A System for Authenticated Policy-Compliant Routing

Barath Raghavan and Alex C. Snoeren
University of California, San Diego
{barath,snoeren}@cs.ucsd.edu

ABSTRACT

Internet end users and ISPs alike have little control over how packets are routed outside of their own AS, restricting their ability to achieve levels of performance, reliability, and utility that might otherwise be attained. While researchers have proposed a number of source-routing techniques to combat this limitation, there has thus far been no way for independent ASes to ensure that such traffic does not circumvent local traffic policies, nor to accurately determine the correct party to charge for forwarding the traffic.

We present Platypus, an authenticated source routing system built around the concept of network capabilities. Network capabilities allow for accountable, fine-grained path selection by cryptographically attesting to policy compliance at each hop along a source route. Capabilities can be composed to construct routes through multiple ASes and can be delegated to third parties. Platypus caters to the needs of both end users and ISPs: users gain the ability to pool their resources and select routes other than the default, while ISPs maintain control over where, when, and whose packets traverse their networks. We describe how Platypus can be used to address several well-known issues in wide-area routing at both the edge and the core, and evaluate its performance, security, and interactions with existing protocols. Our results show that incremental deployment of Platypus can achieve immediate gains.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Packet-switching networks; C.2.2 [**Network Protocols**]: Routing protocols

General Terms

Design, Security, Performance, Measurement

Keywords

Source routing, Authentication, Overlay networks, Capabilities

1. INTRODUCTION

Network operators and academic researchers alike recognize that today's wide-area Internet routing does not realize the full potential of the existing network infrastructure in terms of performance [31], reliability [1, 4, 20], or flexibility [12, 17, 40]. While a number of techniques for intelligent, source-controlled path selection have been proposed to improve end-to-end performance [31, 37], reliability [1, 4, 20, 41], and flexibility [10, 14, 17, 36, 40], they have proven problematic to deploy due to concerns about security and network instability. We attempt to address these issues in developing a scalable, authenticated, policy-compliant, wide-area source routing protocol.

We argue that many of the deficiencies of today's routing infrastructure are symptoms of the coupling of routing policy and routing mechanism [33]. In particular, today's primary wide-area routing protocol, the Border Gateway Protocol (BGP), is extraordinarily difficult to describe, analyze, or manage [24]. Autonomous systems (ASes) express their local routing policy during BGP route advertisement by affecting the routes that are chosen and exported to neighbors. Similarly, ASes often adjust a number of attributes on routes they accept from their neighbors according to local guidelines [27]. As a result, configuring BGP becomes an overly complex task, one for which the outcome is rarely certain. BGP's complexity affects Internet Service Providers (ISPs) and end users alike; ISPs struggle to understand and configure their networks while end users are left to wonder why end-to-end connectivity is so poor.

One approach to reducing this complexity is to consider an alternate approach to routing, where the issues of connectivity discovery and path selection are separated. Removing policy constraints from route discovery presents an opportunity for end users and edge networks: routes previously hidden by overly conservative policy filters can be revealed by ASes and traversed by packets. In this paper we consider one such approach based upon source routing. The key challenge becomes determining whether a particular source route is appropriate. ASes have no incentive to forward arbitrary traffic; currently they only wish to forward traffic for their customers or peers. We argue, however, that this is simply a poor approximation of the real goal: ASes want to forward traffic only if they are compensated for it. Henceforth, we will consider traffic *policy compliant* at a particular point in the network if the AS can identify the appropriate party to bill, and that party has been authorized by the AS to use the portion of the network in question.

We present the design and evaluation of Platypus, a source routing system that, like many source-routing protocols before it, can be used to implement efficient overlay forwarding, select among multiple ingress/egress routers, provide virtual AS multi-homing, and address many other common routing deficiencies. The key advantage of Platypus is its ability to ensure policy compliance during packet forwarding. Platypus enables packets to be stamped at the source as being policy compliant, reducing policy enforcement to stamp verification. Hence, Platypus allows for management of routing policy independent of both route export and path selection.

Platypus uses *network capabilities*, primitives that are placed within individual packets, to securely attest to the policy compliance of source routing requests. Network capabilities are i) trans-

SIGCOMM'04, Aug. 30–Sept. 3, 2004, Portland, Oregon, USA.

ferable: an entity can delegate capabilities to others, ii) composable: a packet may be accompanied by a set of capabilities, and iii) cryptographically authenticated. Capabilities can be issued by ASes to any parties they know how to bill. Each capability specifies a desired transit point (called a *waypoint*), a resource principal responsible for the traffic, and a stamp of authorization. By presenting a capability along with a routing request, end users and ISPs express their willingness to be held accountable for the traffic, and the included authorization ensures the policy compliance of the request.

In addition to its design, we also aim to understand how Platypus might be deployed in today's Internet. Incremental deployability is key in our setting, as it would be unreasonable to expect ASes to cooperate in the deployment of a system that affects *local* policy. To this end, we present results from wide-area measurements and performance evaluation of a prototype UNIX-based Platypus router, which indicate that incremental deployment of Platypus is feasible and may yield substantial benefit even using only a few routers.

2. OVERVIEW & APPLICATIONS

It is well known that multiple paths often exist between any two points in today's Internet. The central tenet of any source-routing scheme is that no single route will be best for all parties. Instead, sources should be empowered to select their own routes according to whatever criteria they determine. Protocols for efficient wide-area route discovery and selection, however, are beyond the scope of this paper. We assume that the network is configured (using BGP, for example) with a set of default routes and that certain motivated parties become aware of alternative paths, either through active probing [4, 35] or route discovery services [26]. Platypus builds on this basic infrastructure, allowing entities to select paths other than the default. Packets may specify a set of waypoints to be traversed on the way to a destination, but are *not* required to specify each router along the path. A source-routed packet is forwarded using default paths between the specified waypoints; an end-to-end path is therefore a concatenation of default paths.

Platypus is designed to be deployed selectively by ASes at choice locations in their networks. To support incremental deployment, Platypus waypoints are specified using routable IP addresses. When source routing a packet, the routing entity, which may be an end host or a device inside the network, encapsulates the payload and replaces the original destination IP address of the packet with the address of the first waypoint. The original destination IP address is stored in the packet for replacement at the last waypoint. When a Platypus packet arrives at a waypoint, the router updates the Platypus headers and forwards the packet on to the next waypoint.

2.1 Sample applications

We motivate the design of Platypus by describing several possible applications below. These examples are meant to be illustrative, not necessarily comprehensive.

2.1.1 Efficient overlay routing/On-demand transit

Consider the partial network topology shown in Figure 1. Nodes A, B, and C are all willing to cooperate to forward each other's traffic. Assume that A wishes to send a packet to B, but the default route $A \rightarrow R3 \rightarrow R4 \rightarrow B$ is unsatisfactory, perhaps because the link $R3 \leftrightarrow R4$ is congested or down. With prior overlay systems [4], A could use C as a transit point by tunneling its traffic directly to C, who would then forward it along to B. While effective at avoiding the bad link, this route is clearly sub-optimal for all involved, since:

1. C is forced to forward each packet itself, consuming both its bandwidth (in both directions) and processor resources. It would prefer that $R8$ forward the traffic instead; likewise, $R8$ would prefer that $R7$ forward the traffic.

2. Any path from A to B through $R7$ is likely suboptimal unless the $R5 \leftrightarrow R6$ link is congested.

3. If avoiding $R3 \leftrightarrow R4$ is the objective, an alternate route exists using the $R1 \leftrightarrow R2$ link. If C's ISP also owns $R1$ and $R2$, C should be able to authorize use of the link $R1 \leftrightarrow R2$.

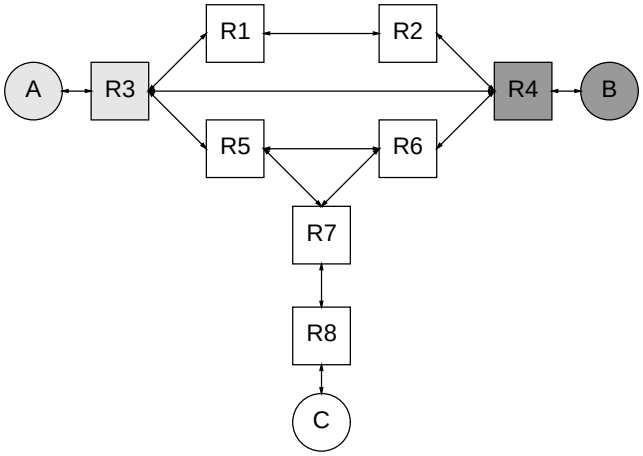


Figure 1: A simple network topology. Hosts A, B, and C all have different ISPs.

The first issue could be addressed by traditional source-routing schemes, requiring that A specify the route $R3 \rightarrow R5 \rightarrow R7 \rightarrow R6 \rightarrow R4 \rightarrow B$. The challenge is in communicating to C's ISP that such a route request is reasonable. In this case, assuming C's ISP is not a transit provider, it is permissible only because C is a customer of the ISP and is willing to be charged for A's traffic. With existing source-routing mechanisms, an AS cannot determine whether a forwarding request complies with local policy, and, if so, who to charge for the service. Currently, an AS assumes that packets should arrive at its border only if it advertised a route to their destinations. In our example, a packet destined for B should not arrive at $R5$ from $R3$; it should go directly to $R4$. Source-routed packets can obviously be made to explicitly transit any AS, violating this precondition. While ISPs can (and do) use filters to prevent unauthorized traffic from entering their network, filters can only act upon information contained within a packet—source and destination addresses, protocol, type of service, etc.—and current network location. These attributes are insufficient to determine policy compliance or the responsible party in this case. Nothing about the source-routed packet from A to B indicates C's cooperation (and resulting policy compliance).

In Platypus, C, by virtue of being a customer of its ISP, may have authority to source route through any of the ISP's routers. In that case, C's ISP would issue C a capability and a secret key that can be used to *stamp* packets. The capability would name C as the *resource principal*—the party responsible for all traffic bearing the capability. Platypus ensures the policy compliance of a given source route by requiring that source-routed packets contain a capability for each waypoint in a packet's source route. Because the secret key needed to stamp packets is known only to the indicated resource principal (or its associates), properly stamped packets certify their policy compliance and allow waypoints to appropriately account for usage.

We posit that ASes conduct *a priori* negotiations with customers and each other to determine mutually agreeable policies about who may source route traffic through which waypoints (similar to today's peering agreements [27]). Efficiently describing or constructing such policies is a complex problem on its own; we do not discuss it here. Instead, we assume the output of this process is a set of rights which can be encoded as a matrix of binary entries: for each waypoint in the network, a given resource principal may or may not forward traffic through it. Capabilities expire periodically and can be revoked, allowing ASes to dynamically update their policies.

Returning to our example, C could transfer its capability to A, allowing A to construct a source route that can alleviate all three issues, depending on the waypoint specified in the capability. If the capability specifies $R7$ as a waypoint, the first problem is solved. If, on the other hand, the waypoint simply refers to any router within C's ISP, the second problem is addressed automatically by the intra-AS routing protocol, which forwards the packet along the most efficient route from $R5$ (which would serve as the waypoint). Finally, if C were to request a capability specifically naming $R1$ as a next hop, even the third issue can be addressed.

While we have described A, B, and C as end hosts for simplicity, Platypus is designed to allow in-network stamping. Hence, each of these entities could correspond to entire ASes, allowing the example to be recast as a type of secondary transit, where C—a stub domain—can resell its transit privileges to other, non-adjacent stub domains without prior involvement of its provider.

2.1.2 Preferential egress points

Continuing to focus on ISPs, we observe that it is often the case that ISPs would like to select egress peering points based upon the peer injecting the traffic. However, since multiple upstream ASes often peer at the same ingress point of an ISP's network, it can be difficult to separate an individual AS's traffic to perform selective forwarding. Currently the only effective means of specifying egress points based upon upstream AS is through inter-provider MPLS, which to our knowledge is rarely deployed. Platypus can address this need, requiring cooperation only between the peering ASes. Upstream ASes can be issued capabilities with waypoints corresponding to the desired egress routers. These ASes stamp traffic with the appropriate capability at their peering router, thereby directing their traffic to the appropriate egress router.

2.1.3 Preferential ingress points

Multi-homed stub ASes often select multiple upstream providers and send different traffic through each depending on network conditions and destination—so-called policy routing. Unfortunately, a stub AS remains at the mercy of its upstream providers to control how incoming traffic arrives; there currently exists no widely deployed mechanism to affect ingress points [1]. Using Platypus, however, an AS could delegate multiple capabilities naming waypoints corresponding to its different upstream providers. Just as with toll-free phone numbers, a stub AS may be willing to be the resource principal responsible for incoming traffic if it can affect how that incoming traffic arrives. While the design of a mechanism for broadcasting capabilities and associated secret keys is outside the scope of this paper (although likely as simple as leveraging DNS or HTTP), Section 4.3 details how capabilities can be restricted to only allow traffic to be sent to a specified destination.

2.1.4 Virtual multi-homing

A stub AS with a single upstream connection is currently limited to the default routes of its provider. Without multi-homing, an AS is incapable of selecting backbone providers to carry its traffic—it must use the backbone selected by its upstream AS. With Platypus, however, a stub AS could request capabilities from providers of its choice, and place these on its out-bound traffic indicating which of its regional provider's upstream backbones to use for particular traffic—in effect making the AS virtually multi-homed. Thus, a stub AS could implement its own policy routing without the need for any configuration on the part of its upstream provider.

As a concrete example, suppose an AS, X, wishes to choose between two indirectly upstream providers A and B. X's ISP, Y, need not provide Platypus support. At the $X \leftrightarrow Y$ gateway, X classifies traffic it wishes to route through either A or B and stamps them with appropriate capabilities. Though Y doesn't support Platypus forwarding, it faithfully delivers packets to A's or B's edge routers, which are aware of Platypus headers, and, thus, deliver the packets as X desired. In such a scenario A and B clearly have a financial motivation to provide such a service since they can bill X, while X benefits by having choice in its indirect upstream providers, potentially providing fail-over or optimized routing. X's provider, Y, has no disincentive to allow Platypus-enabled packets to traverse its network since it has an already established relationship with X.

2.2 Challenges

As these examples demonstrate, source routing can be used to address a number of issues with the existing routing infrastructure. We believe, therefore, that the unavailability or limited deployment of source routing protocols stems not from a lack of utility, but, instead, from the omission of two key features: a mechanism for accountable and composable authorization, and the ability for ISPs to effectively manage link utilization. The need for authorization should be clear from the examples. The relationship to load management, however, is a bit more subtle. Recent research indicates that self-interested source routing can achieve performance gains even in wide deployment, but raises concerns about possible negative interactions with traffic engineering—highly reactive sources may make existing traffic engineering mechanisms ineffective by constantly changing fine-grained route requests [28].

3. NETWORK CAPABILITIES

Platypus addresses both of these issues through the use of network capabilities. Abstractly, a network capability is made up of two fields: a waypoint and a resource principal identifier. The waypoint specifies a topological network location through which the packet should be routed and the resource principal specifies the entity willing to be charged for the routing request. (For now, we will consider waypoints to correspond to a specific router within an AS. We return to evaluate how adjusting the topological granularity of Platypus waypoints in affects their utility in section 6.2.)

In Platypus, packets are stamped with a source-routing request by inserting a Platypus header immediately after the IP header of each packet and including some number of capabilities, encapsulating the existing payload. Figure 2 shows the Platypus header format with one capability attached. The header contains fields for the protocol version (currently 0), a set of bit flags (whose use is described in Section 4.2), a length field (specified in terms of 32-bit words), a pointer to the current capability (also in terms of 32-bit words), and an encapsulated protocol field to facilitate de-encapsulation. Capabilities are appended immediately after the Platypus header, and may be added by in-network stampers while the packet is in transit.

Since anyone can use a capability to forward packets through the specified waypoint and bill the indicated resource principal, Platypus must ensure that eavesdroppers watching packets in the network cannot use capabilities they observe in flight for their own packets. Similarly, attackers should not be able to modify capa-

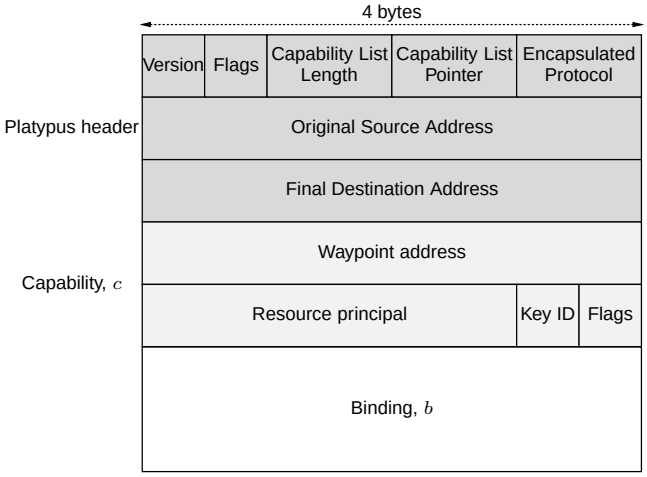


Figure 2: Platypus header format with a single capability and binding attached.

bilities or construct new ones that enable them to use waypoints for which they are not authorized, or to bill other resource principals. To prevent this, each capability in a packet is accompanied by a *binding* that cryptographically ensures the capability is valid and being used by the appropriate party. Bindings are a function of the capability, the packet contents, and a secret known only to the owner of the capability. When a Platypus packet arrives at a waypoint, the Platypus router validates the corresponding capability and its binding. If the capability/binding pair is valid, the router updates the waypoint pointer (indicating the packet has already passed through this waypoint), sets the packet's current destination IP to the waypoint field of the next capability in the capability list, replaces the current source IP with its own (to prevent ingress filters from dropping the packet), and forwards the packet on. If no additional capabilities remain, the router replaces the original destination address.

3.1 MAC-based authentication

Platypus prevents the forging of capabilities or their bindings through what is known as the "double MAC" trick [2], which we have proven to be secure if the underlying MAC is a pseudorandom function, as most modern MACs are believed to be. We define a secret temporal key, $s = \text{MAC}_k(c)$, generated from the capability, c, using a message authentication code (MAC) such as HMAC [18]. The MAC is keyed with k, the key of the specified waypoint. This value s is securely transferred to the resource principal (in a manner described in Section 4). In order to use a capability, an individual packet must be stamped with the capability and a binding, $b = \text{MAC}_s(\text{MASK}(P))$, where $\text{MASK}(P)$ is the invariant [13] contents of the packet (not including Platypus headers) with the real source and destination addresses substituted and the packet length field omitted. Both b and c are included in the packet, as shown in Figure 2. In this way, the binding is dependent upon both the secret key s and the packet's contents, and thus cannot be reused for other packets. Similarly, any changes to the capability c would render bindings computed with the secret temporal key s invalid.

Figure 3 presents pseudocode for Platypus packet verification and forwarding. To verify a packet's binding (and, therefore, capability), a Platypus router only needs the local waypoint key, k, since $b' = \text{MAC}_{\text{MAC}_k(c)}(\text{MASK}(P)) = \text{MAC}_s(\text{MASK}(P))$. If $b \neq b'$, either the capability or the binding (or both) has been forged and the packet should be discarded. An advantage of this construction is that the router needs to maintain only a constant amount of state

```
R: Revocation set, ID: Current key ID
PROCESS(P: Packet)
    c ← *(P.phdr.ptr)
    if |c.id−ID| > 1 or c ∈ R then
        ICMPERROR(P)
    s ← MAC_k(c.way‖c.rp‖GETTIME(c.id))
    b' ← MAC_s(MASK(P))
    if c.b = b' then
        ACCOUNT(c.rp, P)
        if P.phdr.src = 0 then
            P.phdr.src ← P.src
        P.phdr.ptr ← P.phdr.ptr + |c|
        if P.phdr.ptr ≥ P.phdr.len then
            P.dst ← P.phdr.dst
        else
            c ← *(P.phdr.ptr)
            P.dst ← c.way
        FORWARD(P)
    else
        ICMPERROR(P)
```

Figure 3: Pseudocode for Platypus forwarding. P is a packet, P.src is the packet's source IP address, and P.phdr is the Platypus header in which src (dst) is the source (destination) address, ptr is the pointer to the current capability and len is the length of the capability list. c is a capability, c.way is the its waypoint field, c.rp is its resource principal field, c.id is its key ID, and c.b is the binding. $\|$ denotes concatenation.

irrespective of the number of resource principals. In addition, rejected packets elicit ICMP responses to the sender to quell further invalid transmissions (subject to standard ICMP rate limiting).

3.2 Key expiration and timing

If temporal secret keys were never to expire, ASes would have no means to enforce changing policies—resource principals could use their capabilities forever. In addition, if a key were transferred to a third party or compromised, the resource principal would have no way to regain control over its associated capability. To address these issues, Platypus provides automatic key expiration. Once a temporal secret key expires, resource principals must retrieve a new one from the *key server*. To simplify the task of authenticating resource principals to the key server, we introduce the notion of a capability master key, c_k, which is shared between the resource principal and the key server. The capability master key is not used to generate capabilities or bindings, it is only needed to retrieve a new temporal secret key from the key server.

Platypus is designed to avoid the need for tight time synchronization between stamping parties and Platypus routers. Each capability includes a key identifier (key ID) which is a small (4-bit) integer that identifies the temporal secret used to compute each packet's binding. This key ID value changes on a regular basis (*e.g.*, every hour) and a new corresponding temporal secret generated. Since the key ID space is small, the key ID may wrap around often, yielding what would be identical temporal secrets if $s = \text{MAC}_k(c)$. We address this issue by incorporating the current time during generation of temporal secrets.[1] In this way, temporal secrets are guaranteed to be unique despite key ID wraparound.

[1] Specifically, for a given time t, where t is the seconds part of a 32-bit UNIX timestamp in UTC, and an expiration interval of 2^n, the corresponding key ID $i = (t \gg n)$ & 0xF. That is, the key ID is the last 4 bits of t after removal of the lower n bits; the key ID changes every 2^n seconds. To compute a temporal secret s as in Figure 3, a call to GETTIME$(i) = ((t \gg n)$ & 0xFFFFFFF0$)$ | i, which returns the time value that corresponds to the given key ID.

	Waypoint Key k	Revocation List R	Capability Master Key c_k	Temporal Secret Key s	Binding b
Key Server	●	●	●	○	
Platypus Router	●	●		○	○
Resource Principal			●	●	○
Trusted Third Party				●	○
Others					●

Table 1: Capability knowledge hierarchy. ● denotes that the value is/can be known, ○ indicates it is generated on the fly.

To ensure that both stamping agents and routers agree on the current key ID, capabilities are associated with a key expiration interval upon issuance. The length of the expiration interval presents a natural tradeoff between control and overhead—short expiration intervals provide fine-grained control over secrets, but require more frequent key lookup. Expiration intervals must be chosen based upon operational experience with Platypus to suit the needs of the issuing AS and its resource principals. Our only synchronization requirement is that stampers have clocks that do not drift on the order of the expiration interval. In addition, to allow for transitions between secrets, we consider 3 secrets to be valid at any time: those for the current, previous, and next key IDs. To combat clock drift between Platypus routers, we expect that the routers are loosely time synchronized using a standard service such as NTP [25].

3.3 Security

Security in Platypus is provided by the fact that not all parties have the information needed to bind known capabilities to new packets or create new, usable capabilities. Table 1 shows the types of information known to various parties. To generate a temporal secret key, a party must have the waypoint key, k, which is known only to the router and the router's key server. Binding a capability to a packet requires only the temporal secret key, s, which is generated based upon k and the current time. Knowledge of one capability's temporal secret key, however, does not allow a party to generate temporal secrets for others. Resource principals wishing to transfer rights for a particular waypoint to trusted third party can pass both the capability and corresponding temporal secret key.

While the capability can be passed in the clear, the temporal secret key must be communicated privately, ensuring that only the chosen third parties are able to receive it. These third parties can then use s to generate bindings to stamp their own packets. Others, including those sniffing packets on the network, can see capabilities and their bindings, but lack the secret s required to generate valid bindings. Periodic key expiration ensures that third parties cannot use temporal secrets indefinitely. In addition, any temporal secret key may be revoked by the resource principal through communication with the key server as will be described in Section 4.1.

Unfortunately, since bindings include almost all the invariant contents of a packet, intermediate nodes are restricted in power. For example, since the binding covers the payload (including TCP port numbers) Platypus packets are not compatible with port-altering network address translators (NATs), nor can they be fragmented. We do not consider the inability to fragment a significant limitation, as hosts typically perform path MTU discovery for all destinations. The NAT restriction, however, may be more significant. Any port-altering NATs traversed by Platypus packets on their way to a waypoint must be Platypus-aware. Once a packet has passed through its final Platypus waypoint, however, it may pass through NATs without ill effect. Similarly, packets may traverse any number of NATs before being stamped. Since most NATs are deployed at the edges of networks, the above suffices when packets are stamped inside the network. End hosts wishing to stamp their own packets, however, cannot be behind a port-altering NAT.

4. CAPABILITY MANAGEMENT

Platypus gains significant flexibility from the ability to transfer capabilities. Entities can collect capabilities from multiple resource principals, constructing source routes to which no single entity would otherwise have rights. We describe capability management in two phases: First, we discuss how resource principals obtain temporal secrets for their own capabilities. We then present two schemes for the restricted delegation of a resource principal's capabilities.

4.1 Distribution

To bootstrap the capability distribution process, we expect that each AS provides an interface (likely a Web server) through which resource principal accounts are established. This can occur in many ways. For example: the server and resource principal set up a secure channel (using SSL, for example), and, after negotiating payment, the server sends a resource principal ID, randomly generated capability master key c_k, and the capability information to the resource principal.

To look up the current temporal secret s associated with a capability, a resource principal generates a request by encoding the capability and a special request opcode as a string and prepends it to the key-lookup subdomain (specified during the bootstrap process) in a DNS TXT lookup request, which is routed by DNS to an appropriate key server. For example, a request for a capability issued by `ucsd.edu` with key-lookup subdomain `platypus.ucsd.edu` would be `<request>.platypus.ucsd.edu`. The DNS response is a similarly encoded DNS TXT record containing the temporal secret for the requested key ID encrypted under the capability master key. The resource principal decrypts and verifies the response, yielding the current temporal secret s for the specified capability.

The use of DNS for key lookup may seem clumsy; a more natural approach might be to contact the key server directly. To contact the server, however, a resource principal would have to first perform a DNS lookup for the key server and then transmit its lookup request, requiring multiple round trips. Instead, Platypus piggybacks the request for a key, shortening the lookup latency to about one RTT, allowing for extremely short expiration intervals. By using DNS to distribute keys, Platypus realizes caching, distributed authority, and failure resistance without having to build a separate key distribution infrastructure. In particular, Platypus key lookups are cacheable since requests are plain text and replies are encrypted under the capability master key for the requested capability. If multiple requests are made for the same shared capability, DNS caching will automatically decrease the load on the key server.

While expiration provides for coarse-grained control of temporal secrets, a resource principal may want to immediately revoke the current temporal secret when it suspects compromise. Platypus enables such revocation: to revoke a particular temporal secret, the resource principal computes the MAC of the capability and the current time under the capability master key and sends the {capability, time} pair, MAC, and the revocation opcode encoded as a DNS request. Platypus routers periodically receive updated revocation lists from their associated key servers and consult the revocation list

whenever validating capabilities. The revocation list for the current key ID is flushed before key ID rotation.

4.2 Reply capabilities

Protocols such as TCP work best when forward and reverse path characteristics are similar. In order to use Platypus source routes, however, both ends of a flow must have their own capabilities and perform their own routing. Fortunately, it may often be the case that a flow is for the benefit of only one party—an HTTP flow, for example—who may wish to be solely responsible for the flow. Platypus allows for resource principals to include a capability and its corresponding temporal secret as part of a packet stream for the recipient to use in response.

For concreteness, we describe reply capabilities in the context of an HTTP flow. Suppose the client possesses a capability to route through some Platypus router to reach a Web server. The client wishes to provide a capability to the server for reply packets back to the client. (Obviously, the server or some router near the server must support Platypus stamping to make this possible.) Platypus allows for an in-band exchange of capabilities and temporal secrets by conducting a Diffie-Hellman key exchange using a special reply capability flag in the Platypus header. We omit the details for space, but once the capability and corresponding secret is transferred to the server (requiring a three-way handshake; conveniently the secret can be included on the first TCP data segment—the HTTP GET—if no server authentication is required), the server simply uses it to stamp all packets destined for the client.

There must be some degree of trust in this relationship: the client must expect that the server is going to send it useful data if it is willing to be provide a capability for the traffic. However, the client may not wish to divulge its capability and temporal secret key entirely. In particular, the client may want to transfer the appropriate capabilities with a restriction that they be used only to send packets to its address. Thus, the server would only be able to use the restricted capability to route to the client, who would be able to detect any abuse. Such a restricted delegation mechanism is of use in a more general setting; we turn to this problem next, and use the fully restricted variant for reply capabilities.

4.3 General delegation

In general, a resource principal may want to specify a particular IP address prefix to which a third party may send packets using the principal's capability. Furthermore, the third party should be able to sub-delegate (specify a subnet of the previously delegated prefix) the capability without needing to contact the resource principal or key server. Platypus therefore allows the minting of *delegated capabilities*, which are derived from normal capabilities, but limited in their scope. To facilitate the use of delegated capabilities, we extend the capability format as follows. First, when a packet is stamped with a delegated capability, a bit is set in the flags field of the capability specifying that the capability is a delegated capability. Immediately before the associated binding, the stamper places the constraining prefix (a 32-bit value), the prefix length (an 8-bit value), and a delegation ID (a 24-bit value). These values allow a Platypus router to verify both that the binding is valid and that the destination of the packet is within the restricted prefix. In addition, the delegation ID can be used by the resource principal in conjunction with the ISP to track the use of delegated capabilities.

Table 2 presents two distinct protocols for constructing delegated capabilities: *chaining delegation* and *XOR delegation*. The two protocols address different design points and exploit a natural trade-off between the security of delegation and the complexity of delegated-capability verification; ASes would likely select one or the other depending on the delegation habits of their resource principals. Either scheme can be used to create reply capabilities by simply restricting the prefix to the client's IP address.

Chaining delegation is simply a multi-round variant of the double-MAC with similar security guarantees. Since chaining includes bits of the prefix itself, the chain values cannot be precomputed by routers. Thus, while requiring no additional state at the router, this scheme can require significant computation at Platypus routers (one invocation of F—which is likely to be implemented as a MAC in practice—for each bit in the delegated prefix).

Alternatively, XOR delegation is amenable to precomputation, allowing individual routers to trade off storage for per-packet computation, but admits a certain amount of collusion between third parties. By allowing precomputation of the sequence of values used to generate the key under which bindings are computed, verification of delegated capabilities can occur at roughly the same speed as non-delegated capabilities. For each temporal secret s there exists a sequence of pseudorandom values $d_{i,0}$ and $d_{i,1}$, one of which is selected according to whether bit i of the prefix is a 1 or 0. XORing these values yields a pseudorandom key under which MACs can be computed. Delegation is secure in that third parties who receive a d_p for some prefix p cannot compute the values for $d_{i,b}$ for $i \leq |p|$ since those values are computed using s_j values where $j < |p|$. Since XOR is commutative and intermediate values are distributed, the scheme is vulnerable to collusion between parties with different delegated prefixes for the same capability. For example, two parties with capabilities delegated from the same resource principal capability with prefixes of Hamming distance one can extract the $d_{i,b}$ values for the position i at which their prefixes differ.

5. IMPLEMENTATION

We have built prototype software components for UNIX that provide Platypus stamping, key distribution, and forwarding services. Each is described in turn below, as well as several issues that arise with respect to cryptographic primitives and protocol interactions.

5.1 Forwarding and key distribution

We have implemented Platypus forwarding functionality both as a user-space daemon process, `prd`, and a kernel module, `prkm`. We have written `prd` for both FreeBSD and Linux; `prkm` is currently available only for Linux 2.6. The user-space implementation, `prd`, works in conjunction with our key-distribution daemon, `pkd`; the two share a key database and run as separate threads of the same process. `pkd` services DNS key lookup and revoke requests for a delegated subdomain as described earlier. Our prototype of `prkm`, unlike `pkd`, currently does not support revocation.

The two Platypus router implementations differ mainly in the mechanisms they use to intercept Platypus packets. The user-space forwarding daemon, `prd`, captures Platypus packets using raw sockets, while the Linux kernel module registers itself with the IP stack as a protocol handler for Platypus protocol packets. After processing and validating any attached capabilities, the routers either reinject the packet into the local IP stack for delivery or forward it to the next Platypus hop or final destination. User-level packet capture and forwarding requires multiple user/kernel context switches, resulting in poor forwarding performance, thus motivating the need for an in-kernel implementation. `prkm` processes Platypus packets entirely inside the kernel. Upon a packet arrival, in the kernel soft-IRQ context, `prkm` verifies the packet; if the binding is valid, the packet is updated and forwarded. By binding interrupt handling for different network interfaces to different CPUs on a machine, `prkm` can provide good scaling across multiple processors.

	Chaining Delegation	XOR Delegation
Delegation	RP $s_0 \leftarrow s \oplus H(c.\texttt{id})$ **for** $i \leftarrow 1$ to $\|p\|$ $s_i \leftarrow F_{s_{i-1}}(p_i)$ $(s_{\|p\|}, p) \Longrightarrow$ TP	RP $s_0 \leftarrow s, d_p \leftarrow H(s) \oplus H(c.\texttt{id})$ **for** $i \leftarrow 1$ to $\|p\|$ $s_i \leftarrow F_{s_{i-1}}(i)$ $d_p \leftarrow d_p \oplus F_{s_{i-1}}(p_i \| i)$ $(d_p, s_{\|p\|}, p) \Longrightarrow$ TP
Sub-delegation	TP **for** $i \leftarrow (\|p\|+1)$ to $\|p'\|$ $s_i \leftarrow F_{s_{i-1}}(p'_i)$ $(s_{\|p'\|}, p') \Longrightarrow$ TP'	TP $d_{p'} \leftarrow d_p$ **for** $i \leftarrow (\|p\|+1)$ to $\|p'\|$ $s_i \leftarrow F_{s_{i-1}}(i)$ $d_{p'} \leftarrow d_{p'} \oplus F_{s_{i-1}}(p'_i \| i)$ $(d_{p'}, s_{\|p'\|}, p') \Longrightarrow$ TP'
Stamping	TP $b \leftarrow \text{MAC}_{s_{\|p\|}}(\text{MASK}(P))$	TP $b \leftarrow \text{MAC}_{d_p}(\text{MASK}(P))$
Verification	PR Verify $s_0 \leftarrow s \oplus H(c.\texttt{id})$ **for** $i \leftarrow 1$ to $\|p\|$ $s_i \leftarrow F_{s_{i-1}}(p_i)$ $b \stackrel{?}{=} \text{MAC}_{s_t}(P)$ $p \stackrel{?}{=}_{\|p\|} P.\texttt{dst}$	PR Precompute **for** $i \leftarrow 1$ to 32 $s_i \leftarrow F_{s_{i-1}}(i)$ $d_{i,0} \leftarrow F_{s_{i-1}}(0 \| i), d_{i,1} \leftarrow F_{s_{i-1}}(1 \| i)$ PR Verify $d_p \leftarrow H(s) \oplus H(c.\texttt{id}) \oplus \bigoplus_{i=0}^{\|p\|} d_{i,p_i}$ $b \stackrel{?}{=} \text{MAC}_{d_p}(P)$ $p \stackrel{?}{=}_{\|p\|} P.\texttt{dst}$

Table 2: Two protocols for IPv4 capability delegation. p represents a constraining IP prefix, p_i is the ith bit of p, $|p|$ denotes the length of p, p' is the prefix corresponding to the subnet of p to be delegated, $c.\texttt{id}$ is the delegation ID, F_k is a pseudorandom function keyed by k, and H is a hash function. RP is the resource principal, TP and TP' are third parties, and PR is the Platypus router. $\oplus$ denotes bit-wise XOR.

5.2 Stamping

To use Platypus, packets must be stamped with the capabilities appropriate for their selected route. Our current user-level implementation stamps (and routes) packets at the source using a user-level stamp daemon, `psd`. `psd` is implemented as a FreeBSD user-level process that receives stamp registration requests from application processes through a UNIX domain socket. Applications register requests in one of two fashions. First, they may pass `psd` a socket descriptor using `sendmsg()` requesting that all packets sent using that socket be stamped with a specified set of capabilities. Alternatively, processes may request that all packets sent on behalf of a particular user be routed and stamped in the specified fashion. By using a divert socket to intercept outgoing IP packets matching the set of registered stamp requests after kernel IP processing, `psd` can encapsulate, stamp, and resend packets transparent to in-kernel protocol stacks. In addition, we have developed a prototype Linux kernel module, `pskm`, that performs in-network stamping using a mechanism similar to that of `prkm`.

5.3 Cryptographic issues

Our prototype Platypus implementation uses UMAC, a MAC designed for efficient implementation on modern processors [7]. Unfortunately, UMAC requires a per-key setup phase that takes significantly longer than a single MAC computation. Hence, we maintain a capability cache with recent key IDs and their corresponding UMAC contexts. Cached contexts, if available, are used during binding computation, amortizing the key setup over many packets with the same capability. In a PC-based router, this context cache can easily be made large enough to cache most active resource principals: in our unoptimized implementation, each context uses only 316 bytes. Luckily, MACs well suited for hardware implementation have negligible key-setup time, so no cache would be needed.

The use of UMAC for capability verification raises two issues. First, UMAC requires unique nonces in addition to a key for each MAC computation; its designers suggest the use of a 32-bit or 64-bit counter, depending on the lifetime of the key. While nonces need not be private for security of the MAC, they must be unique across all MAC computations with a given key. Our prototype uses the IP ID field and 4 bytes of the encapsulated packet (corresponding to the TCP sequence number) to provide a 48-bit nonce for binding computation.[2] Also, due to the double MAC, the nonce used for the first MAC (to generate the key s) must be different from that of the second MAC (to generate bindings) since the former is fixed for the lifetime of s whereas the latter changes per packet. For the first MAC, we use the upper $(32-n)$ bits of the current time, where 2^n is the key expiration interval.

5.4 Protocol interactions

We have attempted to design around possible negative interactions between Platypus and existing protocols. In particular, proper ICMP delivery is complicated by source routing. Since ICMP responses can occur for many reasons, the appropriate recipient of such messages can be ambiguous. For example, should an ICMP time expired message be sent to the last Platypus waypoint in the source route, the stamper, or the original source? The cause of such expiration may be due to in-network stamping or other problems such as routing loops. Further complicating the matter, non-Platypus routers may generate ICMP responses for source-routed packets and send them to the last waypoint in the source route. In both of the two primary cases—end-host stamping and in-network stamping—the end-host perceives its Platypus-enabled connectivity to be the same as ordinary network connectivity, thus we send all ICMP packets back to the original source address. The first 64 bits of the Platypus header contain the original source address, enabling RFC-compliant routers to include the original source address in ICMP error response packets; Platypus routers forward such ICMP packets along to the source, subject to standard ICMP rate limiting.

Since Platypus uses DNS for key lookup requests, we must consider whether to stamp DNS packets themselves. We have chosen not to, for three reasons. First, DNS requests are typically local and thus will likely not benefit from Platypus-style routing. Second, initial Platypus key-lookup requests via DNS would need to be

[2] Some OSes zero the IP ID field for packets with the "don't fragment" bit set; this typically occurs after path MTU discovery. To ensure the uniqueness of nonces, a stamper may place a random IP ID in such packets before stamping.

Packet size	68 byte	348 byte	1500 byte
Packet processing			
Null	172 ns	173 ns	181 ns
UMAC	695 ns	998 ns	1908 ns
Destination cache lookup	289 ns		
IP hdr build and verify	145 ns		
Packet transmission	1480 ns	1482 ns	1493 ns

Table 3: Micro-benchmarks for `prkm`. All times are as measured by the CPU cycle counter.

stamped with a valid key, creating a bootstrapping problem. Finally, stamping DNS packets would require running a Platypus router at DNS servers or transparently de-encapsulating DNS requests in the network, complicating deployment.

6. EVALUATION

In this section we consider both the performance of our prototype router and stamper, and, more importantly, how the effectiveness of source routing is impacted by waypoint granularity.

6.1 Forwarding and stamping performance

Our experimental testbed consists of a central Linux-based router that performs both forwarding and stamping and several load generators connected through a gigabit Ethernet switch. The server is configured with two 64-bit, 2.2-GHz AMD Opteron 248 processors, two GB of PC2700 DDR memory, and three Intel Pro/1000 XT gigabit NICs; our tests used two of the NICs installed on a 100-Mhz, 64-bit PCI-X bus. The load generating machines have 1.1-GHz Pentium III processors and Intel Pro/1000 XT gigabit NICs.

First we consider the absolute performance of forwarding and stamping. Figure 4 compares the performance of Linux's in-kernel IP forwarding to `prkm`'s forwarding performance and `pskm`'s stamping performance for worst-case (minimum-size) packets. For forwarding tests, the load generators each direct identical 68-byte (20-byte IP header + 28-byte Platypus header + 20-byte TCP header, excluding the Ethernet header) Platypus packets at the router which validates the bindings and forwards the packets to the indicated waypoint. For stamping, the load generators send 40-byte packets which are stamped and forwarded by the router (by insertion of the 28-byte Platypus header with a capability and binding). To increase the offered load in a controlled fashion, we first saturate one router interface and then load the two interfaces at equal levels.

As seen in the figure, `prkm` is capable of forwarding packets with full UMAC authentication at a maximum loss-free forwarding rate of approximately 767 Kpps (using a warm UMAC context cache; initializing the context takes 41.3 μs), which is only slightly less than the performance of native Linux. To help calibrate for the fact that the kernel's forwarding code is more streamlined than that of `prkm`, we plot the performance of the `prkm` forwarding path without verification (labeled *null forwarding* in the figure). The results indicate that a significant portion of the performance degradation is due to factors other than capability verification. When forwarding MTU (1500-byte) packets, `prkm` is able to fully validate at approximately 2.5 Gbps without loss. Stamping performance is slightly worse: `pskm` is capable of loss-free stamping at approximately 633 Kpps. These results indicate that Platypus software routers and in-network stampers can yield good performance on modern hardware, enabling low-cost deployment of Platypus.

In addition to absolute forwarding numbers, we measured the amount of time actually spent validating bindings, as this latency may be observed by end hosts (as opposed to forwarding performance, which is largely a concern of the ISP). Table 3 shows micro-benchmarks of `prkm` in its several stages. We performed these measurements by averaging three runs of `prkm` forwarding 1,000,000 packets spaced 16 μs apart. For tests in which packet size affected performance, we benchmarked forwarding of 68-byte, 348-byte, and 1500-byte packets (including IP and Platypus headers, but excluding Ethernet headers), which correspond to minimum, moderate, and maximum packet sizes. Packets arrive at `prkm`'s packet handler from the Linux `ip_local_deliver` routine which is tasked with passing packets to the appropriate protocol handler. Packet processing includes time to parse the packet headers, verify the binding, and update Platypus headers. Destination cache lookup includes retrieval of a `dst_entry` structure, and IP header building and verification includes time to place a new IP header on the packet. Finally, packet transmission time includes time until the packet is queued for transmission by the device. While the validation time is highly dependent upon packet size, it is less than other overheads even for large packets. In the worst case, binding validation adds less than two μs to forwarding latency.

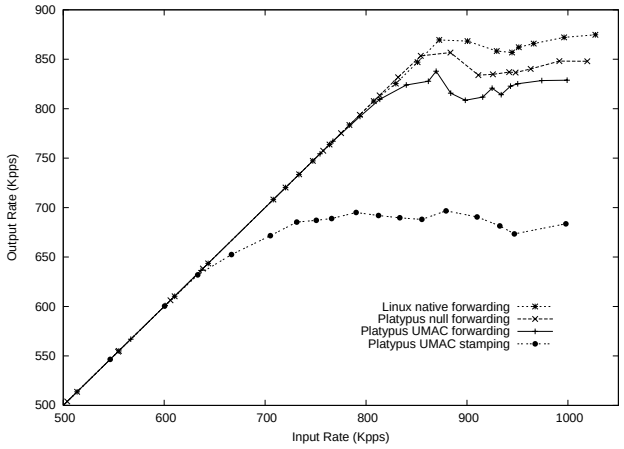
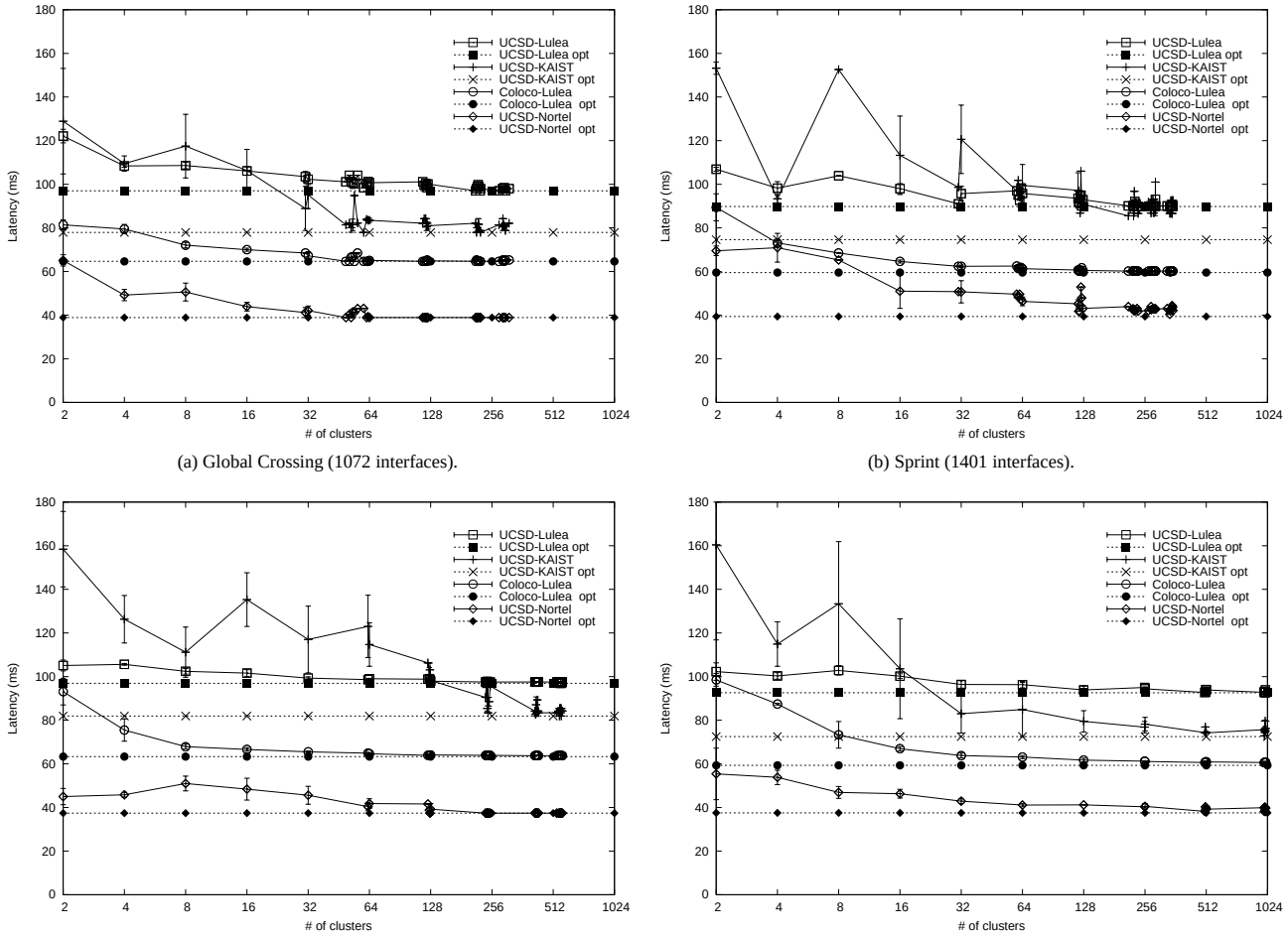


Figure 4: 68-byte packet forwarding/stamping performance.

6.2 Waypoint granularity

We now consider the impact of waypoint granularity on the effectiveness of Platypus-like source routing. Clearly, the finer the waypoint granularity, the more control Platypus can assert over a packet's path. Intuitively, however, intra-AS traffic engineering goals are likely best met by exporting only coarse-grained waypoints to external entities. While we lack sufficient information (such as actual AS topologies with traffic matrices) to address the traffic engineering issues, we attempt in this section to provide some insight into the level of granularity necessary to effectively impact a specific end-to-end path characteristic.

We consider clustering the routers into groups which could be represented by a single Platypus waypoint. In particular, we study the impact on end-to-end one-way path latency of routing indirectly through a waypoint of varying granularity. Previous research indicates that it is often possible to achieve significant performance improvements by inserting one level of indirection in a packet's route [4, 31]. However, for deployment reasons, we would like to know how *well chosen* an indirection point must be to provide substantial benefit. Thus, we consider how the best achievable path latency increases as waypoint granularity is reduced. Intuitively, since POPs represent a collection of routers in a region, and networks are dense near large cities and sparse elsewhere, similarly performing routers can be naturally clustered together. It may be sufficient to place Platypus routers in only a few locations, as speed



(a) Global Crossing (1072 interfaces).

(b) Sprint (1401 interfaces).

(c) Qwest (2022 interfaces).

(d) UUNet (8591 interfaces).

Figure 5: Impact of cluster size on indirection effectiveness. For each ISP, we vary the number of clusters generated based upon observed latencies between the two specified measurement points and every known router interface. For each indicated source/destination pair, we plot the measured one-way path latency using the ISP's optimal indirection router (opt) against the calculated path latency through the center of the optimal cluster. Data points represent averages; ten different clusterings were generated for each k-means input size. Error bars show the standard deviation.

of light delays comprise most of the delay seen by packets in uncongested wide-area backbones. Thus, multiple, local waypoints would not significantly affect latency (but, conversely, might be useful for load balancing, for example). Note that we are not arguing that path latency is the most important metric of interest, nor that improving path latency is the best application of Platypus. Rather, we claim only that latency is a relatively easily measured and well-understood path property that provides initial insight.

We begin by considering the router IP addresses reported by the Skitter project [9] for four major international ISPs: UUNet, Sprint, Qwest, and Global Crossing. We then selected five geographically diverse monitoring locations in the RON testbed [4], UC San Diego, Nortel (Nepean, Ontario, Canada), Coloco (Laurel, Maryland), Lulea (Sweden), and KAIST (Korea). From each monitoring location, we used ICMP timestamp probes to measure both the forward and reverse path latencies for each known router interface of the ISP in question [23]. This set of measurements was collected over a period of six days between January 22–27, 2004. We obtained approximately 240 measurements for each location/router pair and use the mean value. With this data, we compute a one-way, indirect end-to-end path delay between any two monitoring locations through each router interface.

To study the utility of various waypoint granularities, we generated different sized router clusters based upon the observed path latencies. We then calculated the end-to-end path latency between each pair of observation points using the router closest to center of the optimal cluster as the indirection point and compared this to the latency through the optimal router interface (which may or may not be a member of the optimal cluster). Figure 5 shows the results for each of the four ISPs studied. As expected, the more clusters (corresponding to a finer waypoint granularity), the closer the performance of the optimal cluster comes to performance of the optimal router. Somewhat surprisingly, however, the best cluster centers approach the optimal at a relatively small number of clusters even for UUNet, an extremely large ISP. These results indicate that a small number of indirection points are likely sufficient for substantial benefit; this applies equally to Platypus and any overlay or source-routing system. How waypoint granularity affects other metrics remains an open question for future work, but the low number of clusters required to achieve near-optimal latencies give reason for optimism.

We note in parting that the clusters we used are unlikely to result in the best performance, or even necessarily make operational sense, so we expect that intelligent placement would require even

fewer waypoints than our results indicate. We obtained our clusters by running the k-means [22] clustering algorithm over the latencies observed from the Coloco and Lulea measurement points. Each k-means run was given an argument n representing the desired number of clusters; during the run, the algorithm selected n clusters randomly and proceeded to improve or discard the clusters. By adjusting n we could vary the number of clusters requested, but had no control over the exact number of clusters generated or which routers would be clustered together. As such, we consider these results as an upper bound of achievable cluster-path latency.

7. DISCUSSION

Platypus represents one approach to capability-based source routing. During its design, we have considered issues of performance, security, accounting, and the effect of source-routed traffic upon the network. In this section we discuss these considerations.

7.1 Route setup

In its current incarnation, Platypus represents an extreme in terms of routing flexibility: each packet can be routed independently. A less radical approach observes that it is often the case that packets are part of a larger series of packets, or flow. Further, existing transport protocols like TCP are typically more effective when all packets experience similar path characteristics. Hence, all packets in a flow should generally be directed along the same route. It would suffice, then, to specify the desired route once per flow. Implementing such an RSVP-like scheme is tricky, however, as packets must be labeled as belonging to the same flow, initial packets in the flow could be lost, routers may reboot and lose flow state, etc. Nevertheless, existing hardware is extremely efficient at switching packets along previously configured routes. Hence, we are interested in developing ways for Platypus routers to cache forwarding directives for traffic flows. In particular, we are optimistic that we can harness the existing MPLS [29] label swapping support in deployed routers to implement a great deal of the Platypus forwarding functionality. By decoupling verification from forwarding, and offloading the initial verification and path setup to a dedicated Platypus router, it may be possible to switch Platypus packets using MPLS while retaining many of the features of our initial design.

7.2 Distributed accounting

While Platypus specifies a resource principal per capability, we have yet to discuss how accounting would actually be implemented. Fine-grained flow accounting is an established problem in other contexts, but Platypus complicates the use of several common approaches. For example, many end hosts receive flat-rate pricing for their Internet service. ISPs can provide this service with bounded risk because the rate at which an end host can inject packets into the network is limited by the capacity of its access link. More sophisticated pricing plans may depend on the actual utilization, which requires the ISP to meter a customer's traffic, but such metering can be done at the customer's access link.

In Platypus, however, a customer may authorize third parties to inject packets into its ISP as part of a source route. Any accounting scheme that only charges customers for packets that traverse their access link clearly will not properly account for the customer's additional use. A straightforward approach would maintain counters for each resource principal at all Platypus routers within an AS, and bill for the total consumption. While auditing challenges may dissuade many ISPs from per-packet accounting, aggregate rate limiting is likely to be needed to support those customers that wish to pay a flat rate for a fixed amount of bandwidth. In order to implement such a pricing model in Platypus, an AS must have some way to restrict bandwidth consumption for a particular resource principal at one or more routers. One possible approach to this problem is to construct a distributed token bucket that limits the aggregate rate of all packets with a given resource principal identifier.[3]

7.3 Replay attacks

In the context of rate-based accounting, a simple model in which resource principals can use a fixed, aggregate bandwidth will likely suffice. However, while (modulo cryptographic hardness assumptions) packet bindings cannot be forged, they may be replayed by an adversary, who may wish to waste a resource principal's limited bandwidth for a given capability. Since capabilities expire periodically, a natural countermeasure to replay attacks is to track packets that traverse a router within some time window and only count each distinct packet once. A Bloom filter allows for tracking of packets in such a way, but may fill up over time, resulting in false positives. This issue can be addressed by maintaining a small circular array of Bloom filters which are cleared as they fill up [2, 32]. While an adversary may be able to log all packets and replay them after the corresponding Bloom filter is emptied, if the filters are emptied only at key expiration intervals, stored packets cannot be replayed.

7.4 Scalability

While we have thus far only addressed the deployment of Platypus in several limited settings, the system's potential scalability in real-world deployment is of obvious interest. In this section we discuss the scalability of our current design.

Careful selection of MAC algorithms is crucial for peak verification performance. We use UMAC in our software implementation, but expect PMAC would be selected for hardware implementations. While we do not have raw figures on its performance, its inherently parallelizable design makes PMAC is ideal for hardware implementation [8]. Hardware implementations of AES already achieve raw throughput of 48 Gbps [30], giving reason to believe that hardware can be built to perform PMAC computation at high speeds. Since MAC computations are done with local information only, capability issuers can choose a MAC algorithm appropriate to their forwarding hardware or software.

Platypus's double-MAC design requires constant state for capability verification, regardless of the number of resource principals. ISPs may wish to keep additional accounting state for billing purposes, however. In the extreme case of per-packet billing, an ISP would need to keep a packet counter corresponding to each resource principal. While deployments of Platypus in the core may only need to handle a few thousand resource principals (for example, UUNet's 2,569 peers [34] may each represent a principal), deployments for a broadband ISP may have many more (the largest of which, Comcast, currently has 5.7 million customers [21]). Recent work has shown that approximate counters with bounded error can be maintained per flow at very high speeds (OC-768) [19].

We contend that Platypus key management can also be scaled to support large numbers of resource principals. For key distribution, it is unlikely that all requests will arrive exactly at key ID-change boundaries, since Platypus does not require tight time synchronization between resource principals and routers. Even in such an unlikely event, Platypus key servers need only perform two MAC and one block-cipher calls for each request; servicing ten million such requests in one second is well within the limits of approximately 20 well-provisioned key servers. Furthermore, since key lookup requests and responses are small, each lookup requires only one packet receipt and transmission on the part of a key server.

[3] To our knowledge, while work exists on distributed counting [39], none exists on distributed token buckets; we are actively investigating how one might be designed.

Key servers must periodically distribute revocation lists to Platypus routers; while distribution can occur off the critical path, lookup cannot, so revocation lists must be stored in high-speed memory. In our current design, each revocation entry is twelve bytes, so a 16-MB SRAM chip could store just under 1.4 million revoked capabilities. In the case of Comcast, that would correspond to almost a quarter of all its users' capabilities in any given expiration interval—a fraction much larger than we expect in practice.

7.5 Traffic engineering

Conventional wisdom holds that widespread source routing deployment would complicate traffic-engineering efforts. While there admittedly is cause for concern, we have reasons for optimism. Recent simulations by Qiu *et al.* show that while source-routed traffic can have deleterious interactions with intra-AS traffic engineering mechanisms in extreme cases, certain techniques may be able to mitigate these effects [28]. In their studies, however, source-routed traffic was capable of completely specifying intra-AS paths. Our measurements indicate that such fine-grained intra-AS hop selection may not be necessary; hence, we expect that Platypus can be deployed in concert with existing traffic engineering techniques. Furthermore, while we have thus far equated waypoints and physical router interfaces, waypoints can be more flexible in practice. Our design for Platypus is meant to allow ISPs to specify any globally routable IP address within their IP space as a Platypus waypoint and dynamically adjust the actual (set of) internal router(s) to which the IP corresponds in response to traffic load. We intend to explore this expanded ISP control in future work.

Independent of its interaction with traditional traffic engineering, Platypus opens up a new dimension for traffic provisioning: time. Routing in today's Internet has no temporal dimension—the advertisement of a route makes it immediately available. With Platypus, however, routes may have time-limited availability; that is, a route is available only when users possess the correct temporal secrets. By appropriately choosing expiration intervals and expressing route selection policy upon key lookup, ISPs can control the temporal aspects of traffic flow; in this way, Platypus may even serve to help achieve traffic engineering goals. While it is technically possible to implement a similar scheme using BGP, it has been shown that rapid, repeated announcement and withdrawal of routes can have a destabilizing effect on the routing system [20].

8. RELATED WORK

Source routing has been included as a feature in many Internet architectures over the years. For example, Nimrod [10] defined mechanisms for packets to be forwarded in both flow-based and source-routed, per-packet fashions. Similarly, IPv6 provides support for the source demand routing protocol, SDRP [14]. SDRP allows for hosts to specify a strict or loose source route of ASes or IP addresses through which to route a packet. More recently, Yang described a new addressing architecture called NIRA [40] with the explicit goal of providing AS-level source routing. NIRA path selection consists of two stages: an initial discovery phase followed by an availability phase in which a host determines the quality of a particular route. A contemporary proposal, BANANAS, allows for explicit path selection in a multi-path environment, but does not allow for the insertion of arbitrary intermediate hops [38]. None of these proposals, however, have addressed the need to verify policy compliance of the specified route on the forwarding plane. To the best of our knowledge, we are the first to present a fully decentralized, authenticated source-routing architecture.

Frustrated with the lack of control provided by current wide-area Internet routing, researchers have proposed circumventing it entirely by forwarding packets between end hosts in an effort to construct routes with more desirable path characteristics [4, 31]. Unfortunately, the effectiveness of any overlay-based approach is fundamentally limited by both the number and the locations of the hosts involved in the overlay. We believe Platypus addresses both of these issues: overlay networks can view far away Platypus routers as additional members of the overlay and use nearby Platypus routers to increase the efficiency of their forwarding mechanisms.

Stoica *et al.* suggest that indirection be explicitly supported as an overlay network primitive; in the Internet Indirection Infrastructure (*i*3) packets may include a set of indirection points through which they wish to be forwarded [36]. Unlike Platypus waypoints, however, *i*3 IDs specify logical entities, not necessarily network routing hops. Each ID is associated with one or more application-installed triggers that can involve arbitrary packet processing; there are no guarantees about the topological location of the overlay node(s) responsible for a particular ID.

Packet-level authentication credentials have been suggested in a number of other contexts. IPsec-enabled packets may contain an authentication header with information similar to a network capability [6], except without a routing request. In order to verify authentication headers, however, IPsec routers must hold one key for each source, far more than with Platypus. Per-packet authenticators have also been proposed to prevent DoS attacks [3, 5]; it would be straightforward to implement a similar scheme using Platypus. Perhaps the most closely related use is due to Estrin *et al.*, who introduced the notion of visas that confer rights of exit from one organization and entry into another [15]. Stateless visas provide a mechanism for per-packet authentication between two independent organizations, but not for expressing routing requests. Visas are the result of a bilateral agreement between a packet's source and destination; each packet contains exactly two visas—one for the source organization and one for the destination. In contrast, network capabilities are concerned with authentication and routing through intermediate ASes. In a subsequent paper [16], the authors also considered implementing preventative security measures within Clark's policy routing framework [11].

9. CONCLUSIONS & FUTURE WORK

Capabilities are well known in the operating systems literature, but have failed to catch on in many mainstream systems, likely because they are perceived as too heavyweight a mechanism to address the relatively simple access problems of single-user systems. In contrast, we believe capabilities are extremely well-suited for use in wide-area Internet routing. Unlike today's PCs, which typically are used by at most a small number of users with similar goals and policy constraints, the Internet serves an extremely large number of users with an even larger number of motivations, all attempting to simultaneously share widely distributed resources. Most importantly, there exists no single arbiter (for example, a system administrator or user logged in at the console) who can make informed access decisions.

Looking forward, while much work has gone into understanding existing Internet routing policy and describing how to specify it better, we believe that much of the complexity of Internet routing policy stems from inflexibility of existing routing protocols. We aim to study how one might implement inter-AS traffic engineering policies through capability pricing strategies. For example, an AS with multiple peering routers that wishes to encourage load balancing may be able to do so through variable pricing of capabilities for the corresponding Platypus waypoints. While properly modeling the self-interested behavior of external entities may be difficult, we are hopeful that this challenge is simplified by the direct mapping

between Platypus waypoints and path selection (as compared, for example, to the intricate interactions of various BGP parameters).

Acknowledgments

We thank Alvin AuYoung, Mihir Bellare, Nick Feamster, Ratul Mahajan, Daniele Micciancio, Travis Newhouse, Saurabh Panjwani, Sriram Ramabhadran, Jennifer Rexford, Chris Tuttle, Amin Vahdat, and David Wetherall for helpful discussions and feedback. We are indebted to Neil Alldrin for his help with k-means clustering and to David Andersen for the use of the RON testbed. Finally, we would like to thank our shepherd, Ken Calvert, and the anonymous reviewers for their comments. This work is supported in part by the National Science Foundation (CNS-0347949).

10. REFERENCES

[1] S. Agarwal, C.-N. Chuah, and R. H. Katz. OPCA: Robust interdomain policy routing and traffic control. In *Proc. IEEE OPENARCH*, June 2002.

[2] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. G. Andersen, M. Burrows, T. Mann, and C. A. Thekkath. Block-level security for network-attached disks. In *Proc. USENIX FAST*, Apr. 2003.

[3] D. G. Andersen. Mayday: Distributed filtering for internet services. In *Proc. USITS*, Mar. 2003.

[4] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. T. Morris. Resilient overlay networks. In *Proc. ACM SOSP*, Oct. 2001.

[5] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. In *Proc. HotNets*, Nov. 2003.

[6] R. Atkinson. Security architecture for the Internet protocol. RFC 1825, IETF, Aug. 1995.

[7] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. *Advances in Cryptology – CRYPTO '99. LNCS*, 1666, 1999.

[8] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. *Advances in Cryptology – EUROCRYPT '02. LNCS*, 2332, 2002.

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

[10] I. Castiñeyra, N. Chiappa, and M. Steenstrup. The Nimrod routing architecture. RFC 1992, IETF, Aug. 1996.

[11] D. D. Clark. Policy routing in Internet protocols. RFC 1102, IETF, May 1989.

[12] D. D. Clark, J. Wroclawski, K. R. Sollins, and R. Braden. Tussle in cyberspace: Defining tomorrow's Internet. In *Proc. ACM SIGCOMM*, Aug. 2002.

[13] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proc. ACM SIGCOMM*, Aug. 2000.

[14] D. Estrin, T. Li, Y. Rekhter, K. Varadhan, and D. Zappala. Source demand routing: Packet format and forwarding specification. RFC 1940, IETF, May 1996.

[15] D. Estrin, J. C. Mogul, and G. Tsudik. Visa protocols for controlling interorganizational datagram flow. *IEEE J. SAC*, 7(4), May 1989.

[16] D. Estrin and G. Tsudik. Security issues in policy routing. In *Proc. IEEE Symposium on Security and Privacy*, May 1989.

[17] G. Huston. Commentary on inter-domain routing in the Internet. RFC 3221, IETF, Dec. 2001.

[18] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, IETF, Feb. 1997.

[19] A. Kumar, J. Xu, L. Li, J. Wang, and O. Spatschek. Space-code Bloom filter for efficient per-flow traffic measurement. In *Proc. IEEE Infocom*, Mar. 2004.

[20] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet routing convergence. *IEEE/ACM ToN*, 9(3), June 2001.

[21] Leichtman Research Group. A record 2.3 million add broadband in first quarter of 2004, May 2004.

[22] J. B. MacQueen. On convergence of k-means and partitions with minimum average variance. *Ann. Math. Stat.*, 36, 1965.

[23] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *Proc. ACM SOSP*, Oct. 2003.

[24] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM*, Aug. 2002.

[25] D. L. Mills. A brief history of NTP time: Memoirs of an Internet timekeeper. *SIGCOMM CCR*, 33(2), 2003.

[26] A. Nakao, L. L. Peterson, and A. Bavier. A routing underlay for overlay networks. In *Proc. ACM SIGCOMM*, Aug. 2003.

[27] W. B. Norton. Internet service providers and peering. In *Proc. NANOG*, June 2000.

[28] L. Qiu, Y. R. Yang, Y. Zhang, and S. Shenker. On selfish routing in Internet-like environments. In *Proc. ACM SIGCOMM*, Aug. 2003.

[29] E. C. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. RFC 3031, IETF, Jan. 2001.

[30] B. Sanzone, D. Katz, D. Asher, D. Carlson, G. Bouchard, M. Bertone, M. Hussain, R. Kessler, and T. Hummel. NITROX II: A family of in-line security processors. In *Proc. IEEE Hot Chips*, Aug. 2003.

[31] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *Proc. ACM SIGCOMM*, Sept. 1999.

[32] 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 ToN*, 10(6), Dec. 2002.

[33] A. C. Snoeren and B. Raghavan. Decoupling policy from mechanism in Internet routing. In *Proc. HotNets*, Nov. 2003.

[34] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *Proc. ACM SIGCOMM*, Aug. 2003.

[35] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, Aug. 2002.

[36] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proc. ACM SIGCOMM*, Aug. 2002.

[37] I. Stoica and H. Zhang. LIRA: An approach for service differentiation in the Internet. In *Proc. NOSSDAV*, June 1998.

[38] H. Tahilramani Kaur, S. Kalyanaraman, A. Weiss, S. Kanwar, and A. Gandhi. BANANAS: An evolutionary framework for explicit and multipath routing in the Internet. In *Proc. ACM SIGCOMM FDNA*, Aug. 2003.

[39] R. Wattenhofer and P. Widmayer. An inherent bottleneck in distributed counting. In *Proc. ACM PODC*, Aug. 1997.

[40] X. Yang. NIRA: A new Internet routing architecture. In *Proc. ACM SIGCOMM FDNA*, Aug. 2003.

[41] D. Zhu, M. Gritter, and D. R. Cheriton. Feedback based routing. In *Proc. HotNets*, Oct. 2002.