We believe that the throughput of the Click router directly corresponds to its CPU utilization as well; an unmodified Click router on a 100 Mbps connection used up about 23 to 25% of a CPU's time. Since increasing the probe rate causes throughput to decline, which causes a decrease in the Click router's workload, we believe that this results in an overall decline in CPU utilization as the probe rate increases.

6. ALTERNATIVE APPROACHES

Set of secrets. In our prototype implementation, we chose to have the prover store its secrets within a 4-way cuckoo hash table [7] in order to guarantee constant-time lookups (k hash functions) at the expense of $O(|\mathbb{S}_P^{R_i}|)$ space complexity. If space rather than computational power is the prover's bottleneck, the prover could distribute secrets of the form $(s_i, H_k(s_i), H_k(H_k(s_i)))$, where H_k is a keyed cryptographic hash function and k is a secret key known only to the prover. The prover then only has to remember k for all verifications within a generation. When a verifier tags a packet, it either embeds two random numbers or $(s_i, H_k(s_i))$. The prover identifies a pair as a shared secret by applying H_k to the first element and checking whether this equals the second element. Since only the prover knows k, no one else can distinguish a secret pair from a pair of random numbers. The prover responds by sending back $H_k(H_k(s_i))$ in a similar fashion as in the original protocol.

While conceptually attractive, this approach introduces cryptographic operations into the critical path of the prover's border routers. In our informal tests, we found that computing a SHA-1 hash over an 8-byte buffer on one of our Pentium III/933 router machines was approximately 60-70 times slower than the hash functions we used for our secret table [11]. Moreover, since verifiers cannot generate secrets on their own, entire sets of secrets have to be distributed during the setup phase and stored on the verifiers' routers.

Tagging packets. In our implementation, we embed 16-bit tag fragments in consecutive packets. This method is sensitive to packet re-ordering and forces the prover to maintain temporary route-specific state to rebuild the original tag.

An alternative is to use IP options to embed whole tags. This avoids reordering issues and removes any route-specific state from the prover. In addition, since IP options are variable-sized, multiple verifiers along a given route could simultaneously perform verification using the same packet. Adding IP options, however, increases the size of packets and may cause larger packets to become fragmented. More importantly, packets with IP options will be routed via the slow path on *all* routers, significantly impacting throughput.

7. CONCLUSIONS

We have presented a lightweight protocol for verifying consistency between Internet route advertisements and actual paths of data traffic. The main features of our protocol are its scalability and robustness against malicious routers on the path. At the cost of an infrequent offline setup phase, our protocol enables an autonomous system to prove both its and its predecessor's presence on the data path while handling high traffic loads and without maintaining a secure channel with each verifier. Our evaluation of the prototype implementation demonstrates that the impact of our protocol on router performance is small.

Acknowledgements. This material is based upon work supported by the National Science Foundation under grants CNS-0509033, CNS-0509338, IIS-0534198 and CyberTrust award 0430510, and the Department of Defense under Contract No. H98230-05-C-1650.

8. REFERENCES

[1] I. Avramopoulos and J. Rexford. Stealth probing: Efficient data-plane security for IP routing. In *Proc. USENIX Annual Technical Conference*, 2006.

[2] E. Billauer. Frandom. `http://frandom.sourceforge.net`, 2007.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[4] Cisco Systems. Cisco CRS-1 carrier routing system. `http://www.cisco.com/application/pdf/en/us/guest/products/ps5763/c1031/cdccont_0900aecd800f8118.pdf`.

[5] M. H. DeGroot. *Probability and Statistics*. Addison-Wesley, 1986.

[6] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[7] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. Space efficient hash tables with worst case constant access time. In *Proc. STACS*, 2003.

[8] S. Goldberg, D. Xiao, B. Barak, and J. Rexford. Measuring path quality in the presence of adversaries. `http://www.princeton.edu/~goldbe/FDFL-sc.pdf`, 2007.

[9] G. Goodell, W. Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin. Working around BGP: An incremental approach to improving security and accuracy of interdomain routing. In *Proc. NDSS*, 2003.

[10] Iperf. The TCP/UDP bandwidth measurement tool. `http://dast.nlanr.net/Projects/Iperf/`, 2005.

[11] B. Jenkins. Hash functions and block ciphers. `http://www.burtleburtle.net/bob/hash`, 2006.

[12] S. Kent, C. Lynn, and K. Seo. Secure Border Gateway protocol (Secure-BGP). *IEEE Journal on Selected Areas in Communications*, 18(4), 2000.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[14] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfigurations. In *Proc. SIGCOMM*, 2002.

[15] Z. Mao, J. Rexford, J. Wang, and R. Katz. Towards an accurate AS-level traceroute tool. In *Proc. SIGCOMM*, 2003.

[16] A. Mizrak, Y. Cheng, K. Marzullo, and S. Savage. Fatih: Detecting and isolating malicious routers. In *Proc. DSN*, 2005.

[17] V. Padmanabhan and D. Simon. Secure traceroute to detect faulty or malicious routing. *SIGCOMM Comp. Comm. Review*, 33(1):77–82, 2003.

[18] V. Paxson. End-to-end internet packet dynamics. *IEEE/ACM Trans. Netw.*, 7(3):277–292, 1999.

[19] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. *SIGCOMM Comp. Comm. Rev.*, 30(4):295–306, 2000.

[20] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer. Single-packet IP traceback. *IEEE/ACM Trans. Netw.*, 10(6):721–734, 2002.

[21] I. Stoica and H. Zhang. Providing guaranteed services without per-flow management. In *Proc. SIGCOMM*, 1999.

[22] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. Katz. Listen and Whisper: Security mechanisms for BGP. In *Proc. NSDI*, 2004.

[23] T. Wan, E. Kranakis, and P. van Oorschot. Pretty secure BGP (psBGP). In *Proc. NDSS*, 2005.