The design and implementation of a policy framework for the future Internet

Jad Naous[†], Arun Seehra[‡], Michael Walfish[‡], David Mazières[†], Antonio Nicolosi[§], and Scott Shenker[¶]
[†]Stanford [‡]UT Austin [§]Stevens Institute of Technology [¶]UC Berkeley, ICSI

Abstract

Policy has become an important factor in network design, and there is now a bewildering bevy of architectural proposals. Each one aims at a different set of policy goals, and we don't know which one is right. This paper's animating assumption is that we can't predict the future policy requirements of the Internet so should instead seek the most general policy framework we can possibly implement. To that end, we articulate a general policy principle and describe the design, implementation in hardware, and evaluation of its enforcing mechanism.

1 Introduction

This paper is about the Internet's future, but we begin with its past. The history of network routing began as a *topological* problem: how does one find the shortest paths in a graph ([19])? However, with the advent of domain-based Internet routing, *policy* became an important consideration. In fact, policy concerns were embedded in the 1989 requirements document (RFC 1126) that set the groundwork for the first version of BGP:

> *Those resources used by (and available for) routing are to be allowed autonomous control by those administrative entities which own or operate them. Specifically, each controlling administration should be allowed to establish and maintain policies regarding the use of a given routing resource.* [34]

Embodying this principle, BGP allows each domain to unilaterally decide which routes it accepts and exports based on the full AS-level path.

Provider control is not limited to the control plane; providers have imposed usage limits and blocked certain types of traffic that they believe would be injurious to their or other networks.

Moreover, ASes are not the only stakeholders in the Internet. There have been many calls for granting sources some control over their packets' paths (see, for example, [9, 20, 23, 26, 29, 41, 52, 53]). The reasons vary from performance (letting sources find the best quality paths) to preference (letting sources avoid providers they don't trust) to price (letting sources find the cheapest paths).

For exactly the same reasons, receivers too have an interest in controlling the path of their incoming packets. Receivers also care *who* is sending them packets and may wish to allow only a subset of incoming flows (e.g., when under attack, accept packets only from customers).

While all of the above policy considerations seem natural, it isn't clear how to balance the concerns of the various stakeholders or which of these considerations, when they are in conflict, should prevail. Indeed, this uncertainty leads us to the question this paper tries to address: what policy framework should we adopt in a future Internet architecture? This question is one of both policy and mechanism: what policy considerations should the architecture support, and can we build a mechanism to support those considerations?

1.1 The nature of policy

Judging by the bevy of architectural proposals that support policy-oriented features such as interdomain policies, source selection of routes, and interposition of middleboxes by endpoints, there appears to be consensus that the various stakeholders have the right to exert some control over their flows, and that these considerations should be reflected in a future Internet architecture. Table 1 lists many, but by no means all, of these proposals. As the table makes clear, while the union of policy considerations is large, the intersection is small: each proposal generally supports only a particular subset of stakeholder control.

As a community striving to design the future Internet, we have two choices:

- Choose one subset of policy considerations and bet that it will be sufficient to meet all policy needs for the foreseeable future.
- Choose to support all reasonable policy considerations, allowing the Internet's policies to evolve as its usage and organizational structure change.

The first choice, while certainly expedient, seems risky given how unpredictable the Internet has been so far, both in terms of the nature of traffic and the organizational structure of its stakeholders.[1] In fact, we (as a community) have a terrible record in predicting the future of the Internet, and opting for this choice is a gamble that we will finally get it right this time.

Thus, on policy grounds, the second choice is more desirable. However, it poses two challenges: can we identify what constitutes *reasonable* policy considerations, and can we build a mechanism to support all such policies? In response to the first challenge, we offer the fol-

[1] Recall that the modern ISP-oriented Internet arose in the last fifteen years and is not at all what the Internet pioneers envisioned.

Proposed approach	Policy function								
	dest. control of sender	resource attribution	provider policy granularity			src route control	MB* route control	rcvr-invoked MBs*	network-invoked MBs*
			prefix	suffix	subsequence				
Capabilities, filters [6, 11, 35, 50, 51, 54]	x						* *MB = middlebox*		
Visas [21]		x							
Platypus [43]		x				x			
Pathlets [23]				x	x	x			
LSRR, Wiser [7, 36]			x						
MIRO [49]				x					
Src routing [26, 29, 52, 53]						x			
Byzantine routing [9, 38, 40, 41]						x			
NUTSS [25]							x		
DOA, i3 [46, 48]								x	
DONA [31]									x

Table 1—Policy functions provided by many, but not all, network-layer proposals. Many of these proposals cannot be implemented together. The framework in the text is intended to be flexible enough to capture all of these legitimate policy interests.

lowing principle for reasonable policies:

Policy Principle: *A communication should be allowed if, and only if, all participants approve. By participants, we mean the sender, the receiver, the carriers, and any other intermediaries.*

This principle posits that non-participants should have no say in whether a communication occurs. This doesn't mean that governments and other third-parties have no say about the nature of communications, only that the Internet architecture itself does not enforce such third-party concerns. These third-party concerns must be addressed by other means, such as the legal system.

Note that this principle gives every participant veto power. This may be overkill (for instance, as in [52], one might think that receivers should only be able to control the path of packets once they have left the Internet's core), but we conjecture (based on our inability to find one) that there is no intermediate position or weakening of this policy principle that supports the desires of all stakeholders. Moreover, just because the Internet architecture allows such control does not mean that it will be exercised, as economic and social pressures strongly constrain which policies are enacted. For instance, under BGP, ISPs can pick routes based on the entire AS path, but they rarely exercise more than first-hop preferences. And, the policy principle also gives endpoints choice over paths, so ISPs that impose strict constraints risk losing business to more accepting ISPs.

Thus, while one might fret that the policy principle implies the end of network neutrality and universal connectivity, one could equally expect it to create choice where today there is none. In any case, such debates are not new, as (almost) everything the principle expresses has been previously proposed, in isolation; that is, the principle is (mostly) a union of previous policy proposals.

So we did not create this tussle [17], and we cannot end it here. Instead, we can seek an outlet for it to play out with a mechanism that supports all reasonable policy options. This brings us to the second challenge: can we build a mechanism that supports such a general set of policies? The goal of the rest of this paper is to convince the reader that the answer is not an obvious "no".

ICING We designed a protocol, ICING (Incorporating Consent In the Internet's Next Generation), that appears to satisfy the policy principle as well as further requirements explained in the next section. As a proof-of-concept, we implemented a prototype ICING forwarder in hardware. On the NetFPGA platform, the prototype runs roughly at 4 Gbps (line rate) which is deceptively low because the hardware platform is previous-generation—on it, IP forwarding also runs only at 4 Gbps. Indeed, our estimates indicate that on a custom ASIC, as would be in a modern router, ICING forwarding would run at backbone speeds.

A cost of ICING is packet overhead: 42 bytes per participant. To put this in context, we note that upholding our requirements seems to require *some* per-participant cost, and considerable engineering was needed to get it to 42. Also, we are designing for the future, and technology trends often make today's expensive design tomorrow's commodity;[2] jumbo frames, e.g., would make ICING's overhead negligible. But even without jumbo frames, ICING's total overhead (averaged over packet sizes and path lengths) is 23% over today's bandwidth usage, which may be a fair price for its properties.

Ultimately, ICING is not perfect, but we hope it shows that supporting the general policy principle is, today, not implausible and, tomorrow, not impractical. We return to these claims in §9, after describing the design in §2–§4, the implementation in §5, evaluation results in §6, and policy expressiveness in §7. Some related work is covered in §8, but that section is thin because we mostly acknowledge ICING's (considerable) debts as we proceed.

Before we propose ICING as a supporting mechanism for the policy principle, we first ask: what does it actually mean for a mechanism to support a policy?

[2] For example, research in TCP header compression is now obsolete.

1.2 The nature of mechanism

When we say that a mechanism supports a policy, we mean that it enforces the set of policy choices agreed to by the participants; that is, if the participants all approve then the communication should proceed, and if one or more participants don't approve then the communication should not happen. However, there are further mechanism requirements. We now state several *mechanism principles* that should guide the design of any future Internet (and that guided our design of ICING).

Mechanism Principles:

1. *The mechanism should ensure that approved communications occur as described.* This means that if a communication is described as following a particular path and approved as such, the mechanism should enforce that the communication in fact follows that path.
2. *The mechanism should ensure that unapproved communications cannot be initiated.* This means that if one or more of the participants do not approve the communication, then no packets enter the network. That is, the communication is blocked at the source, before the packets consume network resources.
3. *The mechanism should not rely on any central trusted authority.* No long-lived, global architecture can assume the existence of a permanent, single authority.
4. *The mechanism should impose fixed and feasible requirements on the data plane.* Clearly the mechanism must be feasible, but it should also give router vendors a fixed target to implement, avoiding the explosion of options and features that force continual respinning of router ASICs.
5. *The mechanism should implement subsets of policy efficiently.* This means that if only a subset of the participants wishes to exert their control over communications, then the mechanism should be able to simplify the control plane. In short, the mechanism should not make the Internet pay for unused generality, at least not on the control plane.
6. *The mechanism should work even in the face of malicious participants.* Enforcing policy is not difficult if all participants cooperate. A hard problem is how to enforce policies in a non-cooperative environment.

2 Overview of ICING

This section describes ICING at a high level. §3 and §4 fill in many details.

Architecture ICING divides the network into *realms*. Realms are defined by trust boundaries; no two realms need trust each other. ICING does not change the basic topology and peering model: today's ASes map naturally to realms. However, the granularity of a realm is variable. For example, a host could be its own realm, and for deploying ICING, it may be useful to regard the current Internet as one realm.

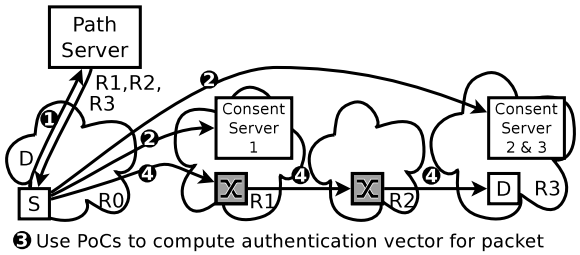

Figure 1—ICING architecture and communication steps. ❶ The sender S communicates with one or more path servers to get a path to D. ❷ S gets a PoC from every realm on the path (above, R_2 has delegated PoC-granting to realm R_3). ❸ S uses the PoCs to construct tokens that ❹ each forwarder verifies and transforms for its successors. Some of these steps are mutually recursive (e.g., the packets to contact a path server travel over ICING so need PoCs); the text explains the bootstrapping needed to end the recursion.

Being concerned with the use and control of realms' resources, ICING allows realms to divide their resources into logical units of control and to name a unit with a compact identifier, called a *vnode* (the term is from Godfrey et al. [23], who introduced a similar abstraction). In the case of a provider realm, these resources are network links and routers (with buffer space) that can prioritize traffic over links, so a vnode refers to a subset of these links and can optionally encode other information, including which customer to bill, router queue priority, etc. As an example, a vnode might include only East-coast links and could be used to charge less for short-haul traffic along this coast, versus long-haul cross-country traffic, which would travel over a different vnode.

Figure 1 depicts the architecture in terms of how a sender communicates with a destination. The sender first exercises control plane functions. It must identify a sequence of (realm, vnode) pairs—a *path*—between it and the destination. (The two endpoints can either be separate realms or vnodes within the first or last realm in the sequence.) Following the policy principle, the sender must, in the general case, get *consent* from each realm. To get a realm's consent, the sender communicates with a general-purpose *consent server* physically separate from the realm's forwarding hardware. The sender proposes the path. In making its decision, the server can incorporate arbitrary factors besides the proposed path (billing relationships [43], authentication, etc.). Upon consent, the server issues a *proof-of-consent* (PoC) that authenticates the path. We cover how the sender gets consent to request consent as we outline the control plane, below.

In the data plane, the sender constructs the packet, which contains its path and cryptographic values that are partially derived from the PoCs. The data plane protocol ensures that forwarders can validate both the path

and whether the packet has taken the path. Following the fourth mechanism principle, the data plane's function is circumscribed (though technically challenging).

The control plane, in contrast, is implemented in general-purpose servers, the outlets by which participants express arbitrary policies. The high-speed forwarders in the data plane are unaware of the control plane, making the control plane modular and pluggable; this decomposition is inspired by [13, 14, 24].

Data plane The forwarders and consent server in a realm share two sets of cryptographic material. The first is a public/private key pair, the *realm key*. A realm's name is its public realm key (as in [5, 39]); such self-certifying names [37] do not require a PKI. Realms use their public keys (but *not* via digital signatures) to provide each other proof that the packet is following its path.

The second is a set of per-vnode symmetric keys, the *PoC keys*. The PoC keys let a realm's consent servers communicate decisions to the realm's forwarders: a PoC includes a MAC of the path, keyed by a PoC key. Because packets contain cryptographic values bound to PoCs, a forwarder can verify that its realm issued consent. Any machine that knows a vnode's PoC key can issue consent to use that vnode; if not located in the realm, such a machine is, de facto, a *delegate* of the realm.

Delegation helps uphold the fifth mechanism principle: to disintermediate itself in the control plane, a realm delegates PoC-issuing authority over a block of vnodes, say to a paying customer. Delegation also aids bootstrapping: a realm can locate its consent server on an isolated vnode and publicly disseminate that vnode's PoC key.

Control plane The control plane's functions include:

- **Configuration.** What information is an end-host, *S*, given when it attaches to a network?
- **Path retrieval.** Given an application-level name (e.g., DNS name) for a destination realm, how does *S* get a path to that realm?
- **PoC retrieval.** Given a path to a destination, how does *S* get permission to send along that path?

Note that these functions themselves require consent because control plane traffic travels in ICING packets.

For *configuration*, when *S* joins a network, it receives *bootstrap state* from a local *configuration server* over a link-layer configuration protocol (analogous to DHCP). This state includes a path to a nearby *path server* (akin to a DNS server). If the path server is not in *S*'s realm, then this state also includes *delegated* PoC keys, which allow *S* to mint PoCs that authorize its own traffic to travel over particular isolated vnodes in the realms between *S* and the path server. Given the preceding state, *S* can now communicate with the path server. The bootstrap state can further include a set of sub-paths between *S* and various upstream realms. For example, if *S* is at a university, *S* might receive sub-paths to: a local provider that peers with the university, the Internet2 network, the university's commercial ISP, and the top-tier ISP from which the commercial ISP buys service.

For *path retrieval*, *S* submits to the path server both the destination's name and, optionally, upstream realms it can reach. The path server, like the configuration server, has network topology knowledge, gained from a routing protocol. It uses this knowledge to return to *S* a path, *P*, that terminates at a host, *H*; *P* travels through one of the upstream realms that *S* can reach,[3] and *H* is either the sender's intended destination or the next path server that *S* must query, in which case this process repeats. In both cases, to communicate with *H*, *S* exercises *PoC retrieval*.

For *PoC retrieval*, the sender must, in the general case (which would hardly ever happen), contact the consent servers in every realm between it and *H*. To contact a realm *R*'s consent server, *S* constructs a path that travels through *bootstrap vnodes* between it and *R*; a realm's bootstrap vnodes are connected only to its own consent servers and its neighbors' bootstrap vnodes, thereby isolating the bootstrap traffic. The PoC keys for these vnodes are well-known, and *S* receives them from the path server during path retrieval. However, a far more efficient option is to conflate PoC retrieval with path retrieval, as follows. A provider realm disintermediates itself by delegating consent-granting to, say, its customers (on particular vnodes, ensuring isolation among customers); its customers do the same with their customers, etc. The result is that (1) *S*'s configuration server can give *S* keys needed to mint PoCs for sub-paths between *S* and upstream realms; and (2) a realm's path server can take on the role of consent server for that realm *and* upstream realms. Now, a path server can give *S* not only a path to a host *H* but also PoCs for a suffix of the path, with *S* minting its own PoCs for the prefix of the path.

3 Detailed design of the data plane

We begin this section with the core technical problem that ICING's data plane protocol must solve. We then describe the protocol and show how it solves the core problem (§3.2). We next describe other aspects of ICING's data plane, including ICING's handling of network failures, delegation, and revocation of consent (§3.3). We discuss attacks and limitations in §3.4.

3.1 Problem statement

Our last mechanism principle requires that ICING enforce policy in a non-cooperative environment. To ensure that ICING is robust in scenarios of varying hostility, we re-

[3] This approach to constructing paths—concatenating sub-paths that intersect in a common intermediate realm—is inspired by NIRA [52]. Like NIRA, it can be generalized to other approaches to path construction because the path server is a pluggable component.

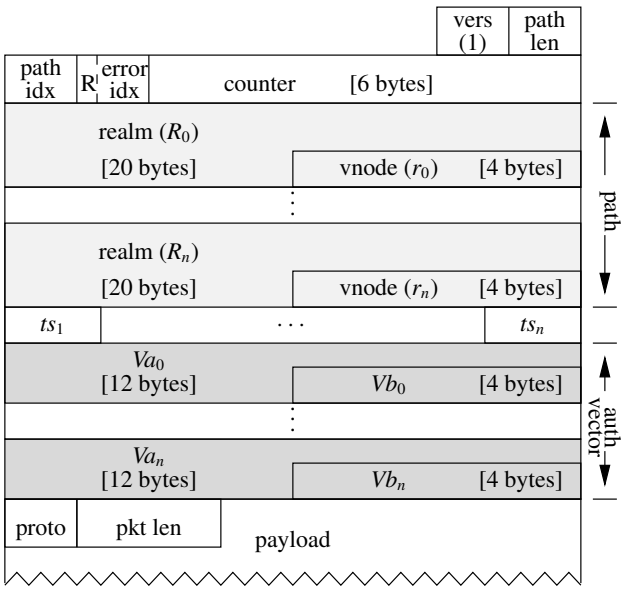

Figure 2—ICING packet format (to follow a 14-byte Ethernet header). The overhead of 42 bytes/realm may seem high, but we explain in §9 why we think that it is not outrageous, even for small packets.

quire it to work under a strongly adversarial model of "non-cooperative", given immediately below.

Threat model We assume that some realms (end-hosts and providers) are controlled by attackers. Such *malicious* realms can deviate arbitrarily from our protocols, including sending arbitrary packets or flooding the links they have direct access to. We make no assumptions on how malicious realms are implemented (they may directly connect to one another and be controlled by a single attacker). Realms that obey the protocol we term *honest*. The protocols that we describe below concern the behavior of honest realms, in particular determining when they have carried or should carry a packet.

Requirements To uphold the policy principle and the first mechanism principle, the data plane must ensure that a packet transits an honest realm R only under the following conditions:

1. [Path Validity] The path P in the packet's header was previously approved by R; and
2. [Provenance Verification] The packet verifiably transited all honest realms before R in P and arrived from the realm just prior to R.

The two conditions do not explicitly constrain a packet's trajectory *after* R. But taken together, they imply:

3. [Path Adherence] A packet forking off its valid path P by skipping an honest realm R_{skip} cannot traverse any honest realm that succeeds R_{skip} in P. (For example, a packet cannot skip a required deep packet inspector and appear valid.)

The third and fourth mechanism principles induce further requirements. The solution must not rely on a PKI,

P	$\langle T_0, T_1, T_2, \ldots, T_{n-1}, T_n \rangle$. A packet's path.
T_i	A pair (R_i, r_i). r_i specifies a vnode (such as a particular class of service) in realm R_i.
R_i	A public key which is also the realm name.
x_i	The private key of realm R_i.
M	{vers, cntr, proto, pkt-len, data}. A packet's end-to-end contents.
s_{T_i}	A symmetric *PoC key* used by R_i's forwarders to verify packets. PoC keys are specific to pairs (R_i, r_i).
$k_{i,j}$	Symmetric key shared by R_i, R_j; derived from their names through non-interactive Diffie-Hellman key exchange.
ts_i	16-bit consent expiration time.
$\mathsf{poc}_{P,i}$	$(\text{PMAC}\,(s_{T_i}, P \parallel ts_i)\,, ts_i)$. *Proof of consent* (PoC) to path P by realm R_i. Valid until ts_i.
$\mathsf{pm}_{P,i}$	Shorthand for $\text{PMAC}\,(s_{T_i}, P \parallel ts_i)$, the cryptographic material in $\mathsf{poc}_{P,i}$.
V^i	$\langle Va_0^i, Vb_0, Va_1^i, Vb_1, \ldots, Va_n^i, Vb_n \rangle$. Auth vector when pkt leaves R_i; lets downstream realms verify provenance.
A_j	PRF-96 $\left(\mathsf{pm}_{P,j}, 0^8 \parallel H\,(P,M)\right)$. Packet-specific authenticator. For notational convenience, let $Va_j^{-1} = A_j$
Va_j^i	PRF-96 $(k_{i,j}, i \parallel H\,(P,M)) \oplus Va_j^{i-1}$. Proves to R_j that packet has transited P through R_i. Unused if $i \geq j$.
Vb_j	Last four bytes of A_j. Guards forwarder slow path from being invoked spuriously.

Figure 3—Cryptographic values in ICING protocol. PRF-96 is a keyed pseudo-random function that maps 256-bit quantities to 96-bit quantities; it functions as a MAC. Our implementation of PRF-96 uses two applications of AES (see Appendix A for details). $H(\cdot)$ is the bottom 248 bits of CHI-256$(\cdot)$, which is a SHA-3 candidate [27].

prior coordination among realms, or per-packet public key cryptography (which would be prohibitive for performance). The solution must be amenable to high-speed implementation, such as in forwarding hardware.

We believe that the combination of the threat model and the requirements is a new technical problem. For example, [9, 22, 38, 40, 41] assume central coordination, don't enforce Path Validity, or aren't amenable to high-speed hardware implementation.

3.2 Response: ICING's core data plane protocol

High-level approach Our high-level approach is as follows. (1) Realms are named by public keys, R_i, that fit in packets. (2) Every pair of realms $\langle R_i, R_j \rangle$ implicitly shares a symmetric key, $k_{i,j}$, that either realm can derive from the other's name and its own private key; deriving these keys requires public-key cryptography but only the first time R_i's forwarder encounters a path containing R_j (and vice versa). (3) Realms use these symmetric keys to provide each other with cryptographic evidence that the packet flowed through them. (4) The sender incorporates into the packet the PoCs that it retrieved (or minted); re-

1: **function** SENDPACKET(P, pocs, m)
// $P = \langle (R_0, r_0), (R_1, r_1) \ldots, (R_n, r_n) \rangle$
// pocs = $\{\mathsf{poc}_{P,i} = (\mathsf{pm}_{P,i}, ts_i) \mid 1 \le i \le n\}$
// $\mathsf{pm}_{P,i} = \text{PMAC}\left(s_{(R_i,r_i)}, P \parallel ts_i\right)$
// $m = \{$proto, pkt-len, [return path + PoCs,]data$\}$
// to guard against replay attacks, init *cntr* per-flow
// $M =$ vers$\|$cntr$\|m$
2: **for** $(i = 1 \ldots n)$ **do**
3: $A_i = \text{PRF-96}\left(\mathsf{pm}_{P,i}, 0^8 \parallel H(P,M)\right)$
4: $Va_i^0 = \text{PRF-96}\left(k_{0,i}, 0^8 \parallel H(P,M)\right) \oplus A_i$
5: $Vb_i =$ last 4 bytes of A_i
6: $V^0 = \langle 0, 0, Va_1^0, Vb_1, \ldots, Va_n^0, Vb_n \rangle$
7: path-idx $= 1$
8: $pkt =$ vers$\|$path-len$\|$path-idx$\|$cntr$\|P\|ts_1 \ldots ts_n\|V^0\|m$
9: transmit *pkt* to R_1 // may need intrarealm forwarding
10: cntr++

Figure 4—Pseudocode for packet construction. $S = R_0$ constructs a packet to send payload m along path P. If the packet is the first in a flow, m may include a return path and PoCs.

call that a PoC proves to a realm's forwarders that the realm consented (perhaps implicitly) to the packet.

Figure 2 depicts the packet format. The *auth vector* is a sequence of digests. The source, R_0, constructs the initial values of these digests by deriving the ith digest from: the PoC for realm R_i, the path, the packet contents, and $k_{0,i}$. Each of the remaining realms R_i *checks* the ith entry in the auth vector and *modifies* entries $i+1, \ldots, n$. Specifically, on receiving a packet, realm R_i must:

- **Verify provenance and consent**: Re-derive the PoC that corresponds to this path and then check that the ith digest (its "own" digest) derives from: the PoC; the path; the packet contents; and successive modification by realms $R_0, \ldots, R_{i-1}$ under $k_{0,i}, \ldots, k_{i-1,i}$, respectively.
- **Prove provenance to later realms**: modify digests $i+1, \ldots, n$ using the shared secrets $k_{i,i+1}, \ldots, k_{i,n}$.

Details Figure 3 depicts the protocol constructs. To make public keys small, we use elliptic curve cryptography. Every realm name, R_i, is a point on NIST's B-163 binary-field elliptic curve group [4]. The corresponding private key, x_i, is the discrete logarithm of the public key: $g^{x_i} = R_i$, where g is a globally agreed upon generator. (The elliptic curve literature uses additive notation, but here, for readability, we use the more familiar multiplicative notation.) To make the protocol more amenable to hardware implementation, we reduce the representation of R_i from 163 to 160 bits by requiring the top three bits to equal a hash of the lower 160; the cost is a factor of 8 in expected key generation time. The security attained is roughly 80-bit security, comparable to that of 1,024-bit RSA keys [4]. The $k_{i,j}$ are generated by non-interactive Diffie-Hellman key exchanges: R_i and R_j share $k_{i,j} = \text{SHA-1}(R_i, R_j, g^{x_i x_j})$.

1: **function** RECEIVEANDFORWARD(pkt)
// $pkt =$ vers$\|$path-len$\|$path-idx$\|$cntr$\|P\|ts_1 \ldots ts_n\|V^{i-1}\|m$
// $M =$ vers$\|$cntr$\|m$
2: check that ts_i is less than current time
// following line may require deriving $s_{(R_i,r_i)}$.
3: $\mathsf{pm}_{P,i} = \text{PMAC}(s_{(R_i,r_i)}, P \parallel ts_i)$
4: $A_i = \text{PRF-96}\left(\mathsf{pm}_{P,i}, 0^8 \parallel H(P,M)\right)$
// extract components in V^{i-1} that we need to verify
5: let $\langle Va_i^{i-1}, Vb_i \rangle$ = the ith entry in V^{i-1}
// following line protects slow path from spurious calls
6: check that Vb_i equals last 4 bytes of A_i; if not, drop
// following line may require slow path invocation
7: compute $k_{0,i}, k_{1,i}, \ldots, k_{i-1,i}$
// simulate what earlier forwarders should have done to
// the ith component of the authorization vector
8: $W = A_i$
9: **for** $0 \le j \le i-1$ **do**
10: $W = \text{PRF-96}\left(k_{j,i}, j \parallel H(P,M)\right) \oplus W$
11: check that $W = Va_i^{i-1}$: if not, drop
// following line may require slow path invocation
12: compute $k_{i,i+1}, \ldots, k_{i,n}$
// construct V^i
13: $V^i = V^{i-1}$
14: **for** $i+1 \le j \le n$ **do**
15: $Va_j^i = \text{PRF-96}\left(k_{i,j}, i \parallel H(P,M)\right) \oplus Va_j^{i-1}$
16: increment *pkt.path-idx* to $i+1$
17: transmit *pkt* to R_{i+1} // may need intrarealm fwding

Figure 5—Pseudocode for interrealm packet forwarding. R_i validates *pkt*, transforms V^{i-1} to V^i, and forwards *pkt* to R_{i+1}.

We label the source of a packet R_0, the destination R_n, and the path $P = \langle T_0, T_1, ..., T_n \rangle$, where $T_i = (R_i, r_i)$, and r_i is a vnode (§2).

We label realm R_i's symmetric PoC key for vnode T_i as s_{T_i}. A realm has 2^{32} such keys but does not manage them individually, as we describe in §3.3. A PoC for path P issued by realm R_i is $\mathsf{poc}_{P,i} = (\text{PMAC}\left(s_{T_i}, P \parallel ts_i\right), ts_i)$, where ts_i is the expiry time, in seconds; PMAC is a MAC that uses a block cipher in a fully parallelizable mode-of-operation, making it amenable to high-speed implementation in forwarding hardware [12]. Realms change their s_{T_i} periodically to guard against chosen-message cryptanalytic attacks and to prevent an old ts_i value that has wrapped from appearing valid (as in [54]). Realms may also change these keys to invalidate problematic PoCs (e.g., those issued to troublesome senders) before those PoCs expire; such rekeying is detailed in §3.3.

Packet sending follows the pseudocode in Figure 4; line numbers below refer to this figure. The source, R_0, is assumed to have one PoC per realm in P (PoC retrieval is described in §2 and §4). Each A_i (line 3) is a packet-specific authenticator that binds together the PoC for realm R_i, the path P, and the payload; given the packet, R_i can re-derive A_i. R_0 creates the initial values in the auth vector, deriving the ith entry from A_i and a MAC of

the packet contents under $k_{0,i}$ (lines 4–6).

Packet forwarding follows the pseudocode in Figure 5; line numbers below refer to this figure. Note that our focus is on *inter*realm forwarding; the actions we describe are not needed for *intra*realm forwarding. When a realm, R_i, receives a packet, it performs three steps. First, it constructs the A_i that these packet contents ought to produce (lines 3–4). Second, it checks that the packet took the correct path: it verifies that the ith entry in the auth vector is equal to the XOR of $i+1$ terms, the terms being A_i and i applications of PRF-96 to the packet contents, one application each under $k_{0,i}, \ldots, k_{i-1,i}$ (lines 5, 7–11). Third, it provides proof for the later realms: for each of the remaining entries in the auth vector, it applies PRF-96 to the packet contents (using key $k_{i,j}$ for the jth entry) and XORs the result into the given entry (lines 12–15).

Note that the first time R_i encounters R_j, its forwarder must use slow path processing to derive $k_{i,j}$ (lines 7, 12).[4] The cost of deriving $k_{i,j}$ is a few msec in our experiments (§6.4). To guard this slow path, the protocol includes Vb_i, which is verified on the fast path (lines 5–6). Without this check, an attacker could invent realms and bogus paths to force spurious slow path operations on forwarders. Vb_i is only 32 bits, so it does not rule out such attacks altogether, but it decreases their effectiveness by a factor of 2^{32}, which is sufficient to avoid denial-of-service.

Meeting the requirements from §3.1 The condition of Path Validity is upheld because realm R_i, given a packet with path P, re-derives $\mathsf{poc}_{P,i}$ (line 3, Figure 5) and then checks that Va_i^{i-1} suitably derives from $\mathsf{poc}_{P,i}$ (line 11). Similarly, Provenance Verification is upheld because a correct value of Va_i^{i-1} indicates that the precise packet contents flowed along every element in P prior to R_i in the correct order; the order is upheld by the j in the argument "$j \parallel H(P,M)$" in line 10. Also, the protocol is amenable to high-speed implementation (see §5) and does not incorporate a PKI, require coordination among realms, or use per-packet public key cryptography.

3.3 Other requirements

ICING must accommodate bi-directional communication, handle network failures, permit efficient management of PoC keys, enable delegation, and permit revocation of consent. We now describe these functions.

Return paths While a destination, R_n, can reply to a source, R_0, by resolving R_0 at path servers and obtaining its own PoCs, it is more efficient for R_0 to negotiate the return path at the same time that it negotiates the forward one. Thus, the payload of a packet optionally begins with a return path and *return PoCs* that the recipient can use to reply. This approach offloads return path negotiation to clients, which helps in settings where it is important to minimize load on servers. Note that return paths are not included in every packet; they are needed only for the first packet in a flow and after the path changes in the middle of a communication, a case that we discuss now.

Network failures What happens if a network failure (from change in topology, mobility, link failure, etc.) invalidates the path between R_0 and R_n? If R_0 knows of the failure, it either re-runs path and PoC retrieval, or else uses a backup path and PoCs that it may have obtained during normal path retrieval. Note that using a backup path would be much faster than BGP convergence [33], if R_0 is told of the error as soon as it happens.

To inform R_0 of the failure, the realm that experiences the error sends a signal along the reverse of the path. A subtlety is that this signal must not violate Path Validity or Provenance Verification. Our approach is as follows. We assume that when a realm consents to a path P (by issuing a PoC), it also consents implicitly to carry error packets backwards along the reverse of any prefix of P.

When R_i experiences an error, it sets the packet's error-index field to i; sets the R (reverse) bit; applies PRF-96 to the packet contents under $k_{i,j}$ and XORs the result into the jth component of the auth vector, $i-1 \geq j \geq 0$; and then forwards the packet "backwards" to R_{i-1}. The packet will now flow along the reverse path to the source.

When a realm, R_j, $j < i$, receives a packet with the R bit set and the error index set to i, it performs three steps, in analogy with the forward direction. First, it constructs A_j. Second, it checks that the jth component of the auth vector equals the XOR of $i+1$ terms: A_j and i applications of PRF-96 to the packet contents, one application each under $k_{0,j}, k_{1,j}, \ldots, k_{j-1,j}, k_{i,j}, k_{i-1,j}, \ldots, k_{j+1,j}$. This check ensures that the packet flowed from the source through realm R_j, reached realm R_i, experienced an error there, and flowed from R_i back to R_j. Third, it applies PRF-96 to the packet contents using $k_{j,l}$ and XORs the result into the lth component of the auth vector, $j-1 \geq l \geq 0$. Under our assumption above, this approach upholds Path Validity and Provenance Verification.

If our assumption—that consenting to the forward direction implies consent to carry error packets in reverse—does not hold, the above approach would violate Path Validity. Consider, for example, a strict network that wants to receive data from a neighbor but not to emit data to that neighbor, say because doing so would leak information to this neighbor. To handle such cases, a convention is that one of a realm's vnode bits means, "not willing to send in reverse". A strict network R_i expresses its strictness by consenting only to paths where

[4] $k_{i,j}$ and $k_{j,i}$ have different purposes (one is for R_i to make statements to R_j; the other, vice-versa). However, our implementation sets $k_{i,j} = k_{j,i}$, but there is no loss of security because PRF-96 takes an extra bit as input (not notated) such that PRF-96$(k_{i,j}, m) \neq$ PRF-96$(k_{j,i}, m)$. See Appendix A for details.

r_i has this bit set. Then, if a path contains a vnode with this bit set, and if that vnode is in the part of the path that would have to carry an error packet backward, the realms do not send error packets in reverse. R_0 knows at the outset that it is sending along such a path, so it can make other arrangements, such as timing out and/or sending on backup paths speculatively.

PoC key derivation As mentioned above, each realm has 2^{32} vnodes, each of which requires a PoC key shared across all consent servers and forwarders in the realm. Distributing 2^{32} PoC keys would be expensive. Instead, we rely on a small number of shared *prefix keys* to generate many PoC keys. Specifically, let r/p denote the p-bit prefix of vnode r. If $r/(p-1)$ has prefix key $m_{r/(p-1)}$ and r/p has no explicitly shared prefix key, then the prefix key for r/p is computed as $m_{r/p} = \text{MAC}(m_{r/(p-1)}, r/p)$ (a technique suggested by [43]). The per-vnode key $s_{(R,r)}$ is just $m_{r/32}$.

In the simplest case, all vnode keys can be derived from a single key, $m_{0/0}$. This may be done initially, but for revocation purposes (discussed below), realms will need to change individual PoC and prefix keys. For performance, implementations can cache prefix keys to exploit vnode locality. (Our implementation currently just caches PoC keys, pre-filling the needed portion of the cache for simplicity.) Realms can also speed PoC key derivation by MACing more than one bit at a time, i.e., $m_{r/p} = \text{MAC}(m_{r/(p-b)}, r/p)$, for $b > 1$.

Delegation It is highly convenient for realms to be able to *delegate* PoC-issuing ability to other realms. Consider a backbone provider that delegates control to a customer (who might in turn delegate to its customer, and so on). The backbone provider gains by avoiding the burden of granting PoCs for every flow that crosses its network. Customers gain because they can deny upstream PoCs according to their own policies, stopping unwanted traffic before it arrives on their or their providers' networks. Of course, the delegator does not usually wish to delegate control over its entire network; rather, it needs a way to delegate control over a portion of itself.

Given the above approach to PoC key derivation, such *controlled delegation* is easy to implement. A realm can delegate control over a block of vnodes with a common prefix, r/p: the realm simply divulges $m_{r/p}$. The delegate can sub-delegate a portion of this block by divulging $m_{r/q}$, where $q > p$. The end result is that an entity close to the edge, such as an end-host, can issue PoCs on behalf of a block of vnodes inside the backbone provider. When the provider's forwarder verifies packets using the appropriate per-vnode PoC key, the forwarder does not know that the packet's PoC was issued by a delegate.

Note that the delegator may wish to retain some control over the delegated vnode. For example, the delegator may want the vnode to be used only for paths that travel between two particular neighbors. Or, a realm may delegate a vnode to a sender but want the sender to use the vnode only for traffic sent to a particular receiver. While the delegator can enforce some of these policies via *intrarealm* forwarding and filtering, others require the delegator to audit traffic over the delegated vnode.

Revocation As so far described, realms (or their delegates) are "stuck" with their PoC-issuing decisions. If a realm (or its delegate) regrets having issued a PoC to a troublesome sender, the realm's forwarding hardware will continue to carry traffic along the given path until the PoC expires. Yang et al. [54] address this problem in the context of network capabilities by associating a maximum number of bytes with a capability (they argue, and we concur, that quickly blacklisting issued permissions would be too difficult). While our mechanism could be extended to take a similar approach, it only mitigates the problem. Instead, a realm can immediately invalidate PoCs by rekeying the needed vnodes (and then informing its delegates of the change). Re-keying may cause collateral damage; specifically, all current users of a vnode must now obtain new PoCs from the rekeying realm, which may noticeably pause valid communications. Premature rekeying is therefore primarily appropriate in emergencies.

3.4 Attacks and limitations

Given our threat model, ICING must be robust to attack. Below we consider various attacks and their defenses. We then discuss functions that ICING cannot provide.

Replay attacks An attacker who has observed a valid packet may inject a duplicate copy along a suffix of a path. At low rates, such attacks are not problematic: the layers above datagram protocols must handle duplicates anyway. We now consider aggressive replay attacks. If the replayed packets come from a small number of flows, a modest-sized cache of paths and counters from recently seen packets can limit replays to a small fraction of traffic. More challenging is dealing with attackers who can, within a single PoC validity window, amass packets from enough different paths that they overflow the replay cache. Defending against this case is future work; it may require both reducing the PoC validity window and employing Bloom filters or similar techniques to maximize the number of entries in a small replay cache.

Packet floods An attacker can try to flood a network link or destination. If the attacker does not have valid PoCs, this attack will be limited (§4 limits it further). Here, we cover the case that a flooding attacker has valid PoCs. This attack captures classic denial-of-service (directly attacking a server) and "denial-of-capability" [8, 54] (attacking a consent server with spurious PoC requests).

These attacks are the same in our context—PoC requests travel "in-band"—so we do not differentiate them below.

If a realm housing the victimized server can identify clients, say at the granularity of categories (e.g., "employees who need to reach the internal network", "paying customers", "unknown clients who solved a CAPTCHA", "unknown clients, some of which are attackers"), then it can assign each category to a different vnode. Now, when the server is attacked and overloaded, the victim deprioritizes the "unknown" categories, either by not reissuing PoCs on those vnodes after they expire; giving downgraded service to those vnodes; or, in an emergency, rekeying those vnodes (§3.3). A related defense is that an organization can issue PoC keys for a distinguished vnode to employees. If providers fair queue by vnode, then employees are guaranteed to be able to reach their employer, even in the face of massive packet floods.

If clients cannot be assigned to categories, we (blatantly) borrow a mechanism from TVA [54]: a victimized realm or its providers can hierarchically fair queue based on the packet's path, to ensure roughly fair bandwidth consumption among senders. Note that while an attacker can weaken this defense under TVA by faking path identifiers, ICING's properties prevent this weakening.

Cheating providers A cheating realm R cannot violate Path Validity or Provenance Verification (§3.1). For example, R cannot short-circuit a sub-sequence of the agreed path or inject new packets along a portion of the path. However, R can drop packets that it consented to carry. More generally, it can neglect to give a packet the service level specified by the vnode (R, r) listed in the packet (to which R consented). So far, this attack is a limitation of ICING; defending against it is future work.

Compromised secrets ICING's guarantees assume that honest realms' keys are not compromised. But what if such a key is compromised, as will inevitably happen? Because ICING's data plane is off-by-default, the most likely damage is an attacker spuriously consuming resources (versus blacklisting honest participants, the main danger in AIP [5]), or subverting control plane protocols.

To recover from key compromise, a realm must change its keys; we discuss the consequences for each key type:

Private keys (x_i). If a realm's private key is compromised, the realm must change its name (i.e., its public key). The change is propagated through routing protocols (§4) no differently from a topology change. A realm must also update name-to-realm mappings stored in, say, path servers. While doing so is trivial for single-host realms, if a realm comprises many hosts (each on a separate vnode), the large number of (potentially distributed) mappings creates a challenge. On the other hand, A6 records [18] solve a precisely analogous problem in the IPv6 context, and ICING can borrow this solution, though we leave the details to future work.

Shared symmetric keys (k_{ij}). One or both of the compromised realms must again change names.

Per-vnode PoC keys ($s_{(R,r)}$, also notated as s_{T_i} and $m_{r/32}$). The enclosing realm, R, must rekey (§3.3) the PoC or an enclosing prefix of the PoC.

Per-vnode-prefix secret ($m_{r/p}$). The enclosing realm must again rekey, as above.

Compromised secrets are a serious issue, but ICING is not so different in this aspect from many other systems. Ultimately, ICING is designed to give guarantees to realms whose keys are *not* compromised, while permitting realms that discover compromises to change keys.

Limitations and non-goals We now describe what ICING does not, and is not designed to, achieve.

Although realms can enforce policy based on where a packet went and on where it *claims* it will go, they cannot control where a packet *actually* will go. If, for instance, realm B consents to path $\langle A, B, C, D\rangle$ but not to $\langle A, B, C, D'\rangle$, then A, C, and D' can still collude to use the latter path (which could potentially result in B charging D for traffic that actually went to D').

ICING also cannot meaningfully enforce negative policies against edge realms. Consider blacklisting for example. While the Internet's IP address scarcity makes blacklisting bad IP addresses effective, under ICING, in contrast, edge realms can generate new names for themselves. On the other hand, a bad edge realm's provider, and the vnodes assigned to that edge realm, may be harder to change, which would make blacklisting the provider, or one of its vnodes, effective.

ICING does not offer confidentiality.

Finally, ICING's concern is network-level policy, rather than information flow control at higher layers. For example, an application on host A might send data to host B via an application on host C even if A's provider does not approve the network-level path $\langle A, C, B\rangle$.

4 Control plane

This section fills in some details of the control plane. It is highly abridged; Appendix B gives a more complete description.

Consent certificates A key data structure is the *consent certificate*, a signed statement by a realm's private key[5] that expresses both topology and policy information about the realm. In the simplest case, a consent certificate names three realms: the issuing realm, R, and two neighbors, A and B, with the semantics that R is declaring that it connects to A and B and is willing to carry traffic

[5]Recall that realms also use their private keys to generate the $k_{i,j}$ (§3.2). Such "dual purposing" of key material is wisely discouraged by folklore. However, a careful analysis, which is outside of our scope, indicates that our protocols are safe.

from A to B, perhaps on particular vnodes. Consent certificates can express richer semantics too. The exact format, together with proposed extensions, is given in Appendix B. We say that a set of consent certificates is *full* with respect to a path P if, for each realm on P, the set contains a certificate whose expressed policy allows P. A full set proves that all realms on P consent (at least in principle) to P.

The second and fifth mechanism principles So far, ICING upholds the second principle only under honest senders. A dishonest sender can construct a packet from a mix of valid and bogus PoCs, thereby consuming network resources en route to the first non-consenting realm. Put simpler, these packets are dropped later than they ought. However, if consent servers (which, recall, take arbitrary input) require a full set of consent certificates before issuing a PoC, then a sender cannot violate the second mechanism principle: if *one* of the realms on the path doesn't consent—as expressed by its never having issued an appropriate certificate—then *none* of the honest realms issues a PoC. Thus, spurious traffic will be dropped early (specifically, at the first honest realm).

But how do consent servers get consent certificates in the first place? The answer is via routing and path construction protocols. Below we describe several approaches to these functions. Because ICING's control plane is "pluggable", these approaches can coexist.

Before describing them, we note that they uphold the fifth mechanism principle: realms can disintermediate themselves from PoC-issuing (e.g., under the first approach, only the sender and the destination's path server issue PoCs). Disintermediation happens via delegation (realms can offload consent-granting to, e.g., customers) and consent certificates (the *absence* of which help a sender *not* make a PoC request that would be denied).

sIRP As briefly mentioned in §2, path and configuration servers gain network topology knowledge by participating (perhaps through proxies) in a routing protocol. Our implementation uses sIRP (Simple ICING Routing Protocol). A link state protocol, it propagates sets of consent certificates, which function as link state advertisements. A provider R propagates to each of its customers (1) the messages that R itself receives; and (2) consent certificates expressing that the customer can transit R to R's neighbors (R's other customers, R's providers, and R's peers). The end-result is that each edge customer gets a set of consent certificates that validate paths to well-connected providers (e.g., the Internet core) and to intermediate realms (such as its provider's peers, and its providers' providers). Under sIRP, all valid paths—i.e., those for which one can assemble a full set of consent certificates—are valley-free. sIRP's scalability derives from the fact that messages flow only "downward".

We now describe path construction under sIRP. It is inspired by [52]. When a sender requests a path, the path server finds an intermediate realm, I (e.g., a tier-1 ISP, Internet2, etc.) for which (1) there is a valid sub-path from the sender to I; (2) there is a valid sub-path from I to the destination; and (3) the end-to-end path through I is valid. The path server examines consent certificates that the sender supplies (which the sender received from the routing protocol, via the configuration server) and certificates relevant to the destination (which the path server received from the routing protocol, perhaps via the destination), and identifies an end-to-end path for which a full set of certificates exists. Such a set (almost) guarantees PoC retrieval will succeed, avoiding costly path retrieval retries. We say "(almost)" because a realm can always deny to issue a PoC. In any case, the path server returns the full set to the sender, which submits them to consent servers (answering the question above about how consent servers get consent certificates).

Other approaches An extreme approach to routing and path construction is to run BGP on realm names (using signed BGP messages as consent certificates) and to publicly disseminate PoC keys allowing any participant in the routing protocol to mint a valid path suffix. For details, see §B.5. A key point is that [5] proposes nearly exactly this, and shows how it can scale, an analysis that mostly applies here (this approach requires a linear factor more space to store paths, but on general-purpose servers). Another approach to routing and path selection is to embed pathlets [23] in consent certificates (§B.5).

5 Implementation

This section describes our implementation of the hardware and software data plane, the control plane, and endpoint software. All of our software runs on Linux 2.6.25.

Data plane Our prototype forwarder accepts ICING packets carried in Ethernet frames and implements the protocol in Figure 5. The fast path runs on the NetFPGA programmable hardware platform [3], which has 4 Gigabit Ethernet ports. When an ICING packet enters the fast path, if the packet's path contains one or more realms R_j for which the forwarder, representing realm R_i, does not have $k_{i,j}$ cached in hardware, the hardware sends the packet to a software slow path over the PCI bus to an x86 processor. The slow path, implemented in Click [30], calculates the needed keys and installs them in the hardware's key cache, possibly evicting old keys. The Diffie-Hellman key exchange is implemented with the MIRACL cryptographic library [45].

We have not yet implemented PoC expiry via the *ts* field (§3.2), handling packets with the reverse bit set (§3.3), or replay prevention (§3.4).

Figure 6 shows the major hardware blocks. The hard-

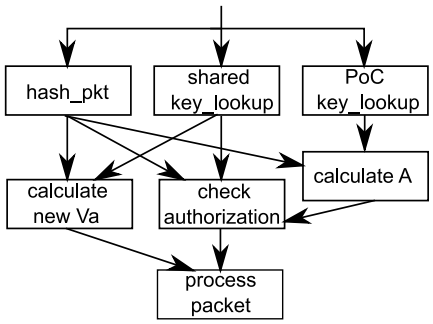

Figure 6—Block diagram for ICING forwarder hardware fast path showing the ICING-specific logic, not including PMAC and AES. In parallel, the blocks in the top layer calculate $H(\cdot)$, look up $k_{i,j}$ from an SRAM cache, and look up s_{T_i} from an SRAM table. The results are passed as needed to modules that, in parallel, calculate $Va^i_{i+1}, \ldots, Va^i_n$, check Va^{i-1}_i, and calculate A_i.

ware image is based on, and uses support modules from, the reference base design from the NetFPGA project. We implemented the ICING-specific logic, including cryptographic modules.

The total equivalent gate count for our NetFPGA forwarder design is 13.4M gates; it uses 89% of the total FPGA logic area. This area is broken down as follows: 38% to the AES, CHI, and PMAC modules, 28% to all other ICING-specific logic, and 34% to the support modules. In comparison, the NetFPGA reference IP router has an equivalent gate count of 8.7M; it uses 50% of the total FPGA logic area. The hardware image is 3100 ICING-specific semicolons of Verilog (1000 for crypto modules, 2100 for other ICING logic); the software is 1260 semicolons of C++, not including cryptographic libraries.

Control plane and endpoints. Our combined consent and path server is embedded in a DNS-like naming hierarchy and exposes a `getpath()` call over XDR RPC (which returns a path to a destination or to another such server). These servers participate in sIRP. The control plane modules are 1500 semicolons of C++, not including cryptographic libraries. An endpoint application sends ICING packets by invoking a path server via a local library function. This function installs the needed path and PoCs in a table maintained by Click, and returns to the application an IP address that routes to `tun`, which is an interface to ICING's packet processing code in Click.

6 Evaluation

ICING introduces space and time overhead from per-packet cryptographic objects and operations. Our principal question in this section is whether these overheads are practical on Internet backbone links. We begin by estimating ICING's total space overhead (§6.2). In §6.3 and §6.4 we present microbenchmarks that evaluate the performance of our prototype forwarder and the supporting software, respectively. Finally, in §6.5, we extrapolate from our results to assess ICING's future feasibility in the Internet core. Table 2 summarizes our main results.

The average ICING overhead is 22.4% above current bandwidth consumption.	§6.2
Our prototype forwarder processes packets at between 3 and 4 Gbit/s.	§6.3
We project that an ICING forwarder could scale to backbone speeds at tolerable hardware cost (IP itself runs only at 4 Gbit/s on our hardware platform).	§6.5
Microbenchmarks suggest that ICING's software costs are tolerable.	§6.4

Table 2—Summary of main evaluation results.

Machine type	CPU	RAM	OS
slow	Intel Core 2 Duo 1.86 GHz	2 GB	Linux 2.6.25
medium1	Intel Core 2 Quad 2.40 GHz	4 GB	Linux 2.6.25
medium2	Intel Core 2 Duo 2.33 GHz	2 GB	Linux 2.6.27
fast	Intel quad Xeon 3.0 GHz	2 GB	Linux 2.6.18

Table 3—Machines for measuring ICING overhead.

6.1 Setup and parameters

Table 3 lists the four machines classes that we use to evaluate ICING. The *fast* machines are installed in three Internet2 Point-of-Presence (PoP) locations: Houston, L.A., and New York. All machines except *medium2* have NetFPGA cards. The NetFPGAs in the Internet2 nodes connect in a full mesh by dedicated 100Mbit/s circuits.

Our experiments often vary packets' path lengths, path indices or sizes. Table 4 gives the fixed and variable parameters used for the forwarding latency and throughput, and software performance measurements.

6.2 Packet overhead

The ICING header size is significant. The header fields that do not depend on the packet's path length use 13 bytes (see Figure 2). Each (R_i, r_i) is 24 bytes, each component of V uses 16 bytes, and the PoC expiration ts_i takes 2 bytes. Thus, if x is the size of a packet and y is the packet's path length, the total header overhead, as a fraction of the packet size, is:

$$J(x,y) = \frac{13 + 42 \cdot y}{x}$$

For a packet whose path is 7 realms long—the average length of an AS level path found in [28] but a conservatively high estimate, according to [5]—the header is 307 bytes, or 20.3% of a 1514-byte packet. For small packets, ICING's header overhead would be far larger; we note, however, that most bytes are carried across the Internet in large packets, so the total contribution to ICING's overhead from small packets may be small. To get a sense of

Varied parameter	Range	Fixed parameters		
		Path len	Path idx	Pkt size
Path length	{3, 7, 10, 20, 30, 37}	—	1	1514
Path index	{1, 5, 10, 15, 18}	20	—	831
Packet size	{311, 567, 823, 1335}	7	3	—

Table 4—Parameters used throughout experiments. Packet size includes header.

how much overhead ICING's headers would impose on Internet-like traffic, we look both at the expected overhead of a randomly selected *packet*, and at the expected overhead of a randomly selected *byte*. The latter statistic captures the total cost of ICING, namely how many total bytes it would add to data plane traffic, assuming today's packet size distributions (this statistic is the one that accounts for the fact that most bytes travel in large packets).

Per-packet overhead The expected overhead of a random packet is given by $\mathbf{E}(J(X,Y))$, where X, Y are random variables for the packet size and the path length, respectively. Assume that the size of a packet and the length of its path are independent, and bound the average path length by 7 (a conservative estimate, according to [5]). The sought average can then be computed as $\mathbf{E}_X(J(X,7))$, *i.e.*, the average of ICING's header overhead for various packet sizes, weighted by their proportion over typical Internet traffic. Using packet size data from a 1-hour long trace of a mid-West to West Coast OC192 backbone link of a US Tier-1 ISP [47], ICING's header overhead for a random packet is 45.4%.

Total overhead from ICING The preceding statistic captured ICING's overhead for a randomly selected packet. But we expect that statistic to overstate ICING's true cost because most bytes travel in large packets (intuitively, the outsized cost of an ICING header on a small packet is not incurred often, relative to the total amount of data traveling). Thus, we now examine ICING's effect on overall bandwidth consumption.

We're interested in (*total*-ICING-*header-bytes* / *total-data-bytes-sent*), which equals:

$$\frac{\textit{total}\text{-ICING-}\textit{hdr-bytes}}{\textit{total-num-pkts-sent}} \Big/ \frac{\textit{total-data-bytes-sent}}{\textit{total-num-pkts-sent}}$$
$$= \textit{avg}\text{-ICING-}\textit{header-bytes-per-pkt}/\textit{avg-pkt-size}$$

or $J(\mathbf{E}(X),\mathbf{E}(Y))$, the reason being that the numerator above is simply the number of bytes added by ICING for a randomly selected packet, which depends only on average path length. If we (conservatively) take the average path length, $\mathbf{E}(Y)$, as equal to 7, and if we take X as distributed according to the dataset in [47], then $J(\mathbf{E}(X),\mathbf{E}(Y))$ equals $307/1370$, or 22.4% above current bandwidth consumption.

We discuss the import of this statistic in §9.

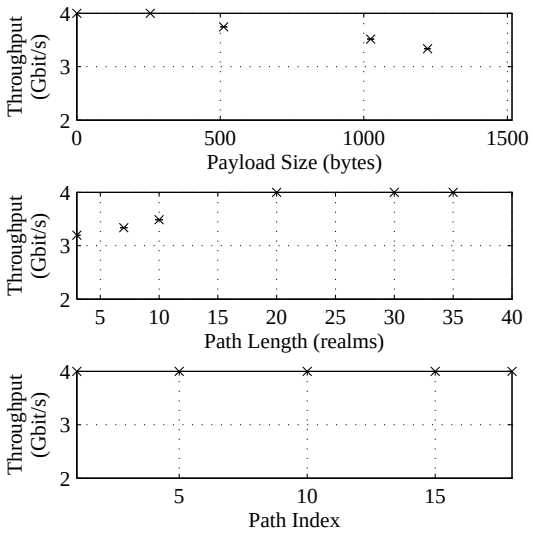

Figure 7—Average throughput as the payload size and path length. Standard deviation was less than 0.2 per thousand of the mean at each measurement point. The minimum throughput was 3 Gbit/s and payload size had the most impact on average throughput when the packet size is constant.

6.3 ICING forwarder

From Figure 5, one might expect the cost of processing a packet to depend on the number of applications of PRF-96 that must be performed. However, because the PRF-96 results are XORed, they can be parallelized. On the other hand, as the ratio of payload size to path length in a packet increases, the number of auth vector entries (that are not hashed) decreases and the hash function becomes the bottleneck. To validate, we measure our prototype's fast path throughput by connecting the four ports of an ICING forwarder to a NetFPGA packet generator that sends ICING packets at line rate. We measure throughput over 5 10-second samples, using the measurement points in Table 4. The ICING forwarder loops ingress packets back to the packet generator, which measures the average bit rate.

Figure 7 plots the measured throughput. Note that we do not report goodput; instead we acknowledge that ICING has a 22.4% overhead, as analyzed in §6.2. The minimum aggregate throughput was 3 Gbit/s. The path index has no effect on performance because it doesn't affect the number of PRF-96 applications or the number of bits hashed.

6.4 Software performance

We now measure the performance of ICING software, including the forwarder's slow path. We focus on calculating shared keys, generating PoCs, and packet handling. Table 5 summarizes.

Shared key ($k_{i,j}$) calculation. We measure the cost of the ICING forwarder's slow path by running 3000 itera-

Action	Processing time	Throughput (1/Proc. time)
Calculate $k_{i,j}$	4 ms ($\sigma = .043$ ms)	250 keys/s
Generate PoC	$0.4l + 1.3$ μs	$2.6 \cdot 10^6/(l + 3.5)$ PoC/s
Create packet	$2.6l + 40.1$ μs	$3.9 \cdot 10^5/(l + 15.4)$ pkt/s
Verify packet	$2.6l + 24.4$ μs	$3.9 \cdot 10^5/(l + 9.5)$ pkt/s

Table 5—Processing time and throughput for software operations, where l is the path length. Each entry in the path increases packet creation and verification times by 2.6 μs, the cost of two AES encryptions. For the last three rows, processing time is derived by linear regression, and $R^2 > 0.99$ in all three cases.

tions of the calculation function in a tight loop on a *slow* machine. On average, a single key calculation takes 4 ms.

PoC creation. To represent the hardware that runs ICING control plane servers, we use *fast* Internet2 machines to benchmark the consent server. To measure the cost of generating a PoC, we run the calculation function in a tight loop, varying the path length per Table 4. Our results show that the cost is proportional to the path length, as expected from the definition of $\mathsf{poc}_{P,i}$.

End-host. An end-host must also perform cryptographic operations: senders initialize the auth vector and receivers validate V^n. To understand these costs, we seek a linear function from path length to processing time. To infer such a function, we vary path length per Table 4, take packet size to be 1514 bytes, and collect 1000 samples per path length. We record total processing cost (of either packet generation or verification, depending on sender or receiver; in both cases, they retrieve the needed $k_{i,j}$ from a cache so as not to count the shared key cost), and then use ordinary least squares linear regression. The inferred coefficients ($R^2 > 0.99$) are in Table 5.

Note that an end-host takes longer to generate a packet than to verify one. This is because senders are so far unoptimized and compute $H(P,M)$ twice The receiver hashes the packet only once (to verify V^n) but incurs a cost that sender does not, namely computing its PoC.

6.5 Scaling

Here, we do some back-of-the-envelope estimates to get a rough sense for whether an ICING forwarder could scale to backbone speeds.

Before delving into more detail, we just note that while our ICING forwarder might seem slow in absolute terms, it runs at almost the same speed as the reference IP router [3] built on this platform (i.e., FPGAs are slow). Since production IP can run at backbone speeds on an ASIC, we believe that there is no fundamental obstacle to running ICING at these speeds on an ASIC. Of course, forwarding an ICING packet requires more per-packet processing than forwarding an IP packet, which is reflected in ICING's requiring 78% more FPGA logic than the reference IP router (as mentioned in §5). Beyond the fact that this difference might be an acceptable price for ICING's properties, we note that it is precisely such differences—factors of two in area cost—that are the subject of Moore's Law.

We now go into more detail, answering two questions:

- Can we build a forwarder that can handle future Internet backbone traffic at a reasonable hardware cost?
- Is the amount of state that an ICING forwarder needs to store reasonable?

6.5.1 Throughput and cost

Current backbone links are 40 Gbit/s (OC-768). For now, we set a target of over 100 Gbit/s.

Because we have been unable to find die sizes for ASICs in commercial networking products from vendors such as Cisco, Juniper, and Broadcom, our comparison is relative to the FPGA chip (Virtex-II pro 50) that we are using. Measurements from [32] suggest that the ratio of chip area consumed by an FPGA to that consumed by an ASIC for the same design varies between 12 and 70, depending on the types of hard macro blocks used and the type of logic implemented by the FPGA design. Our design uses only Block RAM hard macros. Thus, according to Table II in [32], the average ratio is 33. Moreover, the Virtex-II pro uses 0.13 μm technology while today's ASICs use 40 nm technology, so area reduces by an additional $(130/40)^2$, or a little over a factor of 10, giving a factor of roughly 330 altogether.

Moving to an ASIC also allows higher clock speeds from reduced combinational and routing delay. The average delay reduction found in [32] is 3.5 times. And, moving to a smaller technology can further increase clock rates, but we are conservatively disregarding this effect.

Applying the above estimates literally would mean that an ASIC implementing our ICING forwarder design would be at least 330 times smaller than the Virtex-II pro 50 and would run 3.5 times faster—roughly 10 Gbit/s (i.e., 3.5 times faster than the minimum speed of our implementation, which is 3 Gbit/s, from Figure 7). We can now "spend" some of that factor of 330 to replicate processing logic by a factor of 10 to reach our goal of 100 Gbit/s. The end result is still a trivial amount of area at the 40 nm technology.

Looking ahead, we expect ICING to keep pace with increases in backbone link speeds. The reason is that backbone link speeds have roughly followed Moore's Law, and ICING's main bottleneck is one hash calculation per packet, the speed of which will also track Moore's Law, assuming current trends continue.

Our conclusion is that an ICING forwarder could, in *throughput* terms, achieve current backbone speeds without being too expensive. However, we must also consider *goodput*: ICING's 22.4% average overhead means that an ICING forwarder must run at 122.4 Gbps to meet a target goodput of 100 Gbps. This extra required speed would have one of two effects, depending on the bottleneck. If

packet processing logic is the bottleneck, then the 22.4% overhead would add proportional cost to our estimates. On the other hand, if interconnect or I/O (or pin) bandwidth is the bottleneck, then for ICING to achieve the same goodput as IP might require an ICING forwarder to have more pins, which might translate into needing more chips, in which case the cost of an ICING forwarder would rise non-proportionately.

6.5.2 State

An ICING forwarder stores three kinds of state: a table of its directly connected neighbors; a *symmetric key cache* to store the $k_{i,j}$; and the *PoC key cache* to store precalculated PoC keys or prefix keys. The first kind is negligible. In the remainder of this section, we focus on the latter two, asking whether the required state can fit in a commodity SRAM or CAM.

Symmetric key cache To ensure high speed forwarding in all cases, a forwarder's symmetric key cache must hold the $k_{i,j}$ for all realms in all packet paths that pass through the forwarder. Assuming that the number of realms will be, roughly, the number of autonomous systems, we can set an upper bound on the maximum size of the key cache by looking at the number of advertised Autonomous System Numbers (ASNs). As of October 6, 2009 this number is less than 33k and growing at less than 3.2k/year [1], so the key cache size for an ICING forwarder that can handle today's traffic and the traffic for at least the next 5 years—assuming the growth rate remains constant—is less than 100k entries. Note that a forwarder will almost certainly never be receiving flows passing through *every* realm on the Internet, so the actual required number is far less.

Our ICING forwarder already has a hash table that can fit 32k entries, and current IP routers and switches already have tables on the order of hundreds of thousands of entries [2]. Thus, our rough estimate is that the key cache can be implemented easily in SRAM. For further analysis of a nearly identical question, see [5, §4].

PoC key cache As described in §3.3, a forwarder derives the PoC key for a vnode by successively MACing a prefix key. While our implementation uses only 16 bits of the vnode, allowing us to fit all of the PoC keys in SRAM, we must ask whether arbitrary implementations have the needed storage and processing power to cache or derive keys. An extended analysis is outside of our scope, so we just note the following. 32 MB of SRAM that runs at the speeds that we are targeting is not an unreasonably large and fast SRAM. PoC prefix keys are 32 bytes, so with 32 MB of SRAM, an ICING forwarder can cache 2^{20} prefix keys. Thus, the forwarder must either cache commonly used PoC keys or perform enough MACs (which are AES encryptions in our implementation) to go from a 20-bit prefix key to a 32-bit prefix key (which is the PoC key itself). We do not believe that either is a fundamental obstacle. For example, if PoC key derivation occurred at 12 bit boundaries (see §3.3), and if the forwarder cached all 2^{20} prefix keys, then only one AES invocation is needed per packet processed.

Of course, as SRAMs become denser and AES invocations faster, the PoC key cache will become even less burdensome.

7 Expressiveness

How much policy expressiveness does ICING give? Informally, ICING's mechanisms can express the high-level policies of many prior works, including those in Table 1 (making this claim precise would require a formalism that is outside of our scope). A second indication of ICING's expressiveness is that it enables new functions. We now give several examples.

Sink routing. The literature on source routing is vast, yet almost no proposals give receivers analogous control (an exception is NIRA [52]), even though they have the same interests as senders, as noted in §1. Thus, we propose *sink routing*, in which the *destination* chooses, or approves of, the entire interdomain path. ICING's mechanisms naturally enable sink routing.

Security applications. VPNs can be implemented under ICING (an organization gives PoCs only to employees), with the bonus that unauthorized traffic is stopped far upstream. Similarly, an end-host can make firewall-like decisions but see them enforced in or before the Internet core (by naming applications with vnodes). Today many organizations use deep packet inspectors (DPIs); ICING can also enhance these functions, as follows.

Off-path middleboxes. Under ICING, an end-host can direct traffic headed to it through off-path middleboxes. However, unlike in previous work (e.g., [46, 48]), the invocation can be *selective* and *enforced*. Thus, a destination domain can require traffic from unknown sources to go through a third-party DPI or DDoS mitigator (e.g., [42]) but let sensitive traffic travel directly to it.

More exotic policies. Under ICING, a provider can set arbitrary conditions for packet carriage. For example, it can require that all traffic on a particular local vnode has flowed through friendly countries. It can also express analogous policy for the remainder of the path (as in [49]) and mostly (but not completely; see §3.1,§3.4) enforce that policy. It can also issue consent based on factors like whether another entity on the path is a customer and has paid its bill, whether resources are available, etc.

8 Related work

ICING borrows much from many [5, 13, 14, 23, 24, 37, 39, 43, 52, 54], and we have noted those debts throughout. Here, we briefly mention some broader research cur-

rents from which ICING has emerged. First, PoCs are related to network capabilities [6, 43, 50, 54] and Visas [21]. These mechanisms allow receivers to authorize senders but do not fully uphold the principles in §1. (One might wonder if source-routed connection service [44] upholds the principles; unfortunately not: no mechanism constrains packets to follow the agreed path.) Second, the years have seen many routing proposals, from earlier policy frameworks such as [15, 22] to, more recently, the work cited in Table 1. Taken individually, these proposals do not uphold the principles in §1, but taken collectively, they motivated our work. Finally, "clean slate" is now a fashionable trend. Much of this work concerns the application layer, naming, etc. [10, 16, 31, 46] so is orthogonal to our network-layer focus.

9 Discussion and summary

Overhead and feasibility §6 addressed forwarding speed and packet overhead. As noted in the introduction, while ICING's forwarding speed may not seem high in absolute terms, one must calibrate to the hardware platform (on which it runs at almost the same speed as IP forwarding). As for space overhead, it is significant under ICING. But as noted in the introduction, compared to a naive solution, ICING's 42 bytes per participant is a major improvement. And, under jumbo frames, the absolute overhead is negligible. And, even without jumbo frames, our (crude) analysis in §6 indicates that ICING would increase total bandwidth consumption on the Internet only by 22.6%, which might be a fair price for its properties.

Thus, we don't mean to minimize these costs, but our original question was whether ICING is plausible. We believe, tentatively, that it is.

Deployment ICING offers incremental gain if deployed in an edge network or organization. An example is as follows. Today, firewalls' perimeter-based security fails completely once an attacker compromises a single machine on an internal network. With ICING deployed locally, operators can filter traffic not just at the perimeter but between any two hosts, even when they are at different sites, communicating over the legacy Internet. Another question is how to deploy incrementally in the core. Here, we can (again) borrow from Platypus [43]: ICING realms can treat IP as the link layer, rewriting IP source and destination addresses at each ICING hop.

Veto power?! One concern is that ICING shifts away from the Internet's hallowed "default-on" paradigm so would lead to diminished connectivity, network neutrality violations, and unpleasant political consequences, as providers use fine-grained vetoes (or threats of them) to bend communications to their political and commercial wills. On the other hand, connectivity is a powerful economic driver, and ICING also empowers *endpoints*—endpoint control is inherent in the policy principle, and path selection happens in commodity servers—potentially creating competition where today monopoly reigns. In any case, ICING did not create this dialectic, as its control features are not new but merely a union of what has been previously proposed. As noted in our introduction, we think it premature for our community to predict which subset of control features will eventually emerge as dominant, and unwise to embed that prediction in a long-lived architecture, so for now we think it best to provide a neutral platform on which the tussle [17] can play out. This is the role ICING is intended to play.

Future work and summary Our near-term future work is to enhance our evaluation, particularly estimates of ICING on production hardware (§6.5); to implement the unimplemented pieces (§5); and to design a comprehensive replay prevention solution (§3.4). Longer term, our work is to handle providers that cheat by neglecting agreed service levels (§3.4) and to examine how ICING cohabitates with other Internet architecture proposals.

To summarize, we began with the principles in §1 (which really did predate the proposal's particulars!) and sought a feasible architecture to uphold them. We were led to a mechanism that is not perfect: it has some cost and some complexity, though we project technology trends will make it cheaper and better-performing over time. Yet, even if these conjectures are wrong, ICING's properties may be worth its price. Moreover, what we have presented here is not intended to be the last word but rather an existence proof, ripe for optimizations and improvements, that it is feasible to uphold the principles. Indeed, all policy considerations, policy frameworks, and policy functions aside, that our design and hardware implementation upholds the principles means that we have a solution to a formerly unaddressed technical problem: binding a packet to its path with no central authority, in an adversarial environment. This is our most significant technical contribution.

Acknowledgments

Insightful comments and careful reading by Andrew Blumberg, Steve Keckler, and Josh Leners improved this draft.

References

[1] The 32-bit autonomous system number report. `http://www.potaroo.net/tools/asn32/index.html`. Last accessed on 3/2/2009.

[2] Integrated device technology (IDT) quick reference guide. `http://www.idt.com/products/getDoc.cfm?docID=18640144`. Last accessed on 1/30/2009.

[3] NetFPGA: Programmable networking hardware. `http://netfpga.org`.

[4] Digital signature standard (DSS). Federal Information

Processing Standards Publication, November 2008. DRAFT FIPS PUB 186-3.

[5] D. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet protocol. In *SIGCOMM*, Aug. 2008.

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

[7] K. Argyraki and D. R. Cheriton. Loose source routing as a mechanism for traffic policies. In *Proc. SIGCOMM Workshop on Future Directions in Network Architecture*, Sept. 2004.

[8] K. Argyraki and D. R. Cheriton. Network capabilities: The good, the bad and the ugly. In *HotNets*, Nov. 2005.

[9] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Highly secure and efficient routing. In *INFOCOM*, Mar. 2004.

[10] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the Internet. In *SIGCOMM*, Aug. 2004.

[11] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by default! In *HotNets*, Nov. 2005.

[12] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *Advances in Cryptology - EUROCRYPT 2002. Lecture Notes in Computer Science*, pages 384–397. Springer-Verlag, 2002.

[13] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *NSDI*, May 2005.

[14] M. Casado, M. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, Aug. 2007.

[15] D. Clark. Policy routing in internet protocols. RFC 1102, May 1989.

[16] D. Clark, K. Sollins, J. Wroclawski, and T. Faber. Addressing reality: An architectural response to demands on the evolving Internet. In *Proc. SIGCOMM Workshop on Future Directions in Network Architecture*, Aug. 2003.

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

[18] M. Crawford and C. Huitema. DNS extensions to support IPv6 address aggregation and renumbering. RFC 2874, Network Working Group, July 2000.

[19] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec. 1959.

[20] D. Estrin, T. Li, Y. Rekhter, K. Varadhan, and D. Zappala. Source demand routing: Packet format and forwarding specification (version 1). RFC 1940, May 1996.

[21] D. Estrin, J. Mogul, and G. Tsudik. VISA protocols for controlling inter-organizational datagram flow. *IEEE JSAC*, 7(4), May 1989.

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

[23] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica. Pathlet routing. In *SIGCOMM*, Aug. 2009.

[24] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *ACM CCR*, 35(5), Oct. 2005.

[25] S. Guha and P. Francis. An end-middle-end approach to connection establishment. In *SIGCOMM*, Aug. 2007.

[26] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *OSDI*, Dec. 2004.

[27] P. Hawkes and C. McDonald. Submission to the SHA-3 competition: The CHI family of cryptographic hash algorithms. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/2/2c/Chi_submission.pdf`.

[28] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Measurement and classification of out-of-sequence packets in a Tier-1 IP backbone. In *INFOCOM*, 2003.

[29] H. T. Kaur, A. Weiss, S. Kanwar, S. Kalyanaraman, and A. Gandhi. BANANAS: An evolutionary framework for explicit and multipath routing in the internet. In *Proc. SIGCOMM Workshop on Future Directions in Network Architecture*, Aug. 2004.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. on Computer Systems*, Aug. 2000.

[31] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *SIGCOMM*, Aug. 2007.

[32] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203–215, Feb. 2007.

[33] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet routing convergence. *ACM/IEEE Transactions on Networking*, 9(3):293–306, June 2001.

[34] M. Little. Goals and functional requirements for inter-autonomous system routing. RFC 1126, October 1989.

[35] X. Liu, X. Yang, and Y. Lu. To filter or to authorize: Network-layer DoS defense against multimillion-node botnets. In *SIGCOMM*, Aug. 2008.

[36] R. Mahajan, D. Wetherall, and T. Anderson. Mutually controlled routing with independent ISPs. In *NSDI*, Apr. 2007.

[37] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, Dec. 1999.

[38] A. T. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage. Fatih: Detecting and isolating malicious routers. In *IEEE DSN*, June 2005.

[39] R. Moskowitz and P. Nikander. Host identity protocol (HIP) architecture. RFC 4423, May 2006.

[40] R. Perlman. *Network layer protocols with Byzantine robustness*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[41] R. Perlman. Routing with Byzantine robustness. Technical Report TR-2005-146, Sun Microsystems, Aug. 2005.

[42] Prolexic Technologies, Inc. `http://www.prolexic.com`.

[43] B. Raghavan and A. C. Snoeren. A system for authenticated policy-compliant routing. In *SIGCOMM*, Sept. 2004.

[44] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching. RFC 3031, Network Working Group, Jan. 2001.

[45] M. Scott. Miracl library. `https://www.shamus.ie/index.php?page=Downloads`.

[46] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *SIGCOMM*, Aug. 2002.

[47] The Cooperative Association for Internet Data Analysis (CAIDA). Packet size distribution comparison between internet links in 1998 and 2008. `http://www.caida.org/research/traffic-analysis/pkt_size_distribution/graphs.xml`.

[48] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, Dec. 2004.

[49] W. Xu and J. Rexford. MIRO: Multi-path interdomain routing. In *SIGCOMM*, Sept. 2006.

[50] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *Proc. IEEE Symposium on Security and Privacy*, May 2004.

[51] A. Yaar, A. Perrig, and D. Song. StackPi: New packet marking and filtering mechanisms for DDoS and IP spoofing defense. *IEEE JSAC*, 24(10):1853–1863, Oct. 2006.

[52] X. Yang, D. Clark, and A. W. Berger. NIRA: A new inter-domain routing architecture. *ACM/IEEE Transactions on Networking*, 15(4), Aug. 2007.

[53] X. Yang and D. Wetherall. Source selectable path diversity via

routing deflections. In *SIGCOMM*, Sept. 2006.
[54] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, Aug. 2005.

A Implementation of PRF-96

The dataplane protocol in §3 relies on pseudo-random digests at several places, notably in the computation of the cryptographic material in PoCs, and in the derivation of the components that get aggregated in the entries of the authentication vector V^i. In principle, our design could employ a single, variable-input-length pseudo-random function (PRF) family like PMAC [12] to compute all digests. To maximize forwarding rates and minimize the number of entries necessary in the hardware cache, however, it is expedient to adopt different PRF constructions at different places.

Specifically, the cryptographic material in a PoC is calculated using PMAC as soon as the (variable-length) path and timestamp information are read from the packet header. The authentication vector components, instead, depend on the (longer) packet's payload, and are computed based on the hash-then-MAC paradigm.

Because the same hash function is used in conjunction with different keys for the MAC stage, it is important that the hash function be collision resistant. As collision-resistant hash function, we use CHI [27]. CHI is a Round-1 SHA-3 candidate that can produce a 256-bit hash. We chose CHI because it featured both fast evaluation in hardware (by virtue of its highly parallel design) and promise of strong security (by virtue of its adherence to the best practices in hash function design). CHI was not selected for Round-2 of the SHA-3 competition. While no significant cryptoanalytic attack has been discovered against CHI, we intend to stay current with the SHA-3 competition, and replace CHI in our system with one of the standing SHA-3 candidates in the next protocol version.

Our fixed-input-length PRFs are summarized in pseudo-code in Figure 8. We chose as our workhorse two rounds of CBC-MAC with the AES block cipher. To save packet space, the resulting 128 bits are chopped down to 96 bits, which is believed to provide full 96-bit security.

Note that while §3 refers to "PRF-96", the ICING protocol actually uses two separate PRF-96 functions: PRF1-96, which the realms use to calculate the A_i, and PRF2-96, which the realms use to construct, verify, and transform V^i. The notation in §3 does not depict the true parameters to the call to PRF2-96. The true parameters reflect a slight technical complication resulting from the need to minimize the cache requirements on the forwarding hardware. Namely, rather than having two keys, one for each direction (the two directions being R_i proving provenance to R_j and R_j proving provenance to R_i), each pair of realms (R_i, R_j) shares a single MAC key $k_{i,j} = k_{j,i}$

function CBC2MAC128(k, blk[0..255])
  X = AESENC(k, blk[0..127])
  Tag = AESENC(k, $X \oplus block$[128..255])
  return Tag

function PRF1-96(k, blk[0..255])
  Tag = CBC2MAC128(k, blk)
  return Tag[0..95]

function PRF2-96(R_{prover}, R_{verifier}, $k_{\text{prover,verifier}}$, blk[0..254])
  $t = (R_{\text{prover}} < R_{\text{verifier}})\ ?\ 1 : 0$
  return PRF1-96(k, $t||blk$)

Figure 8—Pseudocode for the fixed-input-length PRFs used to compute pseudo-random hashes in the dataplane protocol of §3. CBC2MAC128 consists of two rounds of CBC-MAC with the AES block cipher; PRF1-96 is derived from CBC2MAC128 by chopping its output to the bottom 96 bits; and PRF2-96 is derived from PRF1-96 via domain separation.

that they use as input to PRF2-96 in both directions. To prevent hashes computed for one direction from interfering with those of the other direction, we employ a "domain separation" trick: realms prepend a bit that encodes the direction to the value being MACed (which is the output of a hash function). Although this requires adjusting the output of the hash stage by one bit, the bit security of the collision resistance property remains comfortably large (127-bit security).

B Control plane details

This appendix details ICING's control plane, filling in details that were omitted from §2 and §4. In §B.1 we detail client configuration. We then detail sIRP (§B.2) and path finding under sIRP (§B.3). We also discuss extensions to sIRP (§B.4) and other approaches to routing and path construction (§B.5).

B.1 Client configuration

Before a client device can start sending traffic on ICING networks, it must discover paths and gain permission to use those paths. Specifically, a client needs to find out which realm it has connected to and which vnodes it can access. A client also needs a path to a local path server and permission to use that path. This path server is used as a gateway through which the client negotiates paths to other path servers and hosts outside its own realm. Clients receive this information by using a service similar to DHCP.

When a client connects to a network, it contacts an ICING configuration server. The server sends the client the realm's public and private keys, a vnode to use and a path to a local path server. This path is accompanied by either the PoCs that correspond to the path or PoC-keys so that the client can build PoCs on its own.

If the client has declared a qualified name, it passes this name to the configuration server, which in turn

passes it on to the realm's path servers, so that other hosts can discover it. The configuration server also notifies local sIRP servers that the client has connected and which vnode it is using, so that topology updates (§B.2) can be forwarded to the client.

B.2 sIRP

This section describes how path servers and clients use sIRP to discover subsets of the network topology.

B.2.1 Approach

sIRP (the simple ICING routing protocol) is a link-state routing protocol that pushes updates *downward*, from transit providers to customers. It restricts routes to those that are valley-free. While this goal may seem restrictive, sIRP is meant to be only an existence proof; the data structures that we use can be extended in order to enforce more complex policies, as we briefly cover in §B.4 and §B.5.

sIRP update messages are composed of statements that expose who is authorized to send traffic via the realm. For example, realm *A* is permitted to send traffic to realm *C* via realm *B*. sIRP uses *consent certificates* (detailed in §B.2.2) to express these authorization statements. Each certificate is signed. sIRP update messages are composed of the certificates that a realm builds along with the certificates that were received from the realm's neighbors.

Each sIRP server is configured so that, for each neighbor, it knows the neighbor's public key and whether the neighbor is a provider, peer or customer of the realm. This information ensures that only valley-free routes are built: sIRP sends routing updates only to neighbors that are customers. As a result, for a packet to flow through a realm, it must be destined for, or originate from, a customer (or a customer's customer, etc.).

By pushing certificates down the provider chain, sIRP gives each realm the ability to reach the network "core". We assume that, like today's Tier-1 ISPs, the realms in the core are well-connected to each other. As a result, any two edge networks will have certificates that permit them to send traffic to each other via a common realm in the core. Moreover, this path is guaranteed to be valley-free. The reason is that a realm constructs consent certificates expressing that traffic travels between a customer and a provider of the realm, or a peer and a customer of the realm. A realm never creates certificates consenting to traffic that transits over the realm between a provider and a peer or between a peer and another peer. Thus, routes that are not valley-free will not have a full set of consent certificates and hence will not appear legitimate.

B.2.2 Data structures

Consent certificates The consent certificate, shown in Figure 9, is the key data structure of the ICING control plane. It encapsulates the policies that a realm maintains. A certificate names three realms: the issuing realm, the previous hop, and the next hop. A certificate can be interpreted as permitting traffic to flow from the previous hop to the next hop via the issuing realm.

```
struct realm_vnode_mask {
  key_t public_key;
  uint vnode;
  uint mask;
};
struct certificate {
  realm_vnode_mask issuer;
  realm_vnode_mask previous;
  realm_vnode_mask next;
  time_t created;
  time_t expiry;
  uint metric;
  sig_t signature; // uses issuer's private key
};
```

Figure 9—The representation of an ICING consent certificate. Either `previous` or `next` may have a public key made up of all ones, signifying that it is a membership certificate.

A sIRP certificate also contains a metric, which can be used when evaluating sets of possible paths, and a creation and expiration time. Each certificate is signed using the realm's private key.

Reducing the number of signatures As an optimization, sIRP uses vnodes to reduce the number of certificates that it needs to issue. Instead of issuing consent certificates for each pair of neighbor realms, a realm issues a *membership certificate* that authorizes the neighbor to use a range of vnodes. By assigning all neighbors in a given class the same membership certificate, a realm need only issue a number of certificates linear in the number of neighbors (instead of quadratic).

A membership certificate has the same format as a consent certificate, except that either the previous or next hop public key field is all ones. The issuer field indicates which vnodes the neighbor is authorized to use through the use of a mask. When two membership certificates are taken together, vnodes that are shared are permitted to transit traffic between the two realms. For example, a realm can issue a customer a membership certificate permitting it to use vnodes 64–256, and issue a provider a certificate permitting it to use vnodes 128–512. These two certificates, taken together, are interpreted to mean that traffic can flow from the customer to the provider (and vice versa) on the common vnodes, namely vnodes 128–256.

```
1: function BUILDCERTIFICATES(Customers, Peers, Providers)
2:     Certificates = {}
3:     for each c ∈ Customers do
4:         if c is UP then
5:             for each t ∈ Providers do
6:                 if t is UP then
7:                     cert = new Certificate(c, t)
8:                     Certificates += cert
9:                 end if
10:            for each p ∈ Peers do
11:                if p is UP then
12:                    cert = new Certificate(c, p)
13:                    Certificates += cert
14:                end if
15:        end if
16:    return Certificates
```

Figure 10—Pseudocode that builds sIRP consent certificates for each neighbor realm. This is run when a sIRP participant starts up, and whenever a neighbor connects or disconnects. The pseudocode uses only basic consent certificates, but membership certificates can be used as an optimization.

B.2.3 Pseudocode

Building certificates A sIRP participant in a realm (a general-purpose server) is configured with a list of the realm's neighbors. Each neighbor is classified as a customer, peer, or provider. The sIRP participant runs the pseudocode in Figure 10 to create the consent certificates that make up sIRP update messages. BUILDCERTIFICATES is also run when a neighbor goes down or comes back up. While the pseudocode uses only basic consent certificates, membership certificates can be used to reduce the number of certificates that must be created.

Sending updates A realm sends updates to its customers at regular intervals. Updates are also sent whenever a neighbor disconnects or reconnects. This ensures that customers are aware when a link is no longer available. Realms use the pseudocode in Figure 11 to send updates to their customers. Each update includes all the certificates received from neighbors, along with the certificates the realm builds itself using BUILDCERTIFICATES.

B.3 Path construction under sIRP

In this section we discuss how senders identify paths under sIRP. We begin with the requirements for a path server, and then detail our approach.

B.3.1 Requirements

An ICING path server is a host's primary method for finding paths. A path sever takes some set of sub-paths from a client, along with a destination realm, and should return at least one path to that destination. A path server should also ensure that the paths returned do not conflict with

```
1: function UPDATE(Customers, Peers, Providers)
2:     Update = BuildCertificates(Customers, Peers, Providers)
3:     for each n ∈ (Customers ∪ Peers ∪ Providers) do
4:         if n is UP then
5:             u = n.mostRecentUpdate
6:             Update += u
7:         end if
8:     for each c ∈ Customers do
9:         if c is UP then
10:            Send Update to c
11:        end if
```

Figure 11—Pseudocode that sends sIRP update messages. Updates are only sent to customers of a realm, but incorporate certificates received from any neighbor.

other realms' published policies, for instance, by using available consent certificates. While not strictly necessary, doing so will help prevent requests for PoCs from being sent for paths that are not permitted by realms on the path.

Path servers do not only find paths. They can also act as name servers; if they do, path servers must maintain mappings from qualified names to realms. Path servers may also perform the role of consent servers for a realm, returning both the path and some of the PoCs for the path. Realms may choose to delegate their PoC issuing responsibilities to a path server in another realm to speed up requests for paths (such delegated PoCs are analogous to DNS's glue records). For this reason, path servers must be made aware of the remote realm's policies to ensure that PoCs are only issued for paths that agree with the realm's policies.

B.3.2 Approach

In our implementation, path servers find paths and act as name and consent servers. Path servers are configured with mappings from names to realms, and receive topological data from sIRP. So that path servers can issue PoCs, they are also configured with PoC keys.

Path server hierarchy Path servers do not necessarily exist in every realm. Path servers are organized in a hierarchy based on the names that they serve. As there are root name servers in DNS today, so too are there root path servers. Non-root path servers are configured with paths to root servers, as well as PoC keys for the paths.

When a client wishes to contact a destination, the client contacts its local path server, providing the destination's qualified name. If the path server does not know the domain of the destination, it instead returns a path and PoCs that enable the client to reach a root path server. The client then queries the root server, working its way down the hierarchy. §B.3.3 details how the client navi-

```
1: function GETPATHTOHOSTBYNAME(name, partial_-
   paths)
2:     ps = PS0
3:     repeat
4:         (fullpath, fullcerts, pocs, ps) = ps.GetPath(name,
   partial_paths)
5:     until ps is null
6:     return fullpath, fullcerts
```

Figure 12—Pseudocode for GETPATHTOHOSTBYNAME. This function is called by a client in order to perform a name-to-path lookup. The call ps.GetPath() is an remote procedure invocation that uses the function SEND.

gates this hierarchy.

Client interface Clients interact with path servers using a remote procedure call, GETPATH (described in detail in §B.3.3). GETPATH returns paths to a host H; H is either the requested destination or the next path server in the hierarchy. The call also returns a subset of the PoCs necessary to reach H.

Clients query a path server using the qualified name of a destination. A path server consults its mapping from names to realms to see if it knows the destination. If the destination is known, the server attempts to find a path to the destination's realm. If, however, the path server administers the domain but does not know the destination, it returns a failure code to the client. If the server does not administer the domain, it finds and returns a set of paths (and PoCs) to a root server.

Our approach to path finding, taken from [52], requires a path server to find an intersection between the set of realms that the client can reach and those that the destination can reach. A client therefore provides a path server with a set of partial paths, along with matching consent certificates (§B.2.2). The server then constructs candidate paths using this information. The algorithm for building candidate paths is given in Figure 13.

After constructing candidate paths, the path server uses consent certificates to verify whether all realms along the path will consent to it. If the path is not valid, the path server simply discards it and continues to the next candidate path. If the path is valid, however, the path server can then proceed to issue PoCs for some of the realms on the path. It returns to the client the path, PoCs, and certificates.

Reaching the core We refer to the "core" of the Internet as the set of strongly connected realms, similar to Tier-1 ISPs. This definition is taken from [52]. We now argue that for any two realms A and B, there is some realm C in the core of the Internet that both A and B can reach and that the process of path construction will identify this realm. sIRP ensures that each realm has a path to the Internet core: every realm creates consent certificates between its providers and its customers, and pushes these certificates to customers. Their customers to do the same, etc. As a result, every realm gets a chain of certificates that reach at least one realm in the core. Since core realms connect to each other (by assumption) there is such a realm C. Meanwhile, a path server is guaranteed to identify this realm because it examines all legitimate paths from sender to destination.

```
1: function GETPATH(dest, cpartials)
2:     if dest is known then
   //cpartials are the partial paths supplied by the client
3:         dpartials = GetPartialPaths(dest)
4:         intersect = FindIntersection(cpaths, dpaths)
5:         for each d, d_certificate ∈ dpartials[intersect] do
6:             for each c, c_certificate ∈ cpartials[intersect] do
7:                 path = c||d
8:                 certificates = c_certificate||d_certificate
9:                 if VerifyPath(path, certificates) is True then
10:                    pocs = GetPoC (path, certificates)
11:                    Paths += (path, certificates, pocs)
12:                end if
13:        return Paths
14:    else if dest is known then
15:        return DestinationUnknown
16:    else
17:        return GetPath(root_server, cpartials)
18:    end if
```

Figure 13—Pseudocode builds paths from partial paths. An intersection realm is found from set of partial paths provided by the client and those from the destination. The two sets of partial paths are combined; if the path verifies, it is returned to the client with PoCs.

B.3.3 Pseudocode

Navigating the path server hierarchy A client uses GETPATHTOHOSTBYNAME to query path servers. The pseudocode for the function is shown in Figure 12. A client uses the remote procedure call GETPATH on a local path server, querying on the qualified name of the host. The path server either returns paths and PoCs to the destination, or paths and PoCs to a path server that may know the mapping. The client queries each successive path server until a path to the destination is returned.

Path building The pseudocode for the remote procedure call GETPATH is shown in Figure 13. GETPATH finds a realm that both the client and the destination can reach, and constructs a path by concatenating both paths to the common realm. The path server then verifies that the path is valid by using consent certificates in the partial path data structures. If the path is valid, the path server returns the path, the certificates that accompany the path, and PoCs for some of the realms on the path.

B.4 Extensions

sIRP implements a single simple forwarding policy for all realms. In this section we describe how consent certificates can express more complex policies.

Path filters A *path filter* can be added to a consent certificate in order to restrict the use of a consent certificate to paths of a specific form. Using a filter allows a realm to express policies that concern the entire end-to-end path. A path filter field of a consent certificate is a regular expression: it is made up of public keys and wild cards (such as $*$). If a path does not match the filter, the consent certificate may not be used to obtain a PoC for the path.

Required vnodes Realms may want to express policies that involves realms that are not on the path the packet takes. For example, a research center's ISP may want to restrict traffic to that coming from universities and machines that a university "vouches for." In order to accomplish this, a realm R can require a client to provide a membership certificate for a *different* realm S. R adds the pair (S, v), where v is a vnode, to its own consent certificates' *required vnode* field. Unless this consent certificate is accompanied by a membership certificate that entitles a realm on the path to use the pair (S, v), R will not issue a PoC for the path.

B.5 Other approaches

At the end of §4, we briefly mentioned alternatives to sIRP. Here we go into slightly more detail.

ICING-BGP-AIP ICING permits a routing and path construction protocol that is equivalent to BGP in terms of policies expressed and policy privacy but, as a bonus, binds the data plane to the control plane. We now cover the high-level approach, first describing topology propagation, then path selection, and then enforcement.

For topology propagation (and implicit route selection), realms run BGP between themselves, signing their messages (as in [5]); these signed messages are equivalent to consent certificates. While scalability of routing might seem to be a concern because there are no prefixes to aggregate, the authors of [5] show that running BGP on a flat namespace of AS identifiers is feasible. One subtlety under ICING is that a realm, in the general cause, uses a separate vnode for each (prev_hop, path_-suffix) pair and discloses the PoC key for that vnode in the BGP advertisement.

The local path server in a realm is a BGP participant and, for every destination realm, stores the (realm,vnode) sequence needed to reach that destination, along with the disclosed PoC keys. Observe that the state required in commodity path servers is only a linear factor more than AIP [5] requires in *routers*: under AIP, each router stores a map from all destination realms to next hops. And, AIP has already shown that this state is feasible in routers, so a linear factor more should not be overly burdensome in commodity servers.

Path selection works as follows. When the sender makes a request for a destination, the local path server identifies the needed AS path and uses the disclosed PoC keys to mint the PoCs needed to use the path. Observe that this approach upholds the fifth mechanism principle: in using BGP's policies, ICING does not make clients or intermediate realms pay for PoC retrieval and other control plane costs.

We now cover enforcement. Under this scheme, a realm's policies consist of bindings between previous hops and path suffixes (i.e., an adversarial sender should not be able to use a given realm for arbitrary transit between arbitrary neighbors). Because the path is manifest in the packet, and because the packet is bound to the path, an adversarial sender cannot force a realm to violate its own policies—even though a realm's PoC keys are publicly known. Moreover, because a realm, in the general case, gives each binding a separate vnode, the realm can give different classes of service to each binding. An adversarial sender cannot cheat this process by using the wrong vnode to transit the provider because the provider knows which vnodes connect which pairs of neighbors.

ICING-Pathlets Using a similar approach to the one above, ICING can provide the equivalent of Pathlet routing [23]: realms embed pathlets in consent certificates. Indeed, ICING borrowed vnodes from Pathlet routing in the first place, so ICING can naturally express the policies that Pathlet routing can. The arguments about the fifth mechanism principle and enforcement are similar to the approach given just above, and to avoid further tedium, we omit them.