PATH-POLICY COMPLIANT NETWORKING AND A PLATFORM FOR HETEROGENEOUS IAAS MANAGEMENT

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jad Naous

March 2011

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Nick McKeown, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Mazieres, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Michael Walfish

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

Abstract

This thesis is in two parts.

Path-Policy Compliant Networking

The Internet gives little control to the end-points of a communication over the actual interdomain path taken by that communication through the network. Such a control can be quite useful. Unfortunately, unchecked, this capability can threaten providers' viability. Providers want to ensure that they are compensated for their services and that paths that end-points choose comply with the providers' policies. Even if providers did want to give such control to senders and receivers, the Internet does not give providers such ability. Not only does the Internet not have a mechanism to enable senders and receivers to *specify a choice of path* that complies with providers' transit policies, it does not have a mechanism to *enforce that choice* and thus the policies of senders, receivers, and transit providers.

The ICING project aims to fill that void in three stages. The first stage is a packet forwarding mechanism, ICING-PVM, that provides the missing capabilities above. ICING-PVM allows a sender to specify the path to use for its traffic, but it requires the sender to obtain authorization from all entities on that path before using it. Upon receiving a packet, each entity on the path can verify that it had previously approved the packet's path, and that the packet is following that path. Thus ICING-PVM separates policy from mechanism, and authorization from authentication.

The second stage is an overlay network, ICING-ON, that enables senders and receivers to specify overlay waypoints and in-network services for traffic between them. ICING-ON uses DNS as a bootstrapping mechanism for finding paths and obtaining authorization.

But an overlay network side-steps many of the complexities of a network architecture that can replace the Internet, such as how to obtain authorization to obtain authorization. So, the third stage is a network architecture, ICING-L3, that enables a sender to discover and choose paths towards a destination, without using IP as a crutch.

A Platform for Heterogeneous IaaS Management

Many new innovations in networking cannot be deployed today because they require a significant change to the Internet infrastructure or because their scale is too large. To overcome this hurdle, the National Science Foundation (NSF) started GENI, the Global Environment for Network Innovations.

GENI is a federation of existing and new Infrastructure-as-a-Service providers that contribute computing and networking resources to be used for deploying new applications and experiments. Some of these IaaS providers are PlanetLab, Emulab, and campuses running OpenFlow networks. GENI will enable a researcher to obtain a coherent *slice* of infrastructure across resource types and providers.

Building the management framework that enables researchers to do so is an ongoing concern. The community is converging towards a unified API across all types of resources and providers. But it is not clear if such a solution will be the correct one for the future. A rigid unified API may not allow the community to experiment with different design choices and deployment models.

This thesis discusses an alternative design, Expedient. Expedient is a management platform for GENI that follows two principles: *reuse* and *pluggability*. Expedient leverages existing Web technologies to quickly prototype and experiment with new features and user interfaces. It is built as a pluggable website that uses heterogeneous APIs to communicate with each IaaS provider, and uses as a starting design point a centralized deployment model that enables federated management. Expedient has enabled us to quickly build infrastructure to slice OpenFlow networks and to demonstrate coherent continuous slices across PlanetLab IaaS providers and OpenFlow network IaaS providers.

Acknowledgements

My parents, just like any Middle Eastern parents, wanted me to become a doctor, and I always said that it would take too much time. Little did they and I know that not only will I become a doctor, but the wrong kind of doctor. First and foremost, I want to thank my parents for their unrelenting support and love. My parents have sacrificed more than I think I will ever be able to, so that my brothers and I can be happy and successful. I only hope one day I will be able to repay them this incredible debt and be strong enough to do the same for my children.

When I came to Stanford, I was set on graduating with only a Master's in Computer Science. But there were so many interesting things happening in Stanford, so many interesting people to talk to, and my advisor Prof. Nick McKeown. If it had not been for Nick, I would not have had many of the experiences that have shaped my current career path. Nick has taught me much about being an "adult", and most importantly, how to have long-term focus and have an impact. So, thank you, Nick, for being such a powerful role model, for helping me define the term "success", and for giving me the opportunities that have led me to where I am today.

I wish to thank Prof. David Mazières for his incredible dedication to his students. And I really do mean incredible. I have never heard any of David's students ever complain about him. Very few people with David's other-worldly intelligence are as willing to listen to students and give them the benefit of the doubt. Thank you, David, for helping me form the greater part of my thesis at a dark time during my student career.

I met Prof. Michael Walfish when he was a postdoc at Stanford. Mike has taught me much about being careful, about writing, and about being a graduate student.

Thank you, Mike, for being a great mentor.

My work on ICING and Expedient was only possible because of my collaborators. For ICING, thanks go to Michael Walfish, David Mazières, Antonio Nicolosi, Scott Shenker, Arun Seehra, and Mike Miller. For Expedient, thanks go to Guido Appenzeller, Peyman Kazemian, Nick McKeown, Guru Parulkar, Srini Seetharaman, and Rob Sherwood.

Finally, I want to thank my friends and colleagues at Stanford who have made my journey enlightening both at an intellectual and spiritual level. Thank you, Archan Padmanabhan, Corina Dumitrescu, Zuley Rivera Alvidrez, Guido Appenzeller, Neda Beheshti, Andrea Bittau, Prof. Dan Boneh, Martìn Casado, Adam Covington, Jonathan Ellithorpe, David Erickson, Yashar Ganjali, Glen Gibb, Daniel Giffin, Nikhil Handigol, Brandon Heller, Abdul Kader Kabbani, Peyman Kazemian, Sean Kendall, Masa Kobayashi, Prof. John Lockwood, Diego Ongaro, Stephen Rumble, Srini Seetharaman, Raunaq Shah, Rob Sherwood, Ryan Stusman, Paul Tarjan, David Terei, David Underhill, Greg Watson, Paul Weissmann, Tatsuya Yabe, Kok-Kiong "KK" Yap, Yiannis Yiakoumis, Prof. Nickolai Zeldovich, and James Hongyi Zeng.

Preface

My research work at Stanford can be divided into two main projects: ICING and Expedient. The organization of this dissertation reflects this division. But even though both projects are quite independent of each other, they both share a common theme: they both attempt to provide a field over which the concerns and policies of various stakeholders can play out.

The ICING project aims to build a network architecture that empowers the end-points of a communication by giving them control over the path used for the communication. The challenge is providing this control without violating the policies of the providers carrying the communication's traffic and ensuring that the path is followed. ICING consists of a number of sub-projects: ICING-PVM, ICING-ON, and ICING-L3.

Expedient is an answer to the following question: How can users manage resources across a federated set of Infrastructure-as-a-Service (IaaS) providers? These resources may be heterogeneous, consisting of different types of network, compute, or other resources, they may be sliced, virtualized, or fully delegated, and they may cross trust boundaries. Expedient provides a platform that gives users access to all resources using one set of credentials, does not require infrastructure developers to change their systems, and allows infrastructure providers to enforce their policies concerning the use of their resources. This dissertation discusses Expedient in the context of GENI, a federated set of IaaS providers with the goal of providing resources for experimental network research.

The dissertation is divided into two parts, each of which can stand independently of the other: ICING and Expedient. Aspects of ICING have previously appeared in [146] and [123].

To my parents, Toufik and Nahida Naous.

Contents

List of Figures

Part I

ICING

Chapter 1

Introduction

The current Internet provides a simple delivery mechanism: we put destination addresses in packets and launch them into the network. We leave the network to decide the path that our packets take and the intermediate providers that the path passes through. Even network operators have little control over the paths that packets actually take toward them, or after leaving them. There are times, however, when senders, receivers, and operators would prefer to control packets' paths—and be sure that their preferences are enforced.

For instance, if the *fact* of a communication (not just its content) between sender and receiver is sensitive, they might want to select network providers that they trust to be discreet. Or an enterprise might want a guarantee that the packets that it receives have passed through several services, such as an accounting service and a packet-cleaning service. Or a company might want fine-grained control over which providers carry which traffic between its branch offices. But giving endpoints such control might threaten providers' economic security, so providers, too, might want to be sure that they are carrying traffic to or from customers. Or providers might want to make sure they are only carrying traffic from friendly nations.

The functionality above does not exist in the Internet today, though there are proposals in the literature that address various aspects of the problem. However, there is no general-purpose mechanism that *enforces* these policies (short of allocating dedicated connections, which is expensive). And even if such a mechanism did exist, it is not clear how a network architecture can use it.

The ICING project aims to fill that void in three stages. First, we will describe a new networking primitive that provides the missing mechanism above and describe an implementation in Chapter 2. Then, we describe how this primitive can be used to build an overlay in Chapter 3. Finally, in Chapter 4, we describe how to use this primitive to build a network architecture to replace IP in Chapter 4.

Chapter 2

Enforcing path policies with ICING-PVM

2.1 Introduction

As mentioned in Chapter 1, the Internet does not have a general mechanism that allows the endpoints of a communication (and the service providers in between) to control a packet's path and be sure that the path is actually followed. Here, we describe a new networking primitive that we call a PVM (Path Verification Mechanism) that provides this missing mechanism. Some uses of a PVM are described in Section 2.6; here, we concentrate on its technical aspects.

A PVM is a mechanism for packet *forwarding* (sending a packet to its next hop) as distinct from topology discovery and path selection, or *routing*. A PVM provides two properties:

1. **Path Consent**: Before a communication, every entity on the path of the communication (including the sender and receiver) or a delegate of that entity consents to the use of the whole path, based on the entity's or the delegate's particular policy.
2. **Path Compliance**: On receiving a packet, every entity can verify (1) that it or its delegate had approved the packet's purported path, and (2) that the packet has followed that path so far.

Realizing a PVM is a challenging technical problem: when a packet arrives at a node, how can the node be sure that the packet followed an approved path? To illustrate the difficulty in achieving Path Consent and Path Compliance, we give three strawman designs. First, we can centralize policy in a *controller* that knows all entities' policies *a priori*. Then, at connection setup time, if the proposed path matches the entities' policies, the controller enables communication by installing state in every participating entity. This design is reminiscent of Ethane [52], which is designed for enterprises. The centralized model, however, does not fit today's federated, decentralized Internet.

A second approach is to decentralize, using the technique of self-certifying names [117, 28]: every entity mints a public key, and packets contain signed logs that prove to every entity along the path that the packet is following its prescribed path(e.g., [43, 51]). Such a design works in principle but would not perform well in practice: public key signatures are either too large, too expensive to compute, or both, for this approach to be feasible at line rate in edge networks (let alone backbones). For instance, signatures based on Merkle Hash Trees can be relatively fast, but require a few kilobytes per signature [122]. On the other hand, Elliptic Curve Digital Signature Algorithm (ECDSA) signatures are small (e.g., 163 bits), but can take on the order of milliseconds to compute [111].

A third approach is to replace public key cryptography with symmetric key cryptography, which is feasible at line rate (a similar proposal for a different purpose was made in [38, 37]). Unfortunately, this approach requires quadratic configuration state for pairwise shared keys and overhead that is quadratic in path length: each hop creates a unique proof for each other hop on the path. Meanwhile, it is not clear how to originate the keys nor how to manage the state (a limitation implicitly acknowledged in [38, 37] and proved as a lower bound in [82, 41]).

As an alternative to the approaches above, this chapter presents ICING-PVM. ICING-PVM is an existence proof of a PVM that respects the Internet's decentralized nature, is amenable to an affordable line-rate hardware implementation, and does not require quadratic configuration state. We implemented ICING-PVM on NetFPGA [10], achieving a minimum throughput of 3.3 Gbits/s at an equivalent gate cost of 54%

more than a simple IP forwarder running at 4 Gbits/s; thus, per unit of throughput, our ICING-PVM implementation costs 86% more than IP. Our evaluation further suggests that, if implemented on a custom ASIC (as in a modern router), ICING-PVM would scale to backbone speeds at acceptable cost[1].

Chapters 3 and 4 describe how ICING-PVM can be used in an overlay ([29, 143, 101, 136, 149, 87, 63] and at layer 3 as a replacement to IP and illustrate the interface to ICING-PVM. We treat related work later in the Chapter 5. For now we just note that some of ICING-PVM's components are reminiscent of, or inspired by, prior mechanisms, and ICING-PVM can enforce many previously proposed policies. However, no proposal that we are aware of offers both Path Consent and Path Compliance (save one [43, 51]), and no proposal offers these two properties in an environment that is adversarial, high-speed, and federated. More specifically, this chapter describes the following contributions:

- A new network security primitive, the PVM.
- The design of an efficient PVM, called ICING-PVM.
- A fast and affordable hardware implementation of ICING-PVM.
- A packet header format and an optimized cryptographic construction that demonstrate the plausibility of rich per-packet cryptography at network line rates.

Next, we describe ICING-PVM, first giving an overview (Section 2.2), and then giving the details of its design and attack-resistance (Section 2.3). We then describe our hardware implementation and evaluation (Sections 2.4 and 2.5). We give several other example uses of ICING-PVM (Section 2.6) and then reflect on ICING-PVM (Section 2.7).

[1]Recent work [65] has examined high-speed software forwarders. However, backbone forwarders seem likely to continue to require dedicated hardware for the medium-term future: general purpose machines still consume more power and require more rack space than special purpose hardware with the same throughput. If needed, our design can run efficiently on a multicore general-purpose processor.

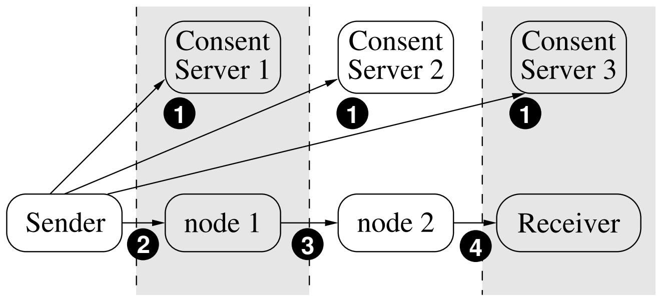

Figure 2.1: ICING-PVM's components and forwarding steps. ❶ In the general case, the sender gets PoCs from the consent servers of all nodes on the path (in practice, a consent server can delegate PoC-issuing, making this step lightweight). ❷ The sender creates and sends the packet to the first ICING-PVM node, having used the PoCs to construct tokens that ❸ each forwarder verifies and transforms for its successors until ❹ it arrives at the receiver.

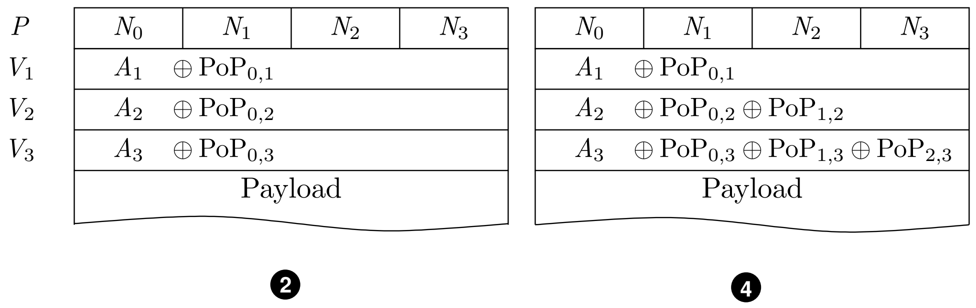
P
V_1
V_2
V_3
N_0 N_1 N_2 N_3
$A_1 \oplus \text{PoP}_{0,1}$
$A_2 \oplus \text{PoP}_{0,2}$
$A_3 \oplus \text{PoP}_{0,3}$
Payload
❷
N_0 N_1 N_2 N_3
$A_1 \oplus \text{PoP}_{0,1}$
$A_2 \oplus \text{PoP}_{0,2} \oplus \text{PoP}_{1,2}$
$A_3 \oplus \text{PoP}_{0,3} \oplus \text{PoP}_{1,3} \oplus \text{PoP}_{2,3}$
Payload
❹

Figure 2.2: Simplified ICING-PVM packet at steps ❷ and ❹ from Figure 2.1. Two crucial header fields are the path (P) and the verifiers (V_j's). The sender (N_0) initializes the verifiers with path authenticators (A_j's) derived from the PoCs and the packet content. Each node N_i checks its verifier (V_i) and updates those for downstream nodes (V_j for $j > i$) to prove that it passed the packet. PoP$_{i,j}$ is a proof to N_j that N_i has carried the packet.

2.2 Overview of icing-pvm

We now describe ICING-PVM at a high level, including its threat model, deferring design details to Section 2.3.

2.2.1 Architecture and components

ICING-PVM is a packet forwarding mechanism that allows a router to verify that a packet is following it pre-approved path before sending it to the next hop. An ICING-PVM network comprises ICING-PVM *nodes*, which enforce Path Compliance. ICING-PVM nodes are the machines that participate in ICING-PVM, possibly including end-hosts.

There are two main ways to deploy ICING-PVM: (1) In a layer 3 network (i.e., as a replacement for IP), and (2) in an overlay. When used as the forwarding mechanism for a layer 3 network, ICING-PVM nodes could be deployed by network transit providers just at the ingress boundaries of their networks; inside their networks, the providers need not implement ICING-PVM. In the overlay case, the ICING-PVM nodes are *waypoints* interconnected by the regular IP network. This section will be agnostic on the deployment scenario.

To communicate with a receiver, the sender first chooses a *path* through ICING-PVM nodes. How senders find paths depends on the scenario; a sender might query DNS to get a path (instead of an IP address), purchase access to a remote ISP via its Web site, statically configure paths, etc. Chapters 3 and 4 elaborate on some possibilities. This chapter assumes that the sender has candidate paths in hand.

Figure 2.1 summarizes how ICING-PVM forwards packets. For each node on the path, the sender requests from the node's provider a *Proof of Consent* (*PoC*). The PoC certifies the provider's consent to carry packets taking that path. The sender uses the PoCs to construct packet headers.

PoC creation is implemented by a *consent server* owned by the provider, or acting on its behalf. Consent servers are general-purpose servers separate from the network's forwarding nodes; this allows the policies to be flexible, fine-grained, and evolvable [83, 50, 52, 85, 136].

As a packet travels through the network, each ICING-PVM node verifies that the packet is following its approved path. This job decomposes into three tasks: (1) the node checks that the path is approved; (2) it checks that the path has been followed so far; and (3) it proves to downstream ICING-PVM nodes that it has seen the packet. Later, we describe in detail how ICING-PVM nodes perform these functions. The high-level construction is depicted in Figure 2.2. It relies on PoCs and on *Proofs of Provenance*, or *PoPs*. PoPs allow upstream nodes to prove to downstream nodes that they carried the packet. These proofs require pairwise *PoP keys*, but these keys do not require significant configuration state, as nodes derive the keys from their IDs.

2.2.2 Goals and non-goals

ICING-PVM seeks to provide a PVM's two properties, Path Consent and Path Compliance. We refine these properties into the following requirements for ICING-PVM:

- **Delegation**: A consent server must be able to delegate its path approval function.
- **Path Consent**: When a node receives a packet with path P, it must be able to verify that its consent server, or a delegate, approved P.
- **Path Compliance**: When a node N_i with index i in path P receives a packet with path P, the node must be able to verify that the packet was sent by the purported sender (index 0) and has been forwarded by each of the nodes at indices $1, 2, \ldots, i-1$, in that order.

ICING-PVM is designed to meet the above requirements while being amenable to an affordable high-speed hardware implementation and while not requiring a central authority, PKI, or significant configuration state. Our threat model, which is strongly adversarial, gives us further constraints. We describe our adversarial model in the next subsection.

There are several functions that ICING-PVM is *not* designed for:

The statement of Path Compliance does not guarantee a packet's future. After a packet departs a node, any downstream node can send it anywhere. In fact, ICING-PVM does not attempt to *prevent* such misbehavior; what ICING-PVM can do is *detect* it. Honest nodes do not accept a packet that has not followed its approved path. Thus, an upstream node can compare counts of accepted packets (for a given path) at itself and at a downstream node that it trusts. If the first count is larger than the second, there is a problem between the two nodes.

ICING-PVM *nodes can copy packets and send them elsewhere, or pass packets through hidden nodes.* ICING-PVM can only prove that nodes that *do* participate in ICING-PVM and implement the protocol correctly have seen the packet. It cannot prove that other nodes did not see the packet. Such a hidden node would be analogous to a transparent middlebox (e.g., a deep-packet inspection middlebox). However, ICING-PVM senders and receivers can *choose* their path—they can include only nodes that they trust not to leak their packets. This choice is complementary to encryption: encryption protects the content of the communication, and (as noted earlier) ICING-PVM gives endpoints the ability to keep discreet the *fact* of the communication.

ICING-PVM *does not attempt to provide authenticated information about the location of silent errors or failures on the path.* ICING-PVM provides a way for a node to signal an error back to the sender (Section 2.3.4); however, the sender cannot discover the location of a fault if none of the nodes on the path signals back to the sender.

ICING-PVM *does not provide information about whether a packet received any contracted-for services at a node.* For instance, a sender may choose to send a packet through a particular node because the node advertised a virus-scanning service. ICING-PVM does not provide a built-in mechanism that enables the sender (or receiver) to verify that a packet was indeed scanned. The receiver can verify only that the packet was forwarded through the node that advertised the service.

ICING-PVM *makes a binary decision about whether a path is acceptable; it does not regulate the* amount *of traffic sent along a path, or associated to a PoC.* Other work [175] has shown how to perform such accounting with minimal forwarder state, and ICING-PVM could be extended to incorporate this technique.

2.2.3 Threat model

Machines that obey the protocol we term *honest*. We assume that some providers, nodes (including end-hosts), and consent servers are not honest and specifically that they are controlled by attackers. These machines can engage in Byzantine behavior that deviates arbitrarily from ICING-PVM's specified packet handling. For instance, the attacker can send arbitrary packets or try to flood links to which it connects. The attacker can also observe legitimate data packets that pass through it. We make no assumptions about how malicious nodes are implemented: they may connect to one another and be controlled by a single attacker, or they may collude, potentially bracketing honest nodes on paths. Furthermore, even honest machines may give service to malicious parties; for instance, a consent server can grant PoCs to an attacker.

The attacker tries to make ICING-PVM fail some of its goals (Section 2.2.2), for instance by trying to abuse the delegation mechanism, or trying to make an honest node N_i accept a packet whose path was not approved by N_i's consent server (or a delegate of N_i), or whose actual path skipped some of the honest nodes upstream of N_i in the approved path.

We make security assumptions about several cryptographic primitives used by our implementation: that AES-128 [126] is a secure keyed pseudorandom function with full 128-bit security, that PMAC [44] is a fully secure deterministic MAC that is also a secure keyed pseudorandom function, and that CHI [89] is a secure hash function even if the hash is truncated to 248 bits.

2.2.4 Naming

Each ICING-PVM node assigns itself an identifier, called a *node ID*, which is a unique public key. The node keeps secret the corresponding private key. With such self-certifying names [117, 28], an entity does not need permission to create a name for itself, so a central naming authority or PKI is not needed. This fits the Internet's federated structure.

A path is a list of $\langle$node ID, tag$\rangle$ pairs. The *tag* identifies a specific set of local actions that a node performs on packets with this tag. For example, a tag can describe a priority level for queueing, identify a customer to bill, select one or more output links (unicast or multicast), request virus-scanning services, or specify a combination of these. It can be thought of as a generalized MPLS label [72] (and shares some functionality with the vnode mechanism in [81]). The provider conveys the particular meaning of a tag on a node to the users of that tag through some out-of-band means, such as an agreement with the user or a Web page.

2.2.5 Proofs of consent (PoCs)

After a sender has determined a path, it contacts the consent servers for each node on the path to obtain PoCs for that path (including the tags on that path). Each consent server is preconfigured with its provider's policy, so it can check that the path complies with that policy. To aid in its check, the consent server may incorporate external information (billing, authentication, etc.). If the check passes, the consent server creates a PoC based on the full path and returns it to the sender. Each PoC is associated with a specific node's tag, so it only permits the sender's traffic to transit that particular node using that particular tag. To ensure this association, the PoC includes a cryptographic token, specific to the path and computed under a *tag key* that is unique to the associated node and tag. This key is also known to the node.

Consent serving is flexible. A provider with multiple ICING-PVM nodes can deploy a single consent server. Or a provider can delegate the ability to create PoCs for a particular node and tag by divulging that tag's key. The recipient of the key can then mint PoCs that give a sender permission to send traffic through the given node and tag. Or a provider can disintermediate itself altogether by making all of its tag keys public.

2.2.6 Packet creation and proofs of provenance (PoPs)

As mentioned above, a sender obtains PoCs for the ICING-PVM nodes on its chosen path. It uses these PoCs to construct the packet header. The construction is such that when a packet arrives at node N, N can tell whether the sender held a PoC issued by N's consent server. The sender also computes PoPs for each of these nodes; a PoP proves to a node that the sender created the packet. A PoP includes a MAC of the packet under the shared symmetric PoP key.

These shared PoP keys do not require the network to be configured with pairwise keys. Instead, an ICING-PVM node (such as the sender) derives the PoP key that it shares with any other node from its own private key and from the other node's ID (which is a public key). The node uses a non-interactive Diffie-Hellman key exchange for the derivation and caches the results.

2.2.7 Packet processing: verification and forwarding

Each node that receives the packet does the following:

1. It computes the PoC from the path using the tag key that the node shares with its consent server.
2. For each upstream node in the path:
   (a) The node derives the PoP key it shares with this upstream node (which it can do from the upstream node's ID found in the path);
   (b) It computes the MAC of the packet (PoP) under the PoP key.
3. It checks that the PoC and PoPs are correct.

The PoPs computed in step 3-a prove to the node that the packet has passed through all the upstream nodes (including the sender). If the PoC is correct and the PoPs are all correct, then the packet has been following an approved path. Otherwise, the node drops the packet. If the checks pass, the node has to prove to downstream nodes that it has seen the packet. To do so, the node does the following:

4. For each downstream node in the path:
   (a) It derives the PoP key that it shares with this downstream node (again from the ID in the path).
   (b) It computes the PoP under this PoP key.
5. It inserts these PoPs into the header.
6. It forwards the packet to the next node.

As so far described, packet header size appears quadratic in the length of the path. However, the header size is in fact *linear* in path length: owing to the packet handling algorithm (Section 2.3.3), when the node receives a packet, the PoC and the PoPs that it inspects in steps 1–3 above are XORed together in an aggregate MAC; this approach reduces space while protecting the security of each of the components [98].

2.3 Design details of ICING-PVM

This section details ICING-PVM's design, which aims to meet the requirements stated in Section 2.2.2. Figure 2.3 describes the notation that we use throughout our design discussion and our pseudocode, while Figure 2.4 summarizes the secret cryptographic material used in ICING-PVM.

ICING-PVM's packet format is shown in Figure 2.5. Each packet includes three types of information for every node in its path other than the sender. The first is per-node information: a node ID, N_i, and a corresponding tag, tag_i (Section 2.2.4). The second is the Proof of Consent (PoC) showing that the consent server (or its delegate) authorized the path (Section 2.2.5). The third is the Proof of Provenance (PoP). The PoC—more precisely, an authenticator derived from it—and PoPs are aggregated into a constant-length verifier (V_i). The PoP allows each node to verify that every previous node has approved the path and forwarded the specific packet (Sections 2.2.6 and 2.2.7). PoCs and PoPs allow ICING-PVM to meet its requirements of Path Consent and Path Compliance. We discuss Delegation later.

Because packets carry node IDs, and because node IDs are public keys, our design needs small public keys to reduce the header size. Thus, we use elliptic curve cryptography (ECC): every node ID, N_i, is a point on NIST's B-163 binary-field elliptic curve group [25], which gives roughly 80-bit security, similar to 1024-bit RSA keys [25]. For the sake of readability, we denote the B-163 curve as an abstract cyclic group under multiplicative notation. We write g for the base point of B-163 (that is, the group generator), and q for the (prime) group order. The private key, x_i, corresponding to N_i is then the 163-bit number modulo q such that $N_i = g^{x_i}$. To make the approach amenable to a hardware implementation, we reduce the representation of N_i from 163 to 160 bits. We do so by requiring the top three bits of N_i to equal the first three bits of the SHA-1 hash of the lower 160 bits of N_i; heuristically, this does not diminish the strength of the keys (except for a brute force attack), though it increases expected key generation time by a factor of 8.

M	$\{$*vers*, *counter*, *proto*, *path-len*, *pkt-len*, *error-path-idx*, *payload*$\}$. A packet's static contents.
P	$\langle N_0\!:\!tag_0, N_1\!:\!tag_1, \ldots, N_n\!:\!tag_n\rangle$. A packet's path: a list of node identifiers and corresponding tags.
N_i	A node's identifier: a public key.
x_i	A node's private key: satisfies $N_i = g^{x_i}$.
tag_i	The tag corresponding to node N_i on path P: an opaque 32-bit string.
m_{N_i}	A node's *master tag key* : used in a key-derivation function (GET-TAG-KEY) to associate key material (*tag keys*) to any tag.
$m_{N_i:t/p}$	The *p-bit-prefix key* for tag t at node N_i: an intermediate key, created by GET-TAG-KEY, that enables calculation of tag keys at node N_i for any tag t' with the same p-bit prefix as t's. Note that $m_{N_i:t/0}$ equals m_{N_i} for any t.
$s_{N_i:tag_i}$	The *tag key* : used by a consent server for node N_i to create a PoC for a path that includes $N_i\!:\!tag_i$. Amounts to $m_{N_i:tag_i/32}$.
PoC_i	$(\mathrm{PoC}_i.expire, \mathrm{PoC}_i.proof)$. *Proof of Consent* to path P by node N_i.
$\mathrm{PoC}_i.expire$	A PoC's 64-bit expiration time indicator.
$\mathrm{PoC}_i.proof$	VPRF$(s_{N_i:tag_i}, P \,\|\mathrm{PoC}_i.expire)$.
A_i	PRF-96$(\mathrm{PoC}_i.proof, 0 \parallel$ HASH$(P \parallel M))$. A packet's *path authenticator* for node N_i.
$k_{i,j} (= k_{j,i})$	NIDH$(x_i, N_i, N_j)(=$ NIDH$(x_j, N_j, N_i))$. The *PoP key*: a symmetric key shared by nodes N_i and N_j, used for PoP computations. Soft state, derivable from nodes' identifiers.
$\mathrm{PoP}_{i,j}$	PRF-96$(k_{i,j}, i \parallel$ HASH$(P \parallel M))$. *Proof of Provenance* designated for N_j: A MAC by which N_i attests that it had approved a packet's path and handled it accordingly.
V	$\langle V_1, \ldots, V_n\rangle$. A packet's *verifier-vector.*
V_i	$(V_i.expire, V_i.proofs, V_i.hardener)$.
$V_i.expire$	Least significant 16 bits of $\mathrm{PoC}_i.expire$: used to check PoC expiration.
$V_i.proofs$	$A_i \oplus \mathrm{PoP}_{0,i} \oplus \ldots \oplus \mathrm{PoP}_{i-1,i}$. Aggregate MAC by which N_i checks out A_i (and hence PoC_i), as well as $\mathrm{PoP}_{j,i}$, for $j < i$.
$V_i.hardener$	PRF-32$(\mathrm{PoC}_i.proof, 0 \parallel$ HASH$(P \parallel M))$. Hardens forwarder slow path against DoS.
NIDH(x_i, N_i, N_j)	HASH2(SORT$(N_i, N_j) \parallel N_j^{x_i})$. (Hashed) Non-interactive DiffieHellman key exchange.
VPRF(s, d)	A keyed function that maps variable-length data d to 128-bit pseudorandom outputs. The current implementation uses PMAC [44].
PRF(k, d)	A keyed function that maps 256-bit data to 128-bit pseudorandom outputs (Appendix A).
PRF-96(k, d)	First 12 bytes of PRF(k, d). Suitable as a 128-bit message authentication code for d.
PRF-32(k, d)	Last 4 bytes of PRF(k, d).
HASH(d)	A collision-resistant hash function that maps variable-length data d to a 248-bit digest. Based on CHI [89]. Future versions of ICING-PVM will use the final SHA-3.
HASH2(d)	A collision-resistant hash function that maps variable-length data d to a 128-bit digest. Based on SHA-1 [125]. Future versions of ICING-PVM will use the final SHA-3.

Figure 2.3: Symbols and notation used in the pseudocode.

	node i ($i \geq 0$)	consent server i ($i > 0$)	delegate of node i	sender
x_i	x			
$k_{i,j}$	o			
m_{N_i}	x	x		
$m_{N_i:t/p}$	o	o	x	
$s_{N_i:tag_i}$	o	o	o	
PoC$_i$.*proof*	o	o	o	x

Figure 2.4: Cryptographic keys in ICING-PVM (rows), and various holders of these keys (columns). The key material is relative to the i-th entry in a packet's path (which is the sender, if $i = 0$). `x` denotes a key that the entity is given; `o` denotes a key that the entity can derive.

2.3.1 Generating PoCs and controlled delegation

Once a sender determines a path P (a process largely orthogonal to ICING-PVM, as discussed in Section 2.2.1), it must obtain PoCs for each $N_i{:}tag_i$ in P. To do so, it contacts N_i's consent server. The PoC consists of a 64-bit expiration timestamp, PoC$_i$.*expire*, and a cryptographic token, PoC$_i$.*proof* = VPRF($s_{N_i:tag_i}$, P ‖PoC$_i$.*expire*).

$s_{N_i:tag_i}$ is a tag-specific secret key that node N_i shares with its consent server. Because managing keys separately for their 2^{32} tags would be cumbersome for a node and its consent server, they instead share one master tag key, m_{N_i}, that pseudorandomly generates many tag keys. This process is encapsulated by GET-TAG-KEY. In more detail, let t/p denote the p-bit prefix of tag t, and define $m_{N_i:t/p}$ to be the corresponding p-bit *prefix key*. We take $m_{N_i:t/0} = m_{N_i}$ for any tag t. Then, GET-TAG-KEY computes $m_{N_i:t/p} = \text{PRF}(m_{N_i:t/(p-1)}, t/p)$. The tag key $s_{N_i:tag_i}$ associated to $N_i{:}tag_i$ is just $m_{N_i:tag_i/32}$. This approach is inspired by a technique in [136].

As so far described, this technique derives $s_{N_i:tag_i}$ from $m_{N_i:tag_i/0} = m_{N_i}$ using 32 serial rounds of PRF, which is too many for processing packets at high speeds. However, three modifications can reduce the number of rounds and their individual cost. First, a node can cache all prefix keys for y-bit prefixes in a *prefix key cache*. Second, a given node may choose to use only the top $32 - z$ bits of the tag field. Our hardware implementation, for example, takes $y = 16, z = 16$, making its number of

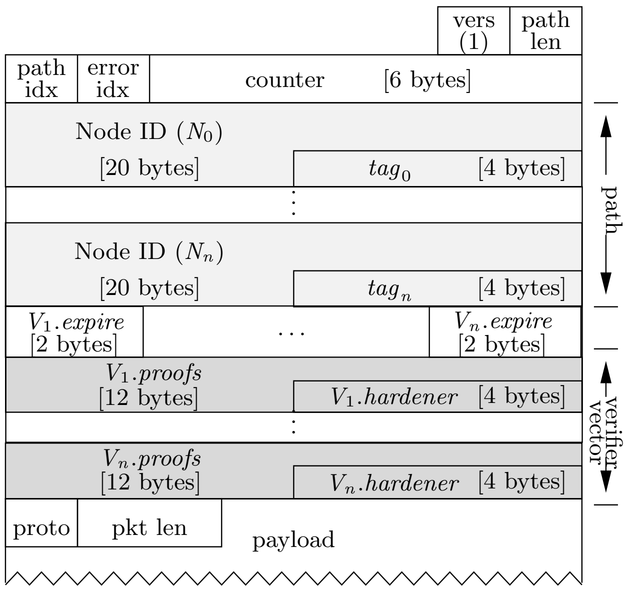

Figure 2.5: ICING-PVM header format.

required PRF invocations equal to $32 - 16 - 16 = 0$ (i.e., the node caches all of its tag keys). The third modification, which is detailed in Appendix A, constrains the admissible values of the prefix length, p.

Controlled delegation. Given the above approach to tag key derivation, *tag prefix delegation* is easy to implement. To delegate the tag block with prefix t/p (i.e., 2^{32-p} tags), the node's provider shares $m_{N_i:t/p}$. The delegate can further sub-delegate tags by sharing $m_{N_i:t/k}$, where $k \geq p$. Delegation lets a provider give customers control over particular tags on the provider's nodes. A customer with such control can, with no provider intervention, act as a consent server on behalf of the provider (creating PoCs for its own traffic if it is an end-host or for its customers if it is itself a provider) or give its customers their own tag keys (to disintermediate itself).

Expiration and revocation. The PoC.*expire* expiration timestamp allows consent servers to mint time-limited PoCs. This requires that consent servers and ICING-PVM nodes be loosely synchronized, via NTP [119] for example. The master tag key m_{N_i} and other prefix keys $m_{N_i:t/p}$ are changed periodically to guard against chosen-message cryptanalytic attacks and to prevent an old timestamp that has wrapped from appearing valid.

How does a provider revoke PoCs before the expiration interval? The provider

```
 1: function INITIALIZE($pkt, \text{PoC}_1, \ldots, \text{PoC}_n$)
 2:     $P = pkt.P$
 3:     $M = pkt.M$
 4:     $V = pkt.V$
 5:     $H = \textsc{hash}(P \parallel M)$
 6:     for $1 \le j \le n$ do
 7:         Set $V_j.expire = \text{PoC}_j.expire$ & 0xffff
 8:         $A_j = \textsc{prf}\text{-}96(\text{PoC}_j.proof, 0 \parallel H)$
 9:         Set $V_j.proofs = A_j$
10:         Set $V_j.hardener = \textsc{prf}\text{-}32(\text{PoC}_j.proof, 0 \parallel H)$
11:         $N_j = j$-th node in $P$
12:         $k_{0,j} = DHCache[N_j]$ or $\textsc{nidh}(x_0, N_0, N_j)$
13:         $\text{PoP}_{0,j} = \textsc{prf}\text{-}96(k_{0,j}, 0 \parallel H)$
14:         Set $V_j.proofs = V_j.proofs \oplus \text{PoP}_{0,j}$
```

Figure 2.6: Pseudocode for packet initialization. The sender initializes the verifiers before sending the packet to the first node.

can change m_{N_i} at the node and consent server. Another option is to change only a prefix key (or tag key), no longer deriving it from m_{N_i}. To do so for the prefix t/p, a node inserts an entry $(t/p, m'_{N_i:t/p})$ into a small override table. Unlike the prefix key cache, this table uses longest prefix matching (so is costlier); however, it is needed only for prefix lengths p larger than y, the prefix key cache index size.

2.3.2 Creating a packet

Before sending a packet, the sender calls INITIALIZE (Figure 2.6), which creates a verifier V_j for each other node j on the path. The sender initializes V_j with A_j (which binds the PoC_j to the packet contents) and $\text{PoP}_{0,j}$; given this V_j, a downstream node can verify that the packet's path is approved by its consent server and that the packet has been created by the packet's purported sender.

```
function RECEIVE($pkt$)
    $P, M, V = pkt.P, pkt.M, pkt.V$
    $i = pkt.path\text{-}idx$
    $N_i : tag_i = i$-th entry in $P$
    $T =$ current time
    $\mathrm{PoC}'_i.expire = (T \,\&\, \tilde{}\,\mathrm{0xffff}) \mid V_i.expire$
    if $\mathrm{PoC}'_i.expire < T$ or $N_i \neq$ my node ID then
        Drop pkt
    $s_{N_i:tag_i} = \textsc{get-tag-key}(m_{N_i}, tag_i)$
    $\mathrm{PoC}'_i.proof = \textsc{vprf}(s_{N_i:tag_i}, P \parallel \mathrm{PoC}'_i.expire)$
    $H = \textsc{hash}(P \parallel M)$
    $A'_i = \textsc{prf}\text{-}96(\mathrm{PoC}'_i.proof, 0 \parallel H)$
    $V'_i.proofs = A'_i$
    $V'_i.hardener = \textsc{prf}\text{-}32(\mathrm{PoC}'_i.proof, 0 \parallel H)$
    if $V'_i.hardener \neq V_i.hardener$ then
        Drop pkt
    // verify upstream PoPs (check Path Compliance)
    for $0 \leq j < i$ do
        $N_j = j$-th node in $P$
        $k_{j,i} = DHCache[N_j]$ or $\textsc{nidh}(x_i, N_j, N_i)$
        $\mathrm{PoP}_{j,i} = \textsc{prf}\text{-}96(k_{j,i}, j \parallel H)$
        $V'_i.proofs = V'_i.proofs \oplus \mathrm{PoP}_{j,i}$
    if $V'_i.proofs \neq V_i.proofs$ then
        Drop pkt
    // create downstream PoPs (prove Path Compliance)
    for $i \leq j \leq n$ do
        $N_j = j$-th node in $P$
        $k_{i,j} = DHCache[N_j]$ or $\textsc{nidh}(x_i, N_i, N_j)$
        $\mathrm{PoP}_{i,j} = \textsc{prf}\text{-}96(k_{i,j}, i \parallel H)$
        Set $V_j.proofs = V_j.proofs \oplus \mathrm{PoP}_{i,j}$
    Set $pkt.path\text{-}idx = i + 1$
    Add all calculated $k_{x,y}$ to $DHCache$
    Perform any special handling prescribed by $tag_i$
    Transmit $pkt$ to next node (or accept if destination)
```

Figure 2.7: Pseudocode for packet forwarding. The node validates the packet and transforms verifier entries before honoring the tag specified in the packet's header and sending the packet to the next node. Note that P is 0-indexed and V is 1-indexed.

2.3.3 Forwarding and receiving a packet

On receiving a packet, a node N_i with index i in the packet's path processes it according to the pseudocode in Figure 2.7. The node must ensure that ICING-PVM's requirements of Path Consent and Path Compliance (Section 2.2.2) are met for *each* packet that it passes. (Delegation was discussed in Section 2.3.1.) First, for Path Consent, N_i must verify that the PoC implicit in the packet, PoC_i, is correct. Second, for Path Compliance, the node must verify the PoPs created by upstream nodes $N_0, \ldots, N_{i-1}$ (that is, it must verify $\text{PoP}_{j,i}$ for $j < i$). The node executes both checks by validating the verifier V_i. To do so, it derives an expected verifier V_i', which requires deriving the expected PoC_i' (based on the path and the relevant tag key, *cf.* Figure 2.7, lines 9–10), the expected A_i', and the expected PoPs. If V_i' does not match the verifier in the packet (V_i), the packet is dropped (Figure 2.7, lines 11–24). To perform its required duty with respect to Path Compliance, the node modifies the verifiers for downstream nodes (V_j for $j \geq i$) by XORing V_j with $\text{PoP}_{i,j}$ (Figure 2.7, lines 26–30).

Computing PoPs for a packet may require deriving PoP keys $k_{i,j}$ $(= k_{j,i})$. Because it is resource-intensive, this calculation happens on a node's separate *slow path*, so it does not delay the processing of other packets that do not require the same calculation. An attacker may try to attack a node's slow path by sending many packets with invented node IDs. To defend against such an attack, the node requires a valid *hardener* (V_i*.hardener*) in the packet, which it checks on the *fast* path (Figure 2.7, lines 14–16). V_i*.hardener* is only 32 bits, so it does not fully rule out such attacks, but it decreases their effectiveness by a factor of 2^{32}, which is sufficient to avoid denial-of-service.

2.3.4 Signaling errors and failures

Because ICING-PVM packets are source-routed, a network using ICING-PVM needs to report errors and other failures back to the sender so that the sender can use a different path if necessary. Note that a sender can hold pre-approved backup paths, so failures need not require the sender to obtain new paths.[2]

But the sender needs to learn of the error or failure. For that, ICING-PVM error packets need to travel along the reverse of a given path toward the sender. The slight complication is Path Consent: for some nodes, consenting to a path's forward direction may not imply consent to carry error packets along the reverse of the path, in which case the sender has to rely on end-to-end failure detection.

For nodes that do consent to carry error packets, ICING-PVM handles errors as follows. To create an error packet, a node sets the *error index* field in the header to the current index and replaces the payload with the original packet's hash, followed by optional error-specific information (analogous to ICMP error code and data). A node recognizes packets with non-zero error index fields as error packets and handles them differently: most importantly, the node forwards such packets to the previous node (rather than the next) and decrements (rather than increments) the path index field. A node that experiences a failure when sending an error packet drops it—error packets never generate further error packets.

The V_i in an error packet contain all the forward-direction PoPs from the original packet that caused the error, in addition to PoPs for the error packet itself. Because the error payload begins with the original packet's hash, nodes can verify these forward-direction PoPs despite not having the original packet. In particular, node i drops an error packet unless $V_i.\mathit{proofs}$ includes a PoP under $k_{i,i}$. This ensures that a node will not forward an error packet if it did not previously forward the original.

2.3.5 Attacks

Attacks against the verification algorithm. The algorithm guards against the following attacks.

- Using incorrect or expired PoCs: This attack fails because each node checks the expiry and recalculates its expected PoC (Figure 2.7, lines 7–10).

[2]Even if the sender needs to obtain new paths, it is not necessarily a disaster. In the status quo, packets do not magically skirt around failures; instead, the routing protocol must run, which also takes time.

- Skipping an honest node i: When the packet is received at honest node j downstream of node i, V_j will lack $\text{PoP}_{i,j}$ and will be flagged (Figure 2.7, lines 11–24).
- Flooding a node's slow path: The attack is mitigated because the node checks $V_i.\textit{hardener}$ before calculating any PoP keys (Figure 2.7, lines 14–16).

Attacks on the cryptographic primitives. We expect the class of primitives in ICING-PVM (PRFs, collision-resistant hash functions, etc.) not to change. However, the particular algorithms in our implementation (PMAC, AES, CHI, etc.) may be broken in the future or require longer key lengths. We anticipate that these changes will happen far more slowly than the rate at which hardware becomes obsolete or network bandwidth increases. As the hardware changes, new versions of the protocol can be rolled out, with the hardware supporting old and new versions as a way to smooth the changeover (as with IPv4 to IPv6).

Attacks that compromise secrets. How should a node handle the inevitable compromises of its cryptographic material (Figure 2.4)? We have discussed PoC revocation (Section 2.3.1): a node changes prefix or tag keys ($m_{N_i:t/p}$, for various t/p). If an $m_{N_i:t/p}$ itself is compromised, a node can simply change it. A more serious concern is the compromise of a private key, x_i, or any derived PoP key, $k_{i,j}$. In that case, the node must generate a fresh public/private key pair so must also change its node ID. This requirement is inconvenient but not disastrous; it is tantamount to advertising a new business name, or to moving, in that the other entities that express policy in terms of the renamed node must be notified about the change. Allowing a node to change a $k_{i,j}$ while retaining its private key is future work.

Attacks that attempt packet replay. 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 layer using ICING-PVM presumably handles duplicates anyway. Meanwhile, an attack that aggressively floods using a few packets can be defeated by a modestly sized replay cache at each node; this cache would store $\langle \text{PoC}, \text{counter} \rangle$ pairs (the counter, from the packet header field, is chosen by the sender to be unique over the flow). A difficult case is if the attacker can amass packets from many flows within a single PoC validity window and then replay each packet a small number of times. Defending against this case is future work; it may require both reducing the

PoC validity window and compressing the information in the replay cache.

Attacks on availability. What if an attacker overwhelms a consent server [34, 175]? One option is to locate the consent server at a high-bandwidth denial-of-service mitigator (e.g., [135]). Another option, if the receiver already knows who should be allowed to reach it (for example, employees or customers), is to give these senders their own tag keys $s_{N_i:tag_i}$ so that they can mint their own PoCs without the consent server. Third, if PoC requests travel in ICING-PVM packets, then ICING-PVM's mechanisms themselves provide a foundation for defense. These mechanisms apply not just to an overloaded consent server but also to any receiver wishing not to hear from a sender. For instance, if senders can be identified at a useful granularity (e.g., "employees", "paying customers", "unknown senders who solved a CAPTCHA"), then the victim can assign each category to a different tag. When overloaded, the victim deprioritizes categories by not renewing expired PoCs for their tags; downgrading service to them; or, in an emergency, changing tag keys. If senders cannot be assigned to categories, we can follow TVA [175], ensuring roughly fair bandwidth consumption among senders by applying Hierarchical Fair Queueing to a packet's path. While attackers can weaken this defense under TVA by faking path identifiers, ICING-PVM does not have this vulnerability.

Some attacks that ICING-PVM does not defend against are mentioned in Section 2.2.2.

2.4 Implementation

This section describes our implementation of the ICING-PVM node's hardware and software. Our prototype node accepts ICING-PVM packets carried in Ethernet frames and implements the algorithm in Figure 2.7.

The implementation is divided into two paths: a fast path that runs in hardware, and a slow path that is executed in software if a PoP key ($k_{i,j}$) is not cached in hardware or if an exception occurs. The fast path is implemented on the NetFPGA[3]

[3]We used NetFPGA-1G as opposed to the NetFPGA-10G, which at the time of writing was not yet available.

Average increase in packet overhead: 23.3%	§2.5.1
Throughput: 82.5-100% of IP on NetFPGA	§2.5.2
Normalized hardware cost: 186% of IP on NetGPA	§2.5.4

Figure 2.8: Summary of main evaluation results.

programmable hardware platform [10], which is a PCI card with 4 GigE ports, a field programmable gate array (FPGA), SRAM, and DRAM. The slow path, implemented in Click [102], calculates the needed keys and installs them in the hardware's key cache. The Diffie-Hellman key exchange is implemented with the MIRACL cryptographic library [145]. All of the node's software runs on Linux 2.6.25.

We have not yet implemented PoC expiry, or the handling of error packets. However, we do not expect these features to change our evaluation, as reported in the next section.

The hardware image uses support modules from the NetFPGA project. We implemented the ICING-PVM-specific logic, including cryptographic modules. The forwarder uses 89% of the total FPGA logic area and has a total equivalent gate count (EGC) of 13.4M. (EGC roughly estimates how many gates a design would use on an ASIC, as reported by the Xilinx ISE synthesis tool, ver 10.1.) The area breaks down as follows: 38% to the AES, CHI, and PMAC modules, 28% to all other ICING-PVM-specific logic, and 34% to the NetFPGA support modules.

By comparison, NetFPGA's reference IP router has an equivalent gate count of 8.7M and uses 50% of the total FPGA logic area.

2.5 Evaluation

ICING-PVM 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 speeds that match Internet backbone links. In this section, we assume that ICING-PVM is deployed at the network layer as in Chapter 4. We begin by estimating ICING-PVM's total space overhead (Section 2.5.1). Sections 2.5.2 and 2.5.3

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

Figure 2.9: Machines for measuring ICING-PVM overhead.

		Fixed parameters		
Varied Parameters	Range	Path len	Path idx	Pkt size
Packet size	{311, 567, 823, 1335, 1514}	—	7	3
Path length	{3, 7, 10, 20, 30, 35}	1514	—	1
Path index	{1, 5, 10, 15, 18}	831	20	—

Figure 2.10: Parameters used throughout experiments. Packet size includes header.

present microbenchmarks of our prototype node and supporting software. In Section 2.5.4, we extrapolate from our results to assess ICING-PVM's future feasibility as part of the Internet core. Our results are summarized in Figure 2.8.

Setup and parameters Figure 2.9 lists the 3 machine classes that we use for evaluation. The NetFPGA is a PCI card inside the *slow* machine. Our experiments vary packets' path lengths, path indices, and sizes. Figure 2.10 gives the fixed and variable parameters for the measurements of forwarding throughput and software performance.

2.5.1 Packet overhead

Relative to IP, ICING-PVM requires larger packet headers so would consume more bandwidth. We now roughly quantify this overhead. An ICING-PVM header includes 13 bytes that do not depend on the packet's path length (see Figure 2.5). 42 bytes are needed for each node in the path other than the sender: 24 bytes for the node ID and tag, $N_j\!:\!tag_j$, and 18 bytes for the verifier V_j. For a packet whose path length is 5—a pessimistic estimate of the average provider-level path length from [100] and [28]—the header is 205 bytes, or 13.5% of a 1,514-byte packet.

To estimate the total increase in bandwidth consumed by ICING-PVM's headers,

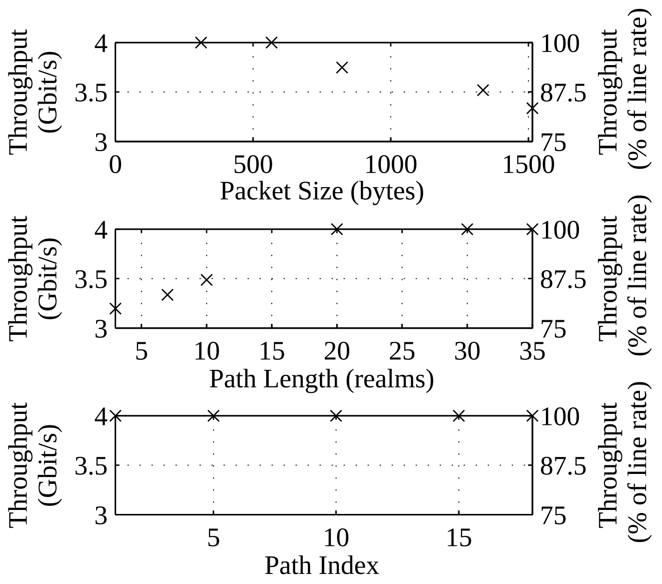

Figure 2.11: Average throughput as a function of packet size (Figure 2.10, row 1), path length (Figure 2.10, row 2), and path index (Figure 2.10, row 3). Percentages are relative to maximum possible throughput on the NetFPGA. Standard deviation is less than 0.02% of the mean at each measurement point. The forwarder's throughput is lowest for packets with large payloads but small path lengths: such packets send the largest number of bits through the hash function, which is the bottleneck.

we look at a sample trace from CAIDA [14]. The total number of packets observed for about 15 minutes was 37,571,701 with a total size of 28,474.70 MiB. For each packet, ICING-PVM's increase in overhead relative to an IP header (of 20 bytes) is $205 - 20 = 185$ bytes (assuming path lengths are uniformly distributed across all packet sizes). So the total increase in bandwidth consumption for this particular dataset would be $37,571,701 \times 185/(28,474.70 \times 2^{20}) = 23.3\%$ relative to IP.

2.5.2 ICING-PVM hardware

We now measure the performance of the (fast path) hardware in our prototype ICING-PVM node, described in Section 2.4.

From Figure 2.7, one might expect the cost of processing a packet to depend on the path length because the work of verifying Path Compliance and proving it

Action	Processing time	Throughput (1/Proc. time)
Calculate $k_{i,j}$	4 ms ($\sigma = .043$ ms)	250 keys/s
Generate PoC	$0.4x + 1.3$ μs	$2.6 \cdot 10^6/(x + 3.5)$ PoC/s
Create packet (w/c)	$2.6x + 40.1$ μs	$3.9 \cdot 10^5/(x + 15.4)$ pkt/s
Verify packet (w/c)	$2.6x + 24.4$ μs	$3.9 \cdot 10^5/(x + 9.5)$ pkt/s
Create packet (n/c)	$33796.1x - 32758.4$ μs	$29.6/(x - 0.9)$ pkt/s
Verify packet (n/c)	$34875.1x - 33647.1$ μs	$28.6/(x - 0.9)$ pkt/s

Figure 2.12: Processing time and throughput for software operations. x is the path length. Packet creation and verification costs are measured both with and without the use of cached shared keys (w/c and n/c resp.). For the last four rows, processing time is derived by linear regression, and $R^2 > 0.99$ in all three cases.

seems proportional to the path length. However, the results of the various PRF-96 operations are XORed, so they can be parallelized in a pipeline and thus removed from the critical path. The only other heavily serialized function in the design is the hash function (HASH), so we expect it to be the bottleneck; i.e., throughput should depend on the number of bits that must be hashed. Since the only fields that are *not* hashed are the path index and the verifiers' V_js, we expect throughput to be lower when the V_js represent a smaller fraction of the total packet bits. In other words, for a constant path length, we expect throughput to decrease as packet size increases.

We measure our prototype's fast path throughput by connecting the four ports of an ICING-PVM node to a NetFPGA packet generator [62] that sends ICING-PVM packets at 4 Gbit/s. We measure throughput over 5 10-second samples, using the measurement points in Figure 2.10. The ICING-PVM node loops ingress packets back to the packet generator, which measures the average bit rate.

Figure 2.11 plots the measured throughput. (Note that we do not report goodput; instead we report ICING-PVM packet header overhead in Section 2.5.1.) The minimum aggregate throughput is 3.3 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.

2.5.3 ICING-PVM software

We now measure the performance of the (slow path) software in our prototype ICING-PVM node. We also measure end-host ICING-PVM operations. Figure 2.12 summarizes.

Shared key ($k_{i,j}$) derivation. A packet invokes our prototype's slow path when the hardware does not have the required shared keys cached (Figure 2.7). We measure the cost of deriving $k_{i,j}$ by running 3,000 iterations of the calculation function in a tight loop on the *slow* machine. On average, a single calculation takes 4 ms.

End-host. An end-host must also perform cryptographic operations: senders initialize all the verifier entries, and receivers validate and modify some of these entries. 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 Figure 2.10, take packet size to be 1,514 bytes, and collect 1,000 samples per path length on the *medium* machine. We record total processing cost (of either packet generation or verification, depending on sender or receiver; in both cases, we record the cost when the $k_{i,j}$ keys are and are not cached), and then use ordinary least squares linear regression. The inferred coefficients ($R^2 > 0.99$) are in Figure 2.12. Each entry in the path increases packet creation and verification times by 2.6 μs. For an average path length of 5, packet verification can be performed at 23K pkts/s.

Packet generation takes longer than verification because senders are so far unoptimized and compute HASH$(P \parallel M)$ twice. Were the endpoints optimized, receiving would likely be more expensive than sending: the receiver also hashes the packet (to verify V) and has an additional cost, namely re-computing the local PoC.

2.5.4 Scaling

We now give a rough assessment of whether an ICING-PVM node could meet the demands of the Internet backbone.

Throughput and cost. In assessing whether ICING-PVM could scale to backbone speeds, our metric is *normalized cost*: it measures the hardware cost, reported as equivalent gate count, per unit of throughput. As a baseline, we consider a simple IP router on the NetFPGA. We obtain gate counts for the ICING-PVM and IP

	NetFPGA ICING-PVM	NetFPGA IP
Min Throughput (Gbits/s)	3.3 (from §2.5.2)	4
(Eq.) Gate Count (Gates)	13.4M	8.7M
Normalized Cost (Gates/(Gbits/s))	4.1M	2.2M

Figure 2.13: Normalized costs of the NetFPGA ICING-PVM forwarder and the NetFPGA IP reference router. The equivalent gate count is an estimate of the cost of the implementation on an ASIC given by the Xilinx synthesis tool ISE ver 10.1.

implementations from the synthesis produced by Xilinx's ISE software (ver 10.1).

Figure 2.13 summarizes the comparison. Using our normalized cost metric, our ICING-PVM forwarder is ∼86% more expensive than the NetFPGA IP router. However, the IP router is a pessimistic baseline because it is bare bones: it has only a 32-entry TCAM for longest-prefix matching (commercial routers have on the order of hundreds of thousands [59], and the TCAM is a big consumer of logic area), and it does not have the functionality of commercial-grade routers (packet filtering, tunneling, etc.). On the other hand, almost all of ICING-PVM's processing can be parallelized, so it seems that there is no fundamental obstacle to scaling ICING-PVM to backbone speeds (around 100 Gbits/s).

Next, we give a very rough estimate of the die size required for ICING-PVM processing at 100 Gbits/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 [105] 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 [105], 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 would reduce 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 [105] 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-PVM 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 3.3 Gbit/s, from Figure 2.11). 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 very little area at the 40 nm technology.

Symmetric key cache. An ICING-PVM node N_i stores a table of $(N_j,\ k_{i,j})$ pairs. Would this cache be too expensive? There are fewer than 40k advertised AS numbers, and the total is growing at less than 3.2k/year [1]. However, the total number of peering points these ASes have is unclear, especially given the fact that many of these ASes peer together at Internet Exchange Points (IXPs). If we assume that each AS owns on average 10 nodes, the key cache would need to be approximately 400k keys. The size of the lookup entries is increased from 32-bit IP prefixes to 160-bit flat node IDs, while the size of the returned data (the shared symmetric key) is increased to 128 bits. The needed table size to fit all ICING-PVM nodes would be almost 14 megabytes, which is within today's SRAM capabilities [23]. In any case, we believe that an optimization many providers will do is use the same node ID for many (if not all) their nodes to ease key management and policy configuration. For further analysis of a nearly identical question, see [28, §4].

Tag key cache. An ICING-PVM node N_i also caches precalculated tag keys $s_{N_i:t}$ or prefix keys $m_{N_i:t/p}$. While an extended analysis is outside of our scope, we just note that viable SRAMs [23] already exist that could fit 2^{20} keys, sufficient for more than 1 million tags.

2.5.5 Overhead of PoC generation

To measure the cost of generating PoCs in software, we run the calculation function in a tight loop, varying path length per Figure 2.10. To represent the consent servers, we use the *fast* machine. Our results (Figure 2.12), show that cost is proportional to path length, as expected from the definition of PoC.*proof* (Section 2.3.1). For a path length of 7, the consent server can generate approximately 248,000 PoCs/s, well within the range of request rates handled by a root DNS server [139].

2.6 Applications of ICING-PVM

ICING-PVM enables a receiver to request services for incoming packets (e.g., outsourced intrusion detection service (IDS) or denial-of-service mitigator [135]) and then verify that received packets actually traversed the services. ICING-PVM also enables these services themselves to specify other intermediate services (e.g., the IDS can specify an accounting service that drops traffic for non-customers). Unlike previous work [149, 159], ICING-PVM provides integrity and authentication in the forwarding mechanism, even for intermediaries, obviating the need for reimplementing it for each service or application.

Another application of ICING-PVM is enforcing routing policy. Today, Internet providers run a policy routing protocol, BGP, to exchange information according to their business policies. Yet, forwarding in the current Internet undermines policy routing: packets can (and frequently do [114]) deviate from the paths specified by the routing protocol [73]. The simplest example of such deviation is providers who use "hot-potato routing", or who ignore BGP's multi-exist discriminators (MEDs). Under ICING-PVM, in contrast, providers can verify whether or not packets are following agreed-upon paths.

A third application is policy-compliant source routing. Under source routing [172, 81, 132, 99, 87, 174, 38, 66, 152], the sender selects paths. For instance, a user could invoke a desktop application to specify which providers carry her traffic, letting her avoid an ISP that she believes is throttling her traffic [172], or choose less congested paths [143, 29]. Or an enterprise could direct packets between its branch offices along vetted paths. Unfortunately, unconstrained source routing is economically unviable: it comes at the expense of providers, by overriding their policies about paths (a point made in [136]). ICING-PVM, however, resolves this tension: the sender gets choice, but the choices are constrained by the policies of providers, or their delegates.

2.7 Discussion

The reason that a PVM can enable the applications above is that it can be viewed as a generalization of the forwarding mechanisms of many network architecture proposals (a point made elsewhere [146]). If we look at several decades of network architecture proposals [136, 28, 108, 48, 67, 100, 93, 151, 132, 131, 38, 120, 32, 113, 172, 87, 174, 99, 81, 86, 159, 149, 77, 68, 47, 154, 152], we find that in each case, participants involved in sending, forwarding, or receiving a packet establish policies based on the other participants (as one of many possible examples, in capability or default-off architectures [64, 168, 175, 40, 109], the receiver elects not to hear from particular senders). To unify such proposals, the network can allow all entities on the path to express policy about the entire path, which is precisely Path Consent. But Path Consent—which is about policy *expression*—is not enough. In an adversarial environment, policy *enforcement* matters. Solving this problem for a general data plane is tantamount to meeting a PVM's second property: Path Compliance.

The generality of a PVM such as ICING-PVM certainly has a price, as it is more expensive than many of the individual mechanisms that we have surveyed. However, we are solving a different problem from these other mechanisms: our goal is to provide a mechanism that is capable of enforcing a wide variety of policies.

Moreover, it is possible to imagine cheaper PVMs than ICING-PVM, if we relax our requirements. For example, under a central authority (a reasonable assumption: the current Internet has IANA), a PVM could use identity-based encryption ([45]), leading to smaller public keys (e.g., 8 bytes instead of 20 bytes) and hence smaller packet headers. Or the central authority can distribute a trusted map from short identifiers to public keys, so packets need not carry public keys at all. Or a PVM could check a fraction of the packets at a fraction of the cost. And if we allow per-flow state in nodes, then packets need not carry the full path, only a token that corresponds to it [175, 53, 52].

Besides ICING-PVM's costs, we must also consider its use complexity. In an overlay, the interface to ICING-PVM is relatively simple (though obviously more involved than IP). At the network layer, there is some complexity in bootstrapping; the details are unsurprising and tedious.

Does ICING-PVM restrict communication, as it empowers each node to enforce policy unilaterally? We note, first, that regardless of ICING-PVM, *any* carrier of a communication can exercise control over it. They are free to drop packets, deprioritize them, or corrupt them. Second, providers in the current Internet routinely sever transit between each other to gain leverage in contract negotiations, effectively partitioning the Internet at users' expense [179, 156]. Under ICING-PVM, at least, end-points get multiple options about paths and intermediate providers, which could create competition where today monopoly reigns.

Last, the Internet seems to work sufficiently well under a threat model far more relaxed than ours, so what types of networks is ICING-PVM best suited for? One example is when packet delivery is expensive, such as with satellite links. Another is in military-grade networks, when it is important to restrict covert channels like leaks of which senders and receivers are communicating. ICING-PVM may also protect future networks, like the smart grid envisioned for power delivery, generation, and markets.

But this is looking far ahead. Looking back, our motivating question was whether it was possible to design a feasible PVM and, if so, what it would cost. This chapter has attempted to answer that question.

Chapter 3

ICING-ON: A Policy-Compliant Overlay Network

3.1 Introduction

The last chapter described packet forwarding under ICING-PVM. But a packet forwarding mechanism is not sufficient to build a network. The Internet is not just IP forwarding. It has a host of other protocols and components that cooperate to achieve end-to-end connectivity. With regards to ICING-PVM, we need to answer at least two more questions: (1) What is the interface to ICING-PVM seen by senders, receivers, and providers? (2) How do the functions of path retrieval, topology discovery, and bootstrapping work? In this chapter, we answer these questions in the context of an overlay network: ICING-ON. ICING-ON uses ICING-PVM as its forwarding mechanism, inheriting ICING-PVM's properties.

Another deployment scenario for ICING-PVM is as a network-layer protocol. As mentioned in Section 2.2.1, this scenario requires details that we leave for Chapter 4, namely how, using the default-off network itself, senders get consent to request consent. Under ICING-ON, in contrast, nodes use IP for this purpose, thus avoiding this bootstrapping hurdle. Like RBF [134] and Platypus [136], ICING-ON can be deployed in the network as an extension to existing routers to completely lockdown access.

Before we continue, let's review some of the applications that such an overlay can enable. We have already mentioned a number of applications in Section 2.6. The two most relevant ones to an overlay are *composable network services* and *policy-compliant source routing*.

In the first application, composable network services, a receiver specifies a number of network-level services through which packets destined to it must pass. The important addition that ICING-PVM provides is verifiability: the receiver can check that packets have indeed gone through the services it has requested before accepting them. For instance, the receiver can specify an outsourced intrusion detection service (IDS) and an auditing service that logs all incoming messages. ICING-PVM also enables these services to specify other intermediate services themselves. So the IDS can depend on sub-services that perform various parts of the intrusion detection function such as accounting, virus-scanning, DoS inpection, etc. Other work has already attempted to implement similar functionality in an overlay (by extending the systems in [149, 159], for example), but because ICING-PVM provides authentication and integrity at the network layer at high-speeds, it obviates the need for reimplementing them for each service or application.

In the second application, policy-compliant source routing, a source specifies the path that packets will take to the destination. In an overlay, the reasons vary from performance (letting sources find the best quality paths [143, 29]) to preference (letting sources override default routing in the underlying network layer to avoid particular areas). Like Platypus [136], ICING-PVM allows the provider of such overlay waypoints to identify which entities to bill for traffic. But ICING-PVM additionally gives the provider control over the overlay path that packets are allowed to take (for example, a provider can specify that routing through its node is only allowed if the packet passes through another peering provider). And while in both, Platypus and ICING-PVM, an entity can delegate purchased access through a particular node to another entity, only ICING-PVM allows the delegator to control the path the delegate uses through that node.

We note that these applications are not mutually exclusive. ICING-PVM's Path Consent allows both applications to coexist on the same fabric. This fabric is provided as an overlay by ICING-ON.

The rest of this chapter will describe ICING-ON's design and its routing protocol, SPB (Section 3.2), work through an example of how these components function (Section 3.3), and discuss design choices and trade-offs (Section 3.4).

3.2 Design

ICING-ON's design does not specify how senders make path choices or how overlay node providers choose their path policies. It does, however, specify how senders discover available path choices. ICING-ON has several components: Overlay nodes and consent servers, ICING-ON gateways, and domain name space (DNS) entries for ICING-ON nodes and tags.

3.2.1 Overlay nodes and consent servers

To express and enforce their policies, overlay providers deploy consent servers and ICING-ON nodes that are connected by IP. Overlay nodes implement ICING-PVM above UDP. A tag specifies the internal disposition or service that an overlay node applies to a packet carrying that tag. For example, the tag can specify virus-scanning on behalf of a particular customer.

Consent servers approve paths on behalf of the providers of the nodes and help senders find approved paths using a protocol we call *Simple Path Building* or *SPB*: when a sender requests consent for a path that the consent server's policy does not permit, it returns to the sender a message informing the sender of the rejection, and it describes the patterns of paths that the consent server will approve based on the originally requested path. We describe how senders find consent servers using DNS in Section 3.2.3.

For example, say the consent server for node Y will only approve paths that pass through node X on tag x (i.e. $X\!:\!x$) before reaching Y. The administrator of the consent server would configure it with the following pattern for approved paths: $\langle *, X\!:\!x, *, Y, * \rangle$ where $*$ means any sequence of *nodeID*:*tag* pairs. Say a sender W wishes to send traffic to a destination Z through node Y. The sender will contact Y's consent server, requesting consent for path $\langle W\!:\!w, Y\!:\!y, Z\!:\!z\rangle$. Instead of returning a PoC for the path, the consent server returns a pattern $\langle W\!:\!w, *, X\!:\!x, *, Y\!:\!y, *, Z\!:\!z\rangle$. The sender uses this pattern to construct a new path, $\langle W\!:\!w, X\!:\!x, Y\!:\!y, Z\!:\!z\rangle$. It uses this new path to request consent from the new node's consent server, X. If X does not approve the path, this process repeats: X returns one or more new, more specific patterns, of which the sender chooses one. The sender then requests consent for the additional required nodes in that chosen pattern. Usually, the sender will eventually build a pattern that represents the combined path policy of all nodes on the path. At that point, it proceeds to request PoCs again, this time using a path that all nodes would agree to.

Sometimes, this process will not converge. This happens when at least two nodes (one of which maybe the sender) on the path have conflicting policies. The sender at that point needs to make different choices during path retrieval or purchase access to some nodes.

This process seems like it would significantly increase connection setup latency. We note three optimizations. First, a sender can cache built paths so they can be reused without having to go through the path and consent retrieval process again. Second, senders can proactively build paths based on their policies and maintain up-to-date PoCs for them. And third, because ICING-PVM enables delegation of PoC creation, we expect one consent server to approve paths for several nodes (for example, if the consent server's owner is a customer of these other nodes' providers) as we expect a sender to hold the PoC keys required to mint PoCs for some of the nodes it wishes to use in its paths (because it might have purchased access to them).

3.2.2 ICING-ON gateways

ICING-ON gateways can either be transparent middleboxes or software installed on end-hosts. They serve as the interface to the overlay network, acting as ICING-PVM senders and receivers. For egress traffic from an end-host or a network, a gateway receives standard IP packets and, acting as an ICING-PVM sender, encapsulates them in ICING-PVM packets. The gateway then transmits the packets through the overlay, which now carries IP-in-ICING-PVM-in-IP packets. For ingress traffic into an end-host or network, the gateway acts as an ICING-PVM receiver, processing the ICING-PVM header in the IP-in-ICING-PVM-in-IP sandwich, and then deencapsulating the innermost IP packet for its final delivery. These gateways build and cache paths and PoCs to minimize connection setup latency. Gateways usually look at packet information (e.g., IP protocol field or TCP/UDP port numbers) to choose paths. Sophisticated gateways may proactively build several paths to frequently contacted destinations and monitor these paths' health to choose among them.

3.2.3 DNS entries

Overlay nodes forward packets to each other over IP, but ICING-ON nodes are identified by ICING-PVM identifiers (node IDs, tags). How does a node find the IP address of the next node in the path? It uses DNS.

Whenever an overlay provider brings up a new node or service, that is whenever it needs to create a new node ID or tag, it will need to create a new entry in DNS. Such entries map *nodeID*:*tag* pairs to IP addresses and allow ICING-ON nodes and gateways to find the IP addresses for consent servers and other ICING-ON nodes and gateways. Gateways and nodes use these entries to find consent servers (named by `consent.`***nodeID***`.ICING-ON.net`) and to find the IP address of the next node in a path (named by ***tag***`.`***nodeID***`.ICING-ON.net`). We call these names the nodes' ICING-ON *names.*

To secure DNS entries, the provider signs each of its DNS entries using the private key corresponding to its node ID. Because the names being looked up are self-certifying [117], there is no need for a PKI to ensure the authenticity of the signatures.

3.2.4 To overlay or not to overlay

How does a sender know that it has to go through the overlay to reach a particular IP address or DNS name in the first place? And how does it know what ICING-ON name to use? Below are three options.

Static configuration: Senders may be configured a priori to use particular

ICING-ON names (node IDs and tags) for particular IP prefixes. The example in the next section shows how this option works in detail. The advantage is that this type of lookup is secure; the binding between name and IP cannot been forged by an attacker, because the attacker would need to change the static configuration of a sender. Unfortunately, static configuration does not handle the case of new IP addresses or DNS names well.

DNS lookups on the name or IP address: The owner of a server may insert an entry in DNS that maps the server's IP address or its DNS name to its ICING-ON name. To prevent an attacker from using fake entries to redirect traffic going to particular names or IP addresses, a system like DNSSEC [31] needs to be used. The disadvantages are clear: DNSSEC needs a PKI and an upgrade of DNS.

Rely on the correctness of IP routing: Our final solution is more end-to-end than DNSSEC. Initially, the sender sends the packet to its destination without going through the overlay, using the destination's IP address. If the destination requires passing through the overlay, the packet will be dropped by the destination's gateway. The gateway returns a new type of ICMP message that indicates the reason, and returns the ICING-ON name of the destination.

This solution is simple, but carries two disadvantages. First, the sender is relying on the correctness of IP routing to deliver packets to their inteded destinations through the destinations' gateways. Until systems like S-BGP [100] are deployed, this solution may not be completely secure. An attacker might be able to hijack the destination's IP address to receive traffic sent to that address. The attacker can then respond with any ICING-ON name it wants. Another disadvantage is the additional round-trip time (RTT) in the connection setup latency. However, this last disadvantage can be significantly mitigated using caching, because we do not expect destinations to be changing their policies about whether or not to go through the overlay or to be changing their gateways' node IDs very often.

None of these solutions is perfect, but they offer different trade-offs that suit different needs. We leave other solutions to future work.

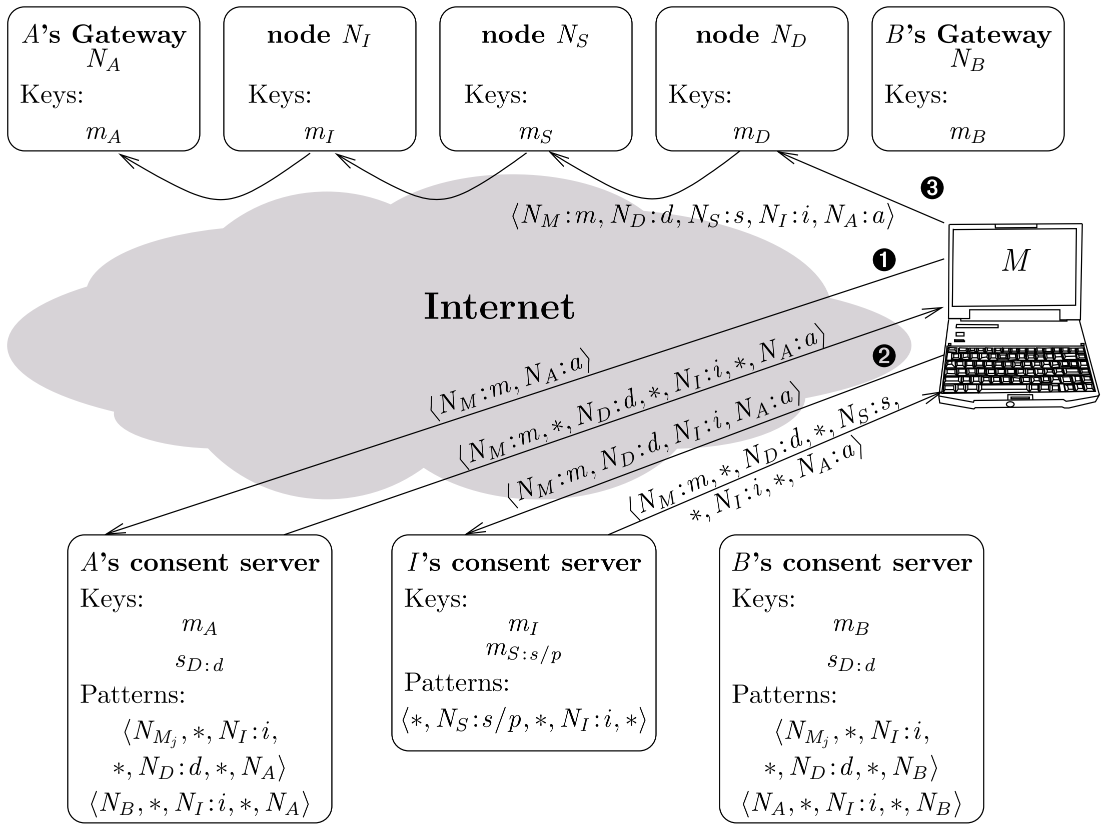
A's Gateway N_A
Keys: m_A
node N_I
Keys: m_I
node N_S
Keys: m_S
node N_D
Keys: m_D
B's Gateway N_B
Keys: m_B
❸
$\langle N_M\!:\!m, N_D\!:\!d, N_S\!:\!s, N_I\!:\!i, N_A\!:\!a\rangle$
❶
M
Internet
$\langle N_M\!:\!m, N_A\!:\!a\rangle$
$\langle N_M\!:\!m, *, N_D\!:\!d, *, N_I\!:\!i, *, N_A\!:\!a\rangle$
❷
$\langle N_M\!:\!m, N_D\!:\!d, N_I\!:\!i, N_A\!:\!a\rangle$
$\langle N_M\!:\!m, *, N_D\!:\!d, *, N_S\!:\!s,$
$*, N_I\!:\!i, *, N_A\!:\!a\rangle$
A's consent server
Keys:
m_A
$s_{D:d}$
Patterns:
$\langle N_{M_j}, *, N_I\!:\!i,$
$*, N_D\!:\!d, *, N_A\rangle$
$\langle N_B, *, N_I\!:\!i, *, N_A\rangle$
I's consent server
Keys:
m_I
$m_{S:s/p}$
Patterns:
$\langle *, N_S\!:\!s/p, *, N_I\!:\!i, *\rangle$
B's consent server
Keys:
m_B
$s_{D:d}$
Patterns:
$\langle N_{M_j}, *, N_I\!:\!i,$
$*, N_D\!:\!d, *, N_B\rangle$
$\langle N_A, *, N_I\!:\!i, *, N_B\rangle$

Figure 3.1: Example ICING-ON deployment and configuration, and how a sender obtains a path. In steps ❶ and ❷, the sender contacts the consent servers to build a path, and in step ❸, it constructs a path, obtains consent for it, and uses it to send data.

3.3 Example

We illustrate how the various components of ICING-ON fit together through an example. This example serves three purposes:

- Give an example of an application that ICING-ON enables;
- Describe how ICING-ON gateways, consent servers, and nodes interact;
- Describe how operators and administrators translate business relationships and policies into ICING-ON configurations and settings.

Say a company X has two branches, A and B, and mobile employees who are scattered globally. X wants all traffic from its mobile employees to either of its branches to go through both an outsourced IDS service, I, and a service, D, that does denial-of-service mitigation (e.g., [135]). On the other hand, X wants traffic between branches to travel only through I. Additionally, I itself requires all traffic arriving to it to pass through an accounting service S that bills I's customers for traffic going through I and drops traffic not arriving from its customers. Figure 3.1 shows ICING-ON's components, their configurations, and the process of obtaining a path. This figure is reminiscent of Figure 2.1. Next, we describe how ICING-ON's components enable X and I to implement their policies.

First, X contracts with D and I for service. D disintermediates itself from all consent granting and gives X tag prefix key $m_{D:d/p}$. X's administrators can thus generate tag keys of the form $s_{D:d}$, where d represents a specific tag from the d/p prefix block. The administrators use these keys to configure their consent servers. I holds on to consent granting because it needs to ensure that any paths through it also go through S. However, I does inform X that it must use a tag from prefix block i/q for traffic that goes through I.

Second, the administrators at A and B deploy on their network's external links standalone ICING-ON gateways, which will function as ICING-PVM senders with node IDs N_A and N_B respectively. Likewise, mobile employees install the ICING-ON gateway on their machines with unique node IDs N_{M_j} for $j = 1, 2, \ldots$. The gateways enable machines at A and B and mobile machines to access the overlay.

Third, the administrators deploy consent servers that mint PoCs consistent with X's policy. For instance, if N_I, N_D, and N_S are the node IDs for the overlay nodes at I, D, and S respectively, then, at A, the administrators configure the following allowed path patterns: $\langle N_{M_j}, *, N_{I:i}, *, N_{D:d}, *, N_A \rangle$ and $\langle N_B, *, N_{I:i}, *, N_A \rangle$. They also configure the server with tag key $s_{D:d}$, which will be used later to mint PoCs for traffic through the overlay. The administrators also configure a shared master key in the gateway and consent server at each branch: m_A at A and m_B at B.

Fourth, to enable other ICING-ON nodes and gateways to find their consent servers and gateways, the administrators at A and B install DNS entries. For example, for the name `consent.`N_A`.ICING-ON.net`, DNS returns the IP address of A's consent server, and for all other names that end with N_A`.ICING-ON.net`, DNS returns the

IP address of A's gateway.

Now, let's consider path and PoC retrieval. When a gateway receives an egress packet, it must decide (1) whether or not to send the packet through the overlay and (2) what path to use if the packet should go through the overlay. It bases these decisions on the destination IP prefix of the original packet. Say, for example, a packet is going from a mobile machine M to A. In that case, M's gateway selects a path $P = \langle N_M : m, N_A : a \rangle$, where m is the source IP address in the original packet and a is the destination IP address. Then it proceeds to obtain PoCs using SPB as described in Section 3.2.1. To obtain $\mathrm{PoC}_{N_A:a}$, the gateway looks up the IP address for A's consent server by looking up `consent.`N_A`.ICING-ON.net` and contacting the server, requesting a PoC for P. Because A requires that traffic from M pass through D and I, it rejects the request, returns pattern $\langle N_M : m, *, N_D : d, *, N_I : i, *, N_A : a \rangle$, and indicates that it can mint $\mathrm{PoC}_{N_D:d}$. M contacts I's consent server, which in turn returns pattern $\langle N_M : m, *, N_D : d, *, N_S : s, *, N_I : i, *, N_A : a \rangle$ and indicates that it can create $\mathrm{PoC}_{N_S:s}$. M finally constructs the path $\langle N_M : m, N_D : d, N_S : s, N_I : i, N_A : a \rangle$ and uses it to obtain all the needed PoCs from I's and A's consent servers. M caches this path, and makes sure to always have up-to-date PoCs to minimize new connection setup latency.

Finally, we consider packet sending. With the four PoCs in hand, M initializes the ICING-PVM header according to INITIALIZE (Figure 2.6). It then looks up $N_D : d$'s IP address in DNS using d`.`N_D`.ICING-ON.net` and sends the encapsulated packet to D over UDP. D, I, and A process the packet according to ICING-PVM's protocol and look up the next hops' IP addresses using DNS. If the gateway at A accepts a packet, A's administrator can be certain that the packet has followed the policies installed at A's consent server.

3.4 Discussion

The design of ICING-ON makes several trade-offs, often preferring simplicity over other features. Below, we discuss the trade-offs that were made in the choice of routing protocol, the use of DNS, and the interface that ICING-ON exposes.

ICING-ON chooses one particular routing protocol to find policy-compliant paths, SPB. Although simple and efficient (in that it allows a sender to discover all available paths), SPB has two limitations: it leaks information about providers' policies and it is an on-demand source routing protocol.

SPB reveals information about providers' policies because it returns patterns that reveal multiple paths to senders. This is not necessary if the sender is to have more than one choice (e.g. [174]), but necessary if that sender needs to know what choices exist. On the other hand, one reason providers want to hide their policies is to protect the routing protocol from being subverted. But because ICING-ON uses ICING-PVM as the forwarding mechanism, which ensures that providers' policies are always enforced, information hiding may be less important.

Second, SPB is an on-demand source routing protocol, similar to Dynamic Source Routing [97, 96], in that routes are constructed at connection setup. This construction increases ICING-ON's connection setup latency and may cause inconsistencies between state at the senders, consisting of cached paths that are supposedly policy-compliant, and state at consent servers, consisting of nodes' actual policies. To mitigate connection setup latency, senders both proactively construct paths and cache built paths. To mitigate inconsistencies, providers can decrease PoC expiration times. Shorter expiration times force senders to obtain PoCs more frequently, ensuring that the time windows of possible inconsistencies are short. And if a provider needs to immediately change its policies and revoke PoCs, it can change prefix or tag keys as described in Section 2.3.1. The sender would then notice a failure in the path, and either use a backup path or attempt to reconstruct a new one. Source routing protocols other than SPB may be used, but our goal in this chapter was to describe how a path *could* be built.

ICING-ON uses DNS to find lower layer addresses similarly to how IP networks use ARP. DNS is a compromise: in exchange for its simplicity and robustness, DNS forces ICING-ON to rely on root servers operated by third parties, thus giving up some of ICING-ON's decentralization. An alternative to DNS may be using DHTs (e.g., [116, 150]) for mapping ICING-PVM identifiers to IP addresses. Similar proposals were made in [159] and [158].

Finally, as so far described, ICING-ON does not expose an interface to applications that may want to choose their own paths (e.g., a peer-to-peer VoIP application choosing paths through participating nodes). The transparent ICING-ON gateways avoid requiring modifications to applications, and, when deployed as network middleboxes, to end-hosts. But to enable applications that do want to take advantage of path choice in ICING-ON, ICING-ON gateways pass through packets that already contain encapsulated ICING-PVM packets. Applications that wish to choose paths would link against an ICING-ON library that allows them to build their own ICING-ON packets.

A simplified version of ICING-ON has been implemented and deployed on EC2 [27] nodes. This simplified version of ICING-ON does not implement SPB and relies on static configuration of paths at ICING-ON gateways. Work is underway for a full implementation and a library.

What benefits does ICING-ON have and what advantages does it have over other overlays architectures? We mention three, corresponding to ICING-PVM's Path Consent, Path Compliance, and Delegation properties (Section 2.3). First, because senders obtain authorization before they send any data packets, and forwarding nodes only perform authentication on packets, policy is separated from forwarding and does not require per-flow state in nodes (unlike [52, 53]). Thus policy decisions may be arbitrarily complex, without compromising nodes' forwarding scalability. Second, because every node verifies that packets are following their approved paths, earlier nodes in the path drop non-compliant packets on behalf of later nodes in the path without explicit communication or coordination between them (unlike [33, 84]). Third, ICING-ON allows fine-grained and controlled delegation. It allows entities to delegate access through an ICING-ON node and constrain that delegation to be only for particular paths (unlike [136]).

ICING-ON is one step closer than just ICING-PVM to a full network architecture. However, ICING-ON does not answer one important question that ICING-PVM's off-by-default nature would raise if ICING-PVM were to be used as the forwarding mechanism of a network architecture: How does the sender get consent to request consent? In SPB, the sender first contacts the consent server of the destination to build a policy-compliant path. This would seem impossible under a default-off network. This question and others are answered in the next chapter.

Chapter 4

ICING-L3: A Policy-Compliant Layer-3 Network

4.1 Introduction

The Internet has spurred phenomenal innovation in large part because of a design philosophy, articulated in the end-to-end argument [141], which placed much functionality within the operating systems and applications at the endpoints of communications. While the end-to-end argument was originally motivated by correctness and the wish to avoid duplication of functionality, it proved to have an even greater benefit: it moved the implementation of many functions out of the network into the end-host, where developers and researchers could freely modify them to serve new unanticipated purposes.

Of course, several key functions still reside in boxes distributed throughout the network itself, where they are less easily modified. Notable examples are routing and various forms of policy—transit policy, ACLs, filters, firewalls, Quality of Service (QoS), etc.—that are implemented as protocols, RFCs, and vendor specific options ([80, 17, 22, 75, 115, 56, 57, 138, 76, 121, 124, 46]). Others ([83, 118, 52]) have already realized this problem and have begun to address it (e.g., OpenFlow [13, 85]) within a single trust domain.

By separating mechanism from policy and functionality, these systems allow the network operators, the actual stakeholders in the network, as opposed to the network vendors, to decide what functions they have in their networks. But OpenFlow does not address the question of how such separation would be implemented in a federated network where there are multiple trust domains interacting with one other, including senders and receivers. In particular, we are interested in interdomain routing policy for full end-to-end paths.

Why routing policy for full paths? Because enriching routing policy is the subject of a large body of interesting research with attractive applications, but that can never be deployed or evaluated in the field [168, 175, 64, 109, 40, 67, 136, 32, 60, 54, 81, 113, 166, 134, 172, 87, 174, 99, 66, 159, 149, 103, 28, 170, 108, 48, 169, 93, 151]. The union of the types of policies this research enables is a routing policy based on the full path, an observation previously mentioned in Section 2.7 and made in [146].

In this chapter, we want to answer two questions: How can routing policy decisions be completely separated from the forwarding mechanism? and, How can routing policy decisions be delegated to end-users to further the reach of the end-to-end principle?

How can routing policy decisions be completely separated from the forwarding mechanism? We require policy routing decisions and path approval to be made outside the network, a requirement reminiscent of [136, 52], with the difference being that the decision must be about the full interdomain path used for communication, including senders and receivers. But to truely enable such routing policy, the network must also enforce the policy's decisions. Otherwise, providers may undermine policies by forwarding packets through unapproved paths.

How can routing policy decisions be delegated to end-users? The main difficulty is that routing decisions for full paths cross trust domains. Thus, routing policy decisions cannot be made unilaterally by one entity (such as in traditional source-routing), because they might violate the policies of other entities on the path. ICING-PVM provides a mechanism that enables end-users to decide the paths they want to use across multiple providers without violating these providers' routing policies

To answer the above questions, we have designed a network architecture we call ICING-L3, which uses ICING-PVM as its forwarding mechanism. One of ICING-L3's main technical challenges is running control traffic—traffic that obtains paths and consent to use those paths—over the same default-off ICING-PVM data plane. It is not obvious how to bootstrap communication, i.e. get consent to request consent. Previous work that has dealt with default-off mechanisms has, to the best of our knowledge, always side-stepped the problem by assuming an always connected control plane. An always connected control plane requires different forwarding mechanisms for data and control traffic and leaves the control plane itself open to attacks [34].

To enable bootstrapping, ICING-L3 uses ICING-PVM's per-tag PoC keys and assumes that some of these keys are public. Its interdomain routing is heavily based on NIRA [172]. Along with distributing topology information, ICING-L3's routing protocol also distributes information on obtaining consent for paths through the topology.

Unlike NIRA, ICING-L3 gives all entities on chosen paths final say on the path if they would like to have it and enforces path choices in the network. ICING-L3 can drop packets that do not have policy-compliant paths early in the network, even if the violated policy is that of a later node in the path. So a receiver can choose to hear from specific senders and have its choices enforced by the first hop on the path.

4.2 Applications

We have already mentioned several applications that ICING-PVM and ICING-ON enable in Chapters 2 and 3. ICING-L3 enables all these applications as well as a few additional ones that arise from its default-off nature: pushing application access control into the network, early transit policy enforcement, and DoS mitigation.

Our first use case concerns application security. ICING-L3 enables an application to push its access control policies to the network. Consider, for instance, a legacy database that cannot be upgraded to fix potential security bugs. However, the remote users of the database are well-known (e.g., they are employees). The login process to the database can be migrated onto a consent server separate from the machine housing the database. The consent server mints PoCs for senders who are allowed to reach the database, after authenticating with the consent server. Unauthorized traffic cannot reach the machine running the database, so it cannot compromise any existing known bugs.

A second application is early transit policy enforcement. This means that packets that do not conform to the transit policies of all entities on the path are dropped by the first honest node on the path, with no explicit prior coordination with all other entities on the path. Dropping packets early is important in cases where packet delivery costs are high, such as for satellite links, emergency networks, or highly sensitive broadcast links (such as vehicular ad-hoc networks or VANETs).

For our final use case, we note that we have already touched on DoS mitigation in Section 2.3.5. Our mechanism (like many others [161, 74, 33, 30, 175, 168]) requires that the victim be able to distinguish malicious from normal traffic. For example, if the victim can distinguish customer, employee, or even just human traffic (e.g., by solving a CAPTCHA) from other undesired traffic, ICING-L3 allows the victim to stop undersired traffic early in the network. We have also mentioned how to protect the network from denial-of-capability attacks by increasing consent server availability or by presharing PoC keys with legitimate senders of traffic. ICING-L3 additionally enables earlier nodes on the path to drop packets that do not conform to later nodes' policies, again, without a priori explicit coordination between the two.

4.3 Overview

We begin our description of ICING-L3 by stating its three technical requirements:

- **ICING-PVM forwarding**: ICING-L3 must use ICING-PVM as its forwarding mechanism. We use this requirement to stand for all the properties that ICING-PVM enables—Path Consent, Path Compliance, and Delegation (Section 2.2.2).
- **Path and consent retrieval**: ICING-L3 must give senders a choice of paths to destinations and information on obtaining consent for those paths.
- **Early drops**: ICING-L3 must enable honest upstream nodes to drop packets that

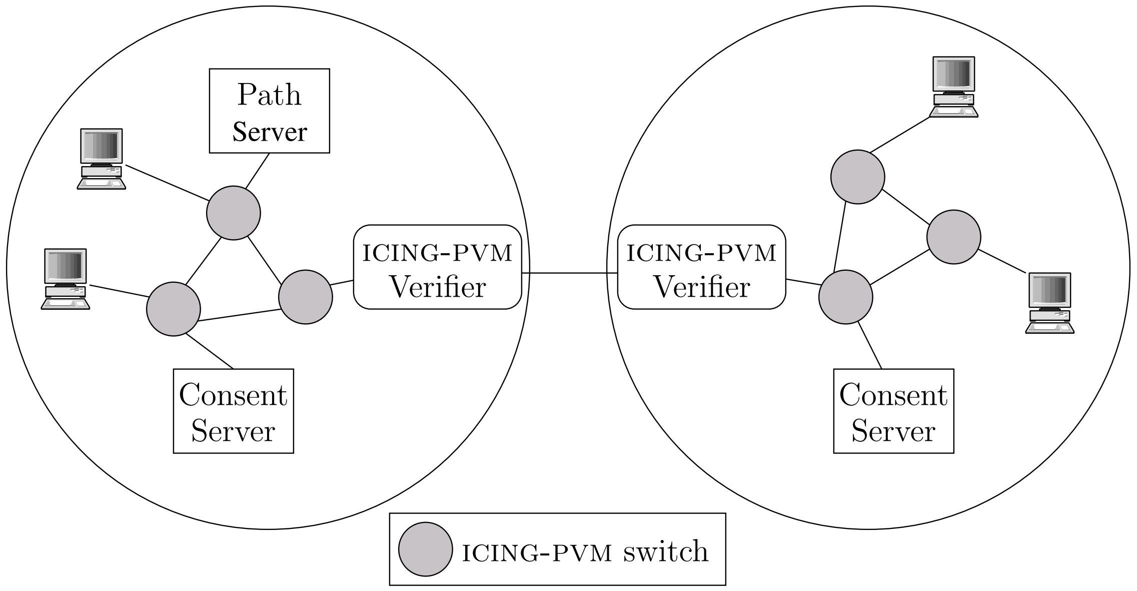

Figure 4.1: Components of an ICING-L3 network: end-hosts, ICING-PVM switches and verifiers, consent servers, and path servers. Shown are two provider networks.

do not comply with the path policies of downstream nodes.

As in all other network architectures, ICING-L3 divides the architecture into two planes: the data plane, which forwards packets through ICING-PVM nodes, and the control plane, which finds policy-compliant paths through the network and obtains consent for them.

4.3.1 ICING-L3 data plane

The data plane is composed of ICING-PVM nodes—forwarders, senders, and receivers. We split the job of forwarding into two types of node: ICING-PVM *verifiers* and ICING-PVM *switches* (Figure 4.1).

An ICING-PVM verifier implements the ICING-PVM protocol. A verifier is placed at each the ingress links of a provider's network to verify that ingress traffic complies with the provider's path policies. Since we expect providers to only install verifiers at their ingress links, we expect paths to only contain one verifier per provider network on the path.

An ICING-PVM switch, on the other hand, only forwards traffic based on the tag. Unlike a verifier, it does not have a node ID and is, thus, invisible to the ICING-PVM protocol. Its job is to provide a packet with the service that the tag it arrived through demands (e.g., forwarding to a particular host or through a low-latency path to a particular next hop). An ICING-PVM switch can be viewed as a very simple MPLS [140] switch. The configuration of ICING-PVM switches is left to the provider, and can use an intra-domain routing protocol or a centralized system using a protocol not unlike OpenFlow [13].

4.3.2 ICING-L3 control plane

Our objective is not to create a new control plane that makes the full power of ICING-PVM available, but rather demonstrate one way of obtaining paths. Because a path must be approved by all its constituents, any method for obtaining paths can be used, and ICING-PVM guarantees that only policy-compliant paths can be used.

Our example solution is heavily based on NIRA [172]. Under NIRA, each host learns of multiple *up-paths* to a set of well-connected providers that form the "Internet core". These up-paths are distributed downwards from the Internet core to customers and customers of customers, etc. using a path-vector routing protocol that exposes all such paths. A separate link-state protocol maintains dynamic information about availabilities and policies.

A host that wishes to be reachable by senders installs an entry in a lookup service. The entry maps a host-specific identifier (e.g., a DNS name) to a set of paths, *down-paths*, from the Internet core to the host. The host's down-paths are simply the reversed up-paths of that host. To send a packet to the host, a sender contacts the lookup service and receives the set of down-paths to the host. The sender chooses an up-path from its own set and a down-path from the received set such that the two paths intersect. The sender forms the full path using the chosen up-path as prefix and the chosen down-path as suffix.

Unlike NIRA, ICING-L3 uses the path-vector protocol for both topology discovery and policy distribution. ICING-PVM ensures that ICING-L3's data plane always uses policy-compliant paths even if the path-vector protocol is subverted or otherwise results in incorrect paths.

The ICING-L3 control plane consists of two services: the *consent service* (CS) and the *path-lookup service* (PS), shown Figure 4.1.

Consent servers have been described in Chapter 2. They store the provider's policy and provide PoCs that enable senders to use paths. They participate in ICING-L3's routing protocol, which we call *simple* ICING *Routing Protocol* or *sIRP*. sIRP is a path-vector protocol that distributes connectivity and policy information from providers to their customers and builds an up-path database at each provider's consent server. Whenever an end-host connects to a provider, it obtains its set of up-paths from the provider's consent server.

The path-lookup service is run by path servers. Path servers implement a lookup service similar to NIRA's name-to-route lookup service that maps host names to down-paths. Like NIRA, we do not specify whether the namespace is hierarchical or not or whether the path servers themselves are hierarchically organized or not. However, for the sake of exposition in the next section, we assume that host names are hierarchical DNS names and that the servers themselves are also arranged hierarchically.

Paths under ICING-L3 are useless to end-points without corresponding PoCs. Thus, whenever a consent server or path server returns a down-path, it also returns information on obtaining PoCs for that down-path. The next section describes the control plane in detail.

4.4 ICING-L3 control plane details

The control plane's job is to provide a sender with paths that it can use for traffic and PoCs for those paths. Our control plane is based on NIRA, and thus allows senders to find policy-compliant end-to-end paths in a scalable manner. Here, we describe how the two services that make up the control plane—the consent service and the path service—cooperate to perform the control plane's job.

We explain our design in the context of an example (Figure 4.2). We assume that

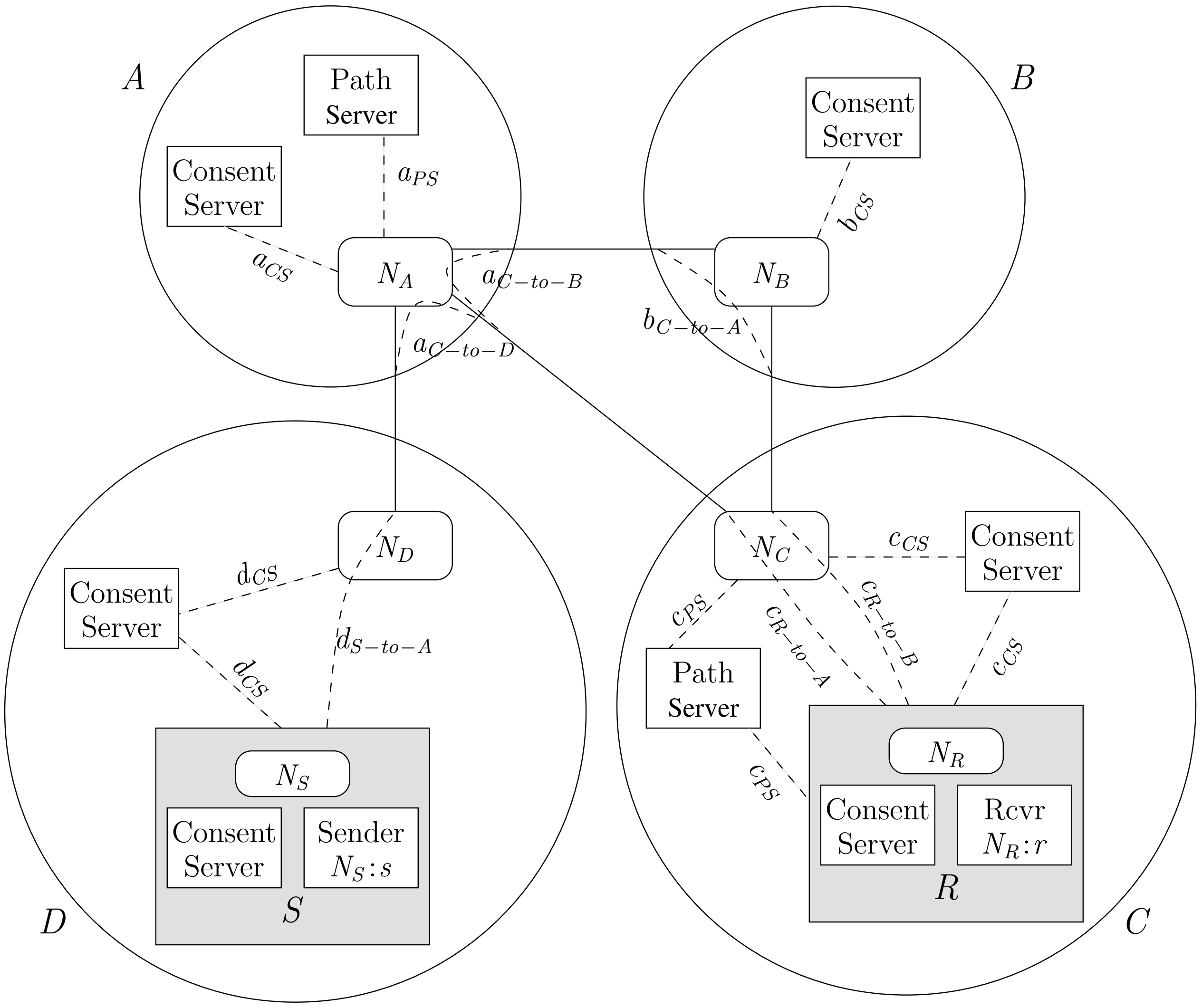

Figure 4.2: Example ICING-L3 network. The network consists of four providers, A, B, C, and D. A and B form the Internet core. S and R are end-hosts in D's and C's networks. Some intra-domain paths are shown (dashed lines) along with their tags.

the network consists of four providers, A, B, C, and D. A and B are peers that form the "Internet core", C is a customer of both A and B, and D is a customer of A. R is an end-host in C's network and S is an end-host in D's network. R runs a server that S wishes to access.

As mentioned previously, tags are used for intra-domain routing. So the prefix a_{C-to-B}/p can indicate a set of tags that all refer to a path through A's network for transiting traffic from C to B. To simplify exposition, we assume that tags can be used bidirectionally: $a_{C-to-B} = a_{B-to-C}$. In reality, however, these two tags may be separate and paths may be unidirectional. For now we assume that the protocols below provide paths that may be traversed in both directions. In our example and figures, we will use the notation of a single tag to refer to a tag prefix. For instance, a_{C-to-B} is short for a_{C-to-B}/p for some $0 \leq p \leq 32$ (i.e., also including fully-specified /32 tags).

We describe how ICING-L3 works without any delegation first, and then describe how delegation can be used to optimize the control plane.

4.4.1 Learning up-paths

Up-paths are distributed downwards through a provider's hierarchy using sIRP. *sIRP advertisements* contain all the up-paths that the receiver may use, as well as information on how to obtain PoCs for each hop on those paths. sIRP advertisements might also include signed policy declarations, called *consent certificates*. Consent certificates contain regular expressions describing paths for which consent servers will mint PoCs. An sIRP advertisement is one of several types of sIRP message, described in Figure 4.3.

For each hop along a path advertised in the sIRP advertisement, the advertisement also has access information. Access information indicates how the receiver of the sIRP message can obtain a PoC for a hop, either by divulging the tag key for that hop or by indicating the path to a consent server that can mint PoCs for that hop. Again, the path to such a consent server must have its own access information

Whenever a consent server receives an sIRP advertisement, it verifies the signatures in the consent certificates, stores the advertisement, modifies the paths according to its policies, and then transmits new sIRP advertisements down through its own hierarchy. A consent server modifies paths in three ways: it augments paths with its own node ID and tag prefixes, it inserts additional access information, and inserts new consent certificates. Each consent server in the provider hierarchy eventually receives sIRP advertisements with up-paths to the Internet core or to peers of providers in

An sIRP message conveys three pieces of information, each of which might be missing depending on the message type:

```
       advertisement  =  <up-paths, consent-certificates>
           bootstrap  =  <up-paths, consent-certificates, path-servers>
path-server-response  =  <path-servers, consent-certificates>
```

Up-paths are paths that the receiver of this message can use to reach the Internet core:

```
            up-paths  =  <path*>
```

`path-servers` provides information on where to reach path servers:

```
        path-servers  =  <path*>
```

A path is a set of hops:

```
                path  =  <hopinfo, hopinfo*>
```

Each hop has 3 pieces of information, a node ID, a tag prefix, and information on obtaining a PoC for the node ID and prefix pair:

```
             hopinfo  =  <nodeid:prefix, <access, access*>>
```

Access information can either be a tag prefix key, making the tag public, or a path to a consent server that can mint the PoC for the node ID and prefix pair:

```
              access  =  tagkey | cspath
              cspath  =  <path, path*>
```

A consent certificate is a signed declaration of the patterns (`pattern`) for which a consent server (`signed-by`) will mint PoCs for the node ID and prefix in `for-prefix`

`consent certificate`	=	`for-prefix, pattern, signed-by, signature`
`for-prefix`	=	`nodeid:prefix`
`nodeid`	=	The ID N_i of an ICING-PVM node
`prefix`	=	A tag prefix (or tag) t_i/p
`tagkey`	=	The prefix or tag key $m_{N_i:t_i/p}$
`pattern`	=	A regular expression describing allowed paths
`signed-by`	=	$N_{CS}:t_{CS}$ pair for a consent server.
`signature`	=	Signature using the private key corresponding to N_{CS} covering the contents of the `consent certificate`

Figure 4.3: EBNP-like description of the contents of an sIRP message.

these up-paths.

In our example, each of A, B, C, and D run their own consent servers that can be accessed at $N_A:a_{CS}$, $N_B:b_{CS}$, $N_C:c_{CS}$, and $N_D:d_{CS}$ respectively. Each provider obtains from its customers a tag key that allows access to the customers' consent

sIRP Advertisement from $N_A\!:\!a_{CS}$ to $N_C\!:\!c_{CS}$			
Up-Paths	$\langle N_A\!:\!a_{any}\rangle$ $\langle N_A\!:\!a_{C-to-B}, N_B\!:\!b_{any}\rangle$		
Access Information	Hop	Type	Value
	$N_A\!:\!a_{any}$	`cspath`	$\langle N_A\!:\!a_{CS}\rangle$
	$N_A\!:\!a_{CS}$	`tagkey`	$s_{N_A:a_{CS}}$
	$N_B\!:\!b_{any}$	`cspath`	$\langle N_B\!:\!b_{CS}\rangle$
	$N_B\!:\!b_{CS}$	`tagkey`	$s_{N_B:b_{CS}}$
Consent Certificates	For	Pattern	Signed by
	$N_A\!:\!a_{C-to-any}$	$\langle *, N_C\!:\!c_{any}, N_A\!:\!a_{C-to-any}, *\rangle$	$N_A\!:\!a_{CS}$

Figure 4.4: sIRP advertisement from A to C.

sIRP Advertisement from $N_C\!:\!c_{CS}$ to $N_R\!:\!r_{CS}$			
Up-Paths	$\langle N_C\!:\!c_{any}\rangle$ $\langle N_C\!:\!c_{R-to-A}, N_A\!:\!a_{any}\rangle$ $\langle N_C\!:\!c_{R-to-A}, N_A\!:\!a_{C-to-B}, N_B\!:\!b_{any}\rangle$ $\langle N_C\!:\!c_{R-to-B}, N_B\!:\!b_{any}\rangle$ $\langle N_C\!:\!c_{R-to-B}, N_B\!:\!b_{C-to-A}, N_A\!:\!a_{any}\rangle$		
Access Information	Hop	Type	Value
	$N_C\!:\!c_{any}$	`cspath`	$\langle N_C\!:\!c_{CS}\rangle$
	$N_C\!:\!c_{CS}$	`tagkey`	$s_{N_C:c_{CS}}$
	$N_A\!:\!a_{any}$	`cspath`	$\langle N_A\!:\!a_{CS}\rangle$
	$N_A\!:\!a_{CS}$	`tagkey`	$s_{N_A:a_{CS}}$
	$N_B\!:\!b_{any}$	`cspath`	$\langle N_B\!:\!b_{CS}\rangle$
	$N_B\!:\!b_{CS}$	`tagkey`	$s_{N_B:a_{CS}}$
Consent Certificates	For	Pattern	Signed by
	$N_C\!:\!c_{R-to-any}$	$\langle *, N_R\!:\!r_{any}, N_C\!:\!c_{R-to-any}, *\rangle$	$N_C\!:\!c_{CS}$
	$N_A\!:\!a_{C-to-any}$	$\langle *, N_C\!:\!c_{any}, N_A\!:\!a_{C-to-any}, *\rangle$	$N_A\!:\!a_{CS}$
	$N_B\!:\!b_{C-to-any}$	$\langle *, N_C\!:\!c_{any}, N_B\!:\!b_{C-to-any}, *\rangle$	$N_B\!:\!b_{CS}$
Path Servers	$\langle N_C\!:\!c_{PS}\rangle$ $\langle N_A\!:\!a_{PS}\rangle$		

Figure 4.5: sIRP bootstrap advertisement from C to R.

servers. For instance, C gives A the tag key $s_{N_C:c_{CS}}$, so A's consent server can send sIRP advertisements to C's consent server using the path $\langle N_A\!:\!a_{CS-to-C}, N_C\!:\!c_{CS}\rangle$.

The sIRP advertisement that C's consent server receives from A is shown in Figure 4.4. B's advertisement to C is similar. The advertisement has three tables:

Up-Paths, Access Information, and Consent Certificates.

The Up-Paths table informs C of the paths that it can use through A. For example, the second entry informs C that it can use the tag a_{C-to-B} through A to get to any tag on B.

The Access Information table informs C how it can get consent for each hop on each path in the Up-Paths table. For example, the first entry gives C the down-path to A's consent server $N_A\!:\!a_{CS}$ that should be accessed for any PoC requests through N_A. How does C get consent to access the consent server? Using the second entry, which gives C the tag (or prefix) key, $s_{N_A:a_{CS}}$, for $N_A\!:\!a_{CS}$. A could have run its consent server somewhere other than in its network. However, A would then need to provide information that would enable C to reach the consent server.

The Consent Certificates table contains entries that describe the policies of the nodes in the Up-Paths and the Access Information tables. These consent certificates can be used to prove to various entities that a path is policy-compliant. The single entry in our example declares that A's consent server is willing to mint PoCs for any paths from C through A, and only for the tags that are connected to C. Consent certificates are signed by the private key corresponding to the node ID of the node making the declaration, so verifying them does not require a PKI [1].

When the end-host R joins C's network, it broadcasts a bootstrap request in its subnet (similar to DHCP), announcing 1) its node ID, N_R, 2) a tag that connects to its consent server (running on the same end-host), $N_R\!:\!r_{cs}$, and 3) the consent server's tag key, $s_{N_R:r_{cs}}$. C's consent server replies with the *bootstrap sIRP message* shown in Figure 4.5. Like a regular advertisement, a bootstrap message informs the receiver of up-paths, access information for these up-paths, and any known consent certificates. Additionally, a bootstrap message, like a DHCP reply, indicates how to reach name resolution servers in the Path Servers table. This table has information on reaching path servers that can be used to map names to down-paths for end-hosts. These

[1]Dual-usage of public key pairs for signatures and for key exchanges is usually discouraged. However, our particular case can be proven secure by a careful combination of standard security arguments for Diffie-Hellman key exchanges and Schnorr signatures. Similar proofs have been done in the past [153], where, under appropriate padding, using the same key pair for RSA decryption and signatures was shown to be safe.

Name	Type	Record	Access Info		
			Hop	Type	Value
`example.com`	`PS`	$\langle N_A\!:\!a_{to-C-CS}, N_C\!:\!c_{CS}\rangle$	$N_A\!:\!a_{to-C-CS}$	`tagkey`	$s_{N_A:a_{to-C-CS}}$
			$N_C\!:\!c_{CS}$	`tagkey`	$s_{N_C:c_{CS}}$

Figure 4.6: *A*'s path server configuration.

paths will be used in the next section. Note that paths to path servers might need additional entries in the Access Information table.

4.4.2 Learning down-paths and sending packets

Distribution of down-paths requires a system parallel to consent servers, consisting of path servers. Path servers translate names to down-paths that can be combined with a sender's up-paths to reach either the sender's destination or another path server to continue the query. For the purposes of this exposition, path servers are hierarchically structured and resolve a hierarchical namespace, just as DNS does. However, other methods of name resolution would also work, using the same concepts and message format we describe below.

As an example, we assume that each of A and C operate public path servers at $N_A\!:\!a_{PS}$ and $N_C\!:\!c_{PS}$ respectively. Note that the path servers can have their own node IDs if desired. Say A's path server is a root server that is authoritative for the name `com`, while C's is authoritative for the name `example.com`. We assume that R wishes to be found using the name `ar.example.com`. Lookups proceed hierarchically just as in DNS.

Figures 4.6 and 4.7 show the configuration of the path servers at A and C respectively. The entry in A's path server is added by the owner of the `example.com` domain, while the entry in C's path server is added by the owner of `ar.example.com`.

When S, an end-host in D's network, wishes to contact `ar.example.com`, it uses its stored information about the location of the root path server, $N_A\!:\!a_{PS}$, and its up-path to construct the path $\langle N_S\!:\!s, N_D\!:\!d_{S-to-A}, N_A\!:\!a_{PS}\rangle$. S requests a PoC from D's consent server at $N_D\!:\!d_{CS}$. And since S learns the tag key for $N_A\!:\!a_{PS}$ during

Name	Type	Record	Access Info		
			Hop	Type	Value
`ar.example.com`	`DST`	$\langle N_A : a_{to-C}, N_C : c_{A-to-R}, N_R : r_{ar} \rangle$	$N_A : a_{to-C}$	`tagkey`	$s_{N_A : a_{to-C}}$
			$N_C : c_{A-to-R}$	`tagkey`	$s_{N_C : c_{A-to-R}}$
			$N_R : r_{ar}$	`tagkey`	$s_{N_R : r_{ar}}$
			Hop	Type	Value
	`DST`	$\langle N_B : b_{to-C}, N_C : c_{B-to-R}, N_R : r_{ar} \rangle$	$N_B : a_{to-C}$	`tagkey`	$s_{N_B : b_{to-C}}$
			$N_C : c_{B-to-R}$	`tagkey`	$s_{N_C : c_{B-to-R}}$
			$N_R : r_{ar}$	`tagkey`	$s_{N_R : r_{ar}}$

Figure 4.7: C's path server configuration.

bootstrapping, S mints its own PoC for that hop. S contacts A's path server, and obtains the entry for `example.com`. Using the returned record and its own up-paths, S constructs a path to C's path server along with PoCs. It obtains down-paths to R from C's path server and proceeds to construct a full end-to-end path. Using the access information for its own up-paths and the access information for the down-paths from the path server entry, S can create or obtain PoCs for the path it chooses. Finally, S stores this path for later use.

Note that the locations and tag keys of root path servers are statically configured. As in NIRA, if the paths to root servers in ICING-L3 change, then this static configuration will need to be changed as well. One way to mitigate this problem is by placing the root servers in the Internet core, where paths are less likely to change.

One issue that we have ignored in our description so far is that of return paths. What path does a path server, for example, use to return a record? ICING-L3 packets that setup a connection carry return paths and the paths' PoCs to reduce the load on the server.

4.4.3 Early drops

So far we have described how ICING-L3 uses ICING-PVM for forwarding and how senders get paths and consent to request consent, thus meeting our first two requirements from Section 4.3. How does ICING-L3 drop packets that are non-policy compliant early? Using Consent Certificates.

Consent servers can refuse to issue consent for a path unless they have consent certificates that certify that every entity on the path approves of the path. Senders can obtain these consent certificates in two ways: through sIRP advertisements or by explicitly requesting them. We have already described how the certificates are transmitted through sIRP advertisements in Section 4.4.1. To explicitly request certificates from their issuers (themselves consent servers), senders go through an additional round of contacting consent servers while obtaining PoCs for a path. In the first round, senders obtain consent certificates from all consent servers of nodes on the path. In the second round, they exchange the consent certificates for PoCs. Senders that cannot assemble a full set of consent certificates for a path would not be able to send traffic through that path.

4.4.4 Delegation

The business of obtaining consent to request consent is complex and may result in large delays in connection setup. It also places a burden on providers to run scalable consent services that can handle the demand for paths through their networks. To mitigate these problems, ICING-L3 enables providers to delegate particular tag prefixes through their networks to customers or to make some public (as was done in our example above for path server tags).

For example, if A wishes to completely disintermediate itself from the consent process, it may give C a private tag prefix key that it can use to mint PoCs for tags through A. Similarly, A can delegate non-intersecting tag prefixes to each of its other customers and peers, allowing them to mint PoCs through its network. A's customers and peers may themselves delegate smaller chunks of their given blocks to their own customers. Eventually, end-hosts (or at least edge-providers) may obtain tag keys that allow them to mint PoCs for most of their up-paths (or down-paths). We expect complete disintermediation to be the usual case since it preserves valley-free forwarding.

A provider that holds delegated keys may mint PoCs for tags in other providers' networks. To enable its customers to take advantage of this optimization, a provider needs to add entries in the Access Information table of propagated sIRP advertisements. The additional entries inform the advertisements' recipients of the particular prefixes for which the provider can mint PoCs and the location of the provider's consent server. The provider may also add consent certificates in sIRP advertisements to inform its customers of the paths it is willing to approve.

4.4.5 Optimizations

Like consent servers and path servers, most end-hosts will not need to have their own node IDs. Instead, such an end-host uses as identifier its provider's node ID and a tag prefix assigned by the provider. When sending a packet, the end-host only puts a path in the packet and initializes the verifiers in the packet with that path's PoCs; i.e., it does not execute lines 12–14 in Figure 2.6. These steps are executed, upon observing an egress packet from an internal end-host, by the provider's ICING-PVM verifier. Similarly, the job of authenticating an ingress packet destined to an internal end-host is left to the provider's border ICING-PVM verifier; i.e., the end-host does not execute the code in Figure 2.7 and always accepts arriving packets. This scenario requires that the end-host trusts the provider's ICING-PVM nodes to obey the ICING-PVM protocol. The end-host can still run its own consent server with prefix keys delegated by the provider.

We also expect that most providers will use only a small number (most likely one) of node IDs for their ICING-PVM verifiers, naming many ICING-PVM verifiers with the same node ID. This reuse eases the provider's burden of public key management and policy writing and makes the architecture more scalable (Section 2.5).

4.4.6 Intradomain Routing

A detailed design of intradomain routing and forwarding is part of our future efforts on ICING. Here, we only give a flavor of a possible solution for intradomain routing.[2]

[2]We also omit the description of how multicast would work.

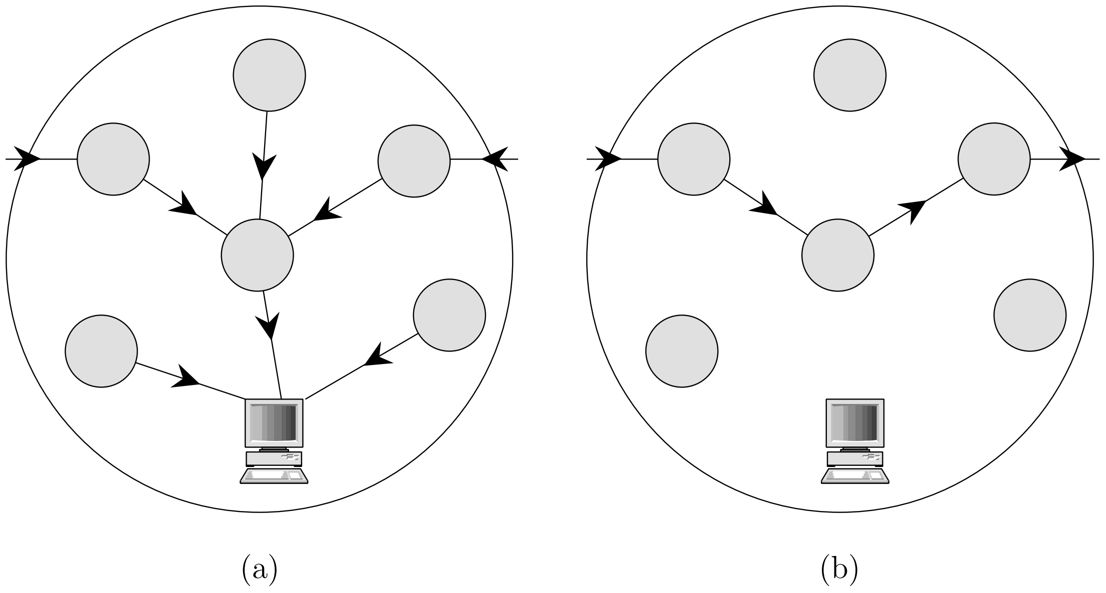

Figure 4.8: Examples of both types of intradomain paths. (a) is a tree path directing all traffic towards an end-host and (b) is a linear path connecting one neighboring network to another.

We assume that any particular provider's network will have a finite number of allowed intradomain paths. These consist of paths that connect a provider's neighbors to each other, paths that end at connected end-hosts, and paths from connected end-hosts to neighbors. The job of the intradomain routing protocol is to find these paths and assign one or more tags to each of them. An intradomain ICING-PVM switch can then simply forward based on the tag (just like an MPLS Label Switch Router or LSR).

There are two types of path: paths to a particular traffic sink (end-host or network exit point) or paths between a particular traffic source (end-host or network entry point) and sink (Figure 4.8). By default, the intradomain protocol finds paths of the earlier type. These paths are directed tree graphs rooted at the sink. Operators can specify additional constraints such as restricting the sources that can send to particular sinks, or restricting the sinks a particular source may send to. The additional constraints either prune some path trees or create new linear paths between particular sources and sinks.

The routing protocol, using hints from the operator, names the resulting paths using tags or prefixes and installs entries in all the ICING-PVM switches in the network for them. The routing protocol can dynamically recalculate paths on failure, but does not rename a path unless its constraining points (i.e. the destination for trees or the source/destination pair for linear paths) change. Since a path may be named by a prefix rather than a single tag, an operator may delegate different tags for one path to different entities. Such a delegation enables the operator to keep track of which delegate is sending traffic through the path (e.g., for accounting purposes). Another use of multiple names for a path is to specify different quality of service levels for the same path.

4.5 Discussion

To the best of our knowledge, ICING-L3 is the only default-off network that runs control traffic on the same default-off data plane as regular traffic. In ICING-L3, there is no "default-on" connectivity. Any network access, even for control traffic, requires some form of authorization. For simplicity of exposition, we have discussed examples where some tag prefix keys required for authorization have been made public. In reality, the network could run only with *delegated* keys that are given to trusted entities, at the cost of yet more complexity.

Even though ICING-L3's design may seem complex, the concepts on which it rests are simple: a routing protocol runs on a bootstrap path accessible only to neighbors; the routing protocol distributes path and access information; and, access information is recursively defined (i.e., might need consent to get consent).

ICING-L3 may seem to have a high overhead, especially with respect to connection setup latencies. However, if we assume that most providers delegate tag prefixes down to end-hosts, then finding down-paths will take latencies on the same order as those of DNS requests. The main cost will be that of data plane functions for sending packets.

As for management cost, ICING-L3 does require neighboring providers to at least exchange bootstrap keys; however, this exchange is similar in cost to providers exchanging certificates or public keys. To reduce the management costs of delegation, the process of obtaining delegated prefixes from peers and providers and redelegating them to customers could be automated and built into the routing protocol.

ICING-L3's routing protocol sIRP does not discover all potentially available paths. This is a compromise to maintain scalability. It also suffers from the possibility of providers hiding some up-paths from their customers (as opposed to simply not consenting to those paths). In any case, sIRP can be used to bootstrap access to other routing protocols or other methods for obtaining paths, without providers worrying about violation of their policies.

Chapter 5

Related Work

We divide related work into three areas: (1) network security, (2) related mechanisms, and (3) policy routing.

5.1 Securing the network

Much work can be placed under the network security umbrella. Routing security [100, 93, 151, 26, 92, 176, 128, 129, 106, 90, 49, 160, 91] ensures the authenticity and correctness of topology propagation and route computation. For instance, S-BGP [100] protects BGP against spurious messages. However, these works do not ensure that the resulting routes are actually used in packet forwarding, mainly because doing so requires a change in the forwarding mechanism, which is ICING-PVM's focus and which we view as complementary.

Like ICING-PVM, other network security proposals bind packets to their purported paths. However, some of this work is not geared to a high-speed, federated environment. For example, [131, 51, 43] require per-packet digital signatures, [38] requires large configuration state in the network and a packet header quadratic in path length, and [52] requires a centralized controller.

Other work on forwarding security is geared toward secrecy or isolation. Virtual Private Networks (VPNs) [157], whether meant as point-to-point IPSec [79] tunnels or as isolated "slices" of a provider's network using MPLS [140], provide neither Path

Consent nor Path Compliance.

Others have also looked at source-routing that complies with transit providers' policies. Rule-Based Forwarding (RBF) [134] and Platypus [136], like ICING, observe that giving end-points additional control must respect the service providers' policies. RBF gives end-points restricted control over some forwarding functionality by allowing senders to specify rules. RBF ensures that these rules are approved by all entities that the rules affect. Platypus enables senders to choose waypoints through the network and enables each provider to identify an entity accountable for every packet through the provider's waypoints. Like ICING-ON, RBF and Platypus run on top of IP. RBF makes several assumptions about the underlying infrastructure: anti-spoofing measures, an existing PKI, and highly available DNS servers. ICING-ON and ICING-L3 only make the last assumption, and in ICING-L3, only for the root server. Unlike RBF, ICING-L3 does allow delegating routing policy, and thus does not need to run on top of IP. Finally, while RBF and Platypus both allow entities to choose which senders can send them traffic, they do not allow them to approve full paths, thus they do not provide Path Consent, nor do they provide Path Compliance.

Other notions of network security are complementary to ICING's. Onion routing [63] aims to provide anonymity. As a bonus side-effect, it also ensures that the sender's specified path is followed because otherwise the onion-encrypted packet fails its cryptographic checks. However, it does not provide Path Consent. It is also computationally expensive: the sender encrypts the packet multiple times, and each hop decrypts the whole packet.

Work on securely localizing faults [127, 142, 165, 39, 178, 177, 41, 35] is complementary. There, the objective is to pinpoint where packets are dropped in an adversarial network in which nodes may attempt to distort this information. Work on Byzantine routing builds on top of secure fault localization to ensure availability in a network if there is at least one non-faulty path between sender and receiver [131, 132, 38, 120, 39, 69]. Other proposals are geared toward denial-of-service (DoS) protection and allow receivers to control which senders can reach them [101, 64, 109, 168, 175, 40, 88, 112, 94, 144, 58, 84, 167, 170, 33, 74]. These mechanisms can be enhanced to securely identify packets' senders [28, 108, 48, 173, 95, 107, 169], enabling accountability. ICING-ON and ICING-L3 likewise provide support for DoS protection and verify not only *sources* but also *paths*. Further, ICING-L3 shows how to use the mechanism to protect control traffic, and prevent the "denial-of-capability" [34] attack often insufficiently addressed in many of these proposals.

5.2 Related mechanisms

PVM is a new primitive, but aspects of ICING-PVM, ICING-ON, and ICING-L3 are inspired by prior works. PoCs generalize network capabilities [136, 175, 168] and Visas [67]. For instance, under TVA [175], receivers can control which senders reach them. However, none of these works provides Path Consent or Path Compliance.

Other mechanisms related to ICING-PVM are as follows. Node IDs resemble the self-certifying [117] ADs in AIP [28]. PoPs are reminiscent of constructions in [41, 38, 37]. However in those works, the number of PoP-like things in a packet is quadratic in path length, whereas an ICING-PVM packet carries a linear number of PoPs. Goldberg et al. [82] hint at using Diffie-Hellman key exchanges for creating pairwise keys between nodes (in analogy with ICING-PVM's PoP keys), but they suggest using a PKI, which ICING-PVM does not need (since node IDs are public keys). Also, ICING-PVM's hierarchical delegation generalizes a technique in Platypus [136], ICING-PVM's tags are reminiscent of Pathlets' vnodes [81] and MPLS labels [140], and expressing policy in general-purpose servers apart from forwarding hardware echoes [136, 83, 50, 52, 85]. ICING-ON is an overlay network [61] and shares many concepts used in other overlay proposals, particularly those dealing with routing [29, 143, 101, 159, 136, 149, 87, 63]. ICING-L3 owes much of its routing mechanism to NIRA [172].

5.3 Policy routing

Like BGP itself, many works [32, 113, 172, 87, 174, 99, 81, 86, 159, 149, 166] allow entities to express various path preferences. Under NIRA [172], for instance, senders choose the path into the Internet core, and receivers choose the path out. Indeed, even default-off and filtering can be regarded as a kind of policy routing, in that the receiver exercises control over the first path component, the sender (e.g. [64, 109, 168, 175, 40, 94, 167, 170]). Source routing [32, 113, 172, 87, 174, 99] and Byzantine routing [131, 132, 38, 120] proposals allow senders to choose paths, but do not take into account the full policies of all providers and receivers on these paths. [174] provides provider-approved path diversity, but does not allow senders or receivers to choose particular paths. RouteScience's PathControl [18] extends outbound BGP path-selection criteria using end-to-end measurements such as latency.

These proposals are either orthogonal to ICING (because they concern only path computation, not enforcement), or else they incorporate a mechanism that enforces something less general than Path Consent or Path Compliance. For example, under Pathlets [81], senders choose paths, and providers specify policies based on the previous hop and a suffix of the path. But Pathlets do not provide verification that a path was actually followed. NUTSS [86], DOA [159], and i3 [149] integrate middle-boxes or other intermediaries into path selection and forwarding but do not provide Path Consent or Compliance. ICING-PVM can be regarded as providing an enforcement mechanism that is general enough to enforce many of the policies in the works cited above.

Chapter 6

Conclusion

6.1 Summary

The ICING project was motivated by the desire for new functionality in the Internet: we wanted to enable new applications by delegating control to end-users (senders and receivers). We wanted to allow end-users to control the paths their packets take through the network and choose the services that are applied on their packets. Unfortunately, this delegation cannot be done without risking the transit service providers' viability—the paths that end-users choose might violate the providers' policies or be very expensive. Additionally, providers need to be compensated for transit or other value-added services they offer.

Given such an environment, two requirements arise. First, the path that eventually gets used for a communication between two end-points needs to be approved by all stakeholders in that communication. These stakeholders are the sender, the receiver, and all traffic carriers in between[1]. Second, the network must ensure that the traffic does actually follow the approved path. We called these two requirements Path Consent and Path Compliance respectively. Path Consent and Path Compliance are not met by today's Internet forwarding mechanism (IP). Thus, we offered a networking primitive that does meet them: PVM. (Section 2.1).

[1]Certainly, if the communication contains bank account information, then the account's owner is also a stakeholder. However, here, we are only concerned with network-level stakeholders.

ICING-PVM shows how a PVM can be built (Sections 2.2 and 2.3). We evaluated ICING-PVM's scaling properties, and found that it should be feasible to scale ICING-PVM to Internet backbone speeds (Section 2.5). We showed how ICING-PVM can be deployed in two ways: ICING-ON, an overlay on top of IP (Chapter 3), and ICING-L3, a layer 3 replacement for IP (Chapter 4). In both cases, we showed how a sender can find a path that complies with its own policies and the policies of all other stakeholders on the path.

6.2 Future Work

This dissertation leaves many questions unanswered for future consideration.

Economic and social impacts: If ICING-L3 were to be deployed as the future Internet, it would impact the providers' behavior and business models as well as how end-users use the network. We leave investigation of these impacts to future work under the ICING project.

Solving replay attacks: ICING-PVM only provides minor protection agains replay attacks. Future work will investigate stronger guarantees.

Linking tag keys to paths: Currently, if a provider delegates a tag key to another entity, that entity gains the ability to create PoCs for any path using that tag. Future work will investigate the possibility of ensuring that particular keys are only used for particular path patterns.

Service level agreement violations and secure localization of faults: ICING-PVM provides a way for nodes to verify whether a packet followed an approved path, but not whether it received a particular service. It also does not give senders the ability to detect the location of *silent* packet drops or other errors. Future work will investigate whether ICING-PVM can help with these two problems given its authentication mechanisms.

Accounting and providing cost information: ICING-L3 and ICING-ON do not give senders any cost information on which to base their decisions. Future work will extend sIRP and SBP to provide additional information that can help senders understand the impact of their choice of paths.

Intradomain routing: Future work will investigate how intradomain routing can interoperate with ICING-L3's interdomain routing protocol, sIRP.

Cryptographic improvements: 163-bit public ECC keys provide 80-bit security, which NIST considers insufficient in the future. Similarly, ICING-PVM's chosen secure hash function, CHI, has been eliminated from the SHA-3 competition. Future work will address both these issues.

6.3 Insights

We conclude this part of the dissertation with insights and lessons learned over the course of the ICING project.

The costs of security, generality, and decentralization The ICING project has provided a mechanism, ICING-PVM, that can enforce a wide variety of policies in a decentralized environment. However, this combination of security, generality, and decentralization has come at a cost. Not only does ICING-PVM have a much higher packet overhead than IP, it also has more expensive hardware. On the other hand, ICING-PVM does enable a wide range of applications that would otherwise be impossible.

The costs of default-off networks ICING-L3 shows how to build a default-off network without using a separate network for control traffic. We expect similar networks to follow the same basic design principle: a sender starts with limited *delegated* access to bootstrap servers that trust the sender (or servers that are otherwise public) and then gains privileges to access other parts of the network. ICING-L3 also highlights the costs of such a default-off network—connection setup time and configuration complexity.

Per-packet cryptographic hardware ICING-PVM shows that it is possible to build a packet forwarding mechanism with sophisticated per-packet cryptography at feasible hardware costs. Our experience has taught us the importance of the following:

- **Parallel and cachable operations**: Obviously, parallelizable operations are conducive to high-throughput designs. The problem is that many cryptographic operations have a serial part that needs to be executed at the beginning. For example, PMAC [44] requires that a key is passed through an AES encryption operation before the parallel execution of the rest of the algorithm. The trick is to store both the key and its encrypted form. As such, similar operations that require serialization can often be cached.
- **FIFOs as module interfaces**: One of the most tedious and difficult problems in building a complex hardware system that relies on results from various complex modules is coordinating when a result is available from one module and when it is consumed by another. Variable packet sizes and variable field lengths only make it even more complicated. To simplify the job, we used shallow FIFOs (queues) at the interfaces between modules, making sure that these FIFOs use flow control all the way to the input. While such a simplification comes at the cost of additional logic area, it significantly reduces development time.
- **Aligning fields to data path width**: Again this is a generally-applied principle: IP, for example, is aligned to 32-bits. However, when designing a new packet format, the designer has considerable leeway, and should keep alignment in mind as it significantly reduces hardware complexity.
- **Speculation**: Since cryptographic operations usually take many cycles to complete, they should all be started as soon as possible by assuming that a packet will pass all checks. A packet that fails a check should only be dropped at the end of its processing to ensure that the FIFOs between modules all remain synchronized (we would not want a stray result from a dropped packet to be left in a FIFO).
- **Reducing the number of cryptographic primitives**: The smaller the number of basic cryptographic primitives, the easier it is to find the design's bottleneck and provision for it. For instance, our ICING-PVM forwarder only uses AES-128 as the base pseudorandom function (PRF).
- **Considering replication early**: Many cryptographic operations are not pipelineable (e.g., CHI in our design). Ensure that it is easy to replicate these modules. This will come in handy when working on meeting design timing or ensuring that

the design fits in an FPGA.

The costs of furthering the end-to-end principle The ICING project attempts to move more functionality such as interdomain path selection and network access authorization from the network to the end-points. Doing so has required some heavy machinery to be built into the network to ensure that the policies of all network stakeholders concerned are respected, namely to ensure Path Compliance.

Moving a function (such as path selection) from the network to end-hosts often has security implications that would need to be addressed, as was needed for ICING and hinted by a survey of earlier work (Chapter 5). Addressing these security implications (such as the possibility of violating service providers' transit policies) is often only possible by replacing the removed function with new mechanisms *in* the network (for example, to enforce Path Compliance). So even though the end-to-end principle is usually interpreted as arguing for less functionality within the network, moving some function from the network to end-hosts can result in a more expensive network, with cost depending on the importance of the function and the strength of any new security guarantees.

Part II

Expedient

Chapter 7

Expedient

7.1 Introduction

Up until recently, businesses that needed computing resources had to build datacenters provisioned for these businesses' peak computing demands. Startup businesses with fast growth potential needed to put a large capital at risk in building datacenters for growth that might not be realized. However, as Armburst et al have noted [36], computing as a utility has recently become more than just a long-held dream [130] and statistical sharing of computing resources is now a reality. Commercial Infrastructure-as-a-Service (IaaS) offerings such as Amazon's Web Services (AWS) [27], Microsoft's Azure [9], and Google's App Engine [3] enable businesses to provision and tear down provisions for computing resources as quickly as minutes, allowing them to more closely match supply to demand and only pay for used computer cycles.

Similarly, researchers have created their own computing IaaS systems. Some examples of these systems are PlanetLab [133], Emulab [164], and TeraGrid [24]. These deployments enable researchers to do compute-intensive research such as protein folding [104] or geological modeling [19] or to deploy publicly accessible experimental systems such as CDNs [78] or DNS-replacements [137].

But for some applications, computing resources are not sufficient. These applications might want to control the network between computing resources or the network connecting users to services. Some applications might want to obtain high bandwidth connectivity only when they need to transfer large bursts of data. Such connectivity can be used to deliver large volumes of data from remote monitoring stations such as radio telescopes to local computing infrastructure for processing [4], share large volumes of data among distributed teams [6], or experiment with new collaborative environments [16, 110]. Other applications might want to control the network topology. For example, map-reduce applications usually have all-to-all communication patterns, suggesting that a fully-connected network mesh would be more efficient. On the other hand, multicast applications usually have one-to-all communication patterns, suggesting that tree network topologies might be more efficient. Yet other applications might want to control the network's routing protocol. For example, they might want to send real-time video traffic through multiple disjoint network paths to decrease end-to-end packet loss probability and increase video quality. Finally, researchers or developers might want to experiment with new network architectures that enable in-network services and require in-network processing such as ICING-ON (Chapter 3) or ICING-L3 (Chapter 4).

Very often, for such applications to be useful, they need to be deployed at scale, requiring a widely distributed network infrastructure to connect users and services. That is, this infrastructure needs to act as a substitute for the Internet, at least for the application's traffic. Unfortunately, no one project has the capability or resources to deploy such infrastructure for an application. So, to meet these needs, developers and researchers turned to overlay networks [61]. But overlays are usually deployed at Internet edges, thus suffering from large latencies, or they have no knowledge of the underlying network's actual topology.

For these reasons, some projects have attempted to develop network IaaS. For example, Internet2's ION service [5] enables researchers to request high-bandwidth dedicated virtual circuits through Internet2's backbone, and have them allocated within minutes. VINI [42] and the Supercharged PlanetLab Platform [155] provide programmable nodes that are connected by isolated high bandwidth circuits to create new network architectures, provide in-network services, or experiment with routing

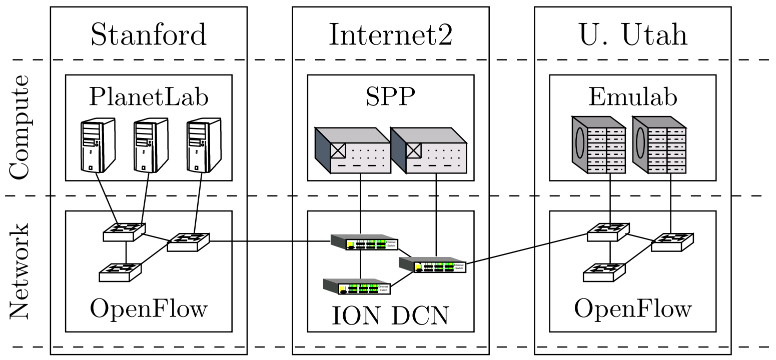

Figure 7.1: Example of heterogeneous IaaS that GENI enables. SPP is Supercharged Planetlab Platform [155]. GENI allows users to deploy an application or experiment across IaaS providers and across resources types.

protocols. And, more recently, Flowvisor [147] was suggested as a way to create isolated slices in production OpenFlow networks. Each such slice can then be controlled by an application or experiment, without the risk of affecting production traffic.

Given all these available heterogeneous resources, a natural question arises: How can a researcher manage the resource provisions—collected in an infrastructure *slice*—across a number of such IaaS offerings for a single application or experiment? For example, a CDN business might want to have a set of virtual machines distributed across the US, interconnected through a particular network topology. Or a researcher might want to experiment with a new network architecture that provides applications with new network-level services (like those provided by ICING-ON and ICING-L3 in Chapters 3 and 4). The application thus might require seamless *vertical* integration between different resource types (compute and network), and *horizontal* integration to (1) manage computing resources across various providers and (2) provide continuous end-to-end network connectivity and control. Figure 7.1 shows an example of such infrastructure.

Providing such a management capability has technical and security challenges. Technically, it is not clear how a user can create and manage allocations across heterogeneous resources to provide a coherent infrastructure for her application. First, each type of resource has its particular interface for creating and managing reservations and then to configure and use these reservations. For example, a VLAN reservation across Ethernet switches is different from a circuit reservation across Internet2 or a virtual server reservation on PlanetLab. Second, connectivity between different resources across IaaS providers is not necessarily visible to the IaaS itself. For example, a PlanetLab node may be connected directly to an Internet2 ION switch, but neither PlanetLab's nor Internet2's provisioning infrastructure knows about this connectivity. This connectivity must be exposed to the user so she can know how to provision network resources. Finally, overcoming both of these challenges cannot come at the expense of the user interface for managing the infrastructure slice. The user must be able to create and manage her slice using rich user interfaces. These interfaces should enable the user to explore the wide variety of available resources and their network connectivity, select available resources, and create end-to-end fully connected provisions interactively or programmatically.

From a security standpoint, the individual IaaS providers need to be able to enforce their own usage policies and hold a user accountable for how resources get used (either for billing or to protect against usage policy violations). On the other hand, users do not want to manage multiple accounts across the different IaaS providers and request authorization for resource access from each provider separately.

Overcoming these challenges in networking research is a prerequisite for GENI's success. GENI, the Global Environment for Network Innovations, was started by the National Science Foundation to promote networking research. The NSF realized that even though the Internet began as an academic research experiment, today, it has become critical infrastructure with billion-dollar businesses counting on its availability. Unfortunately, reliability is the antithesis of experimentation. Thus, it has become almost impossible for networking researchers to deploy and experiment with new protocols and network architectures on top of or to replace the existing Internet. To overcome this hurdle for innovation, the NSF started GENI.

GENI's ultimate vision is that of a researcher being able to deploy an experiment—an experimental application, service, network architecture, routing protocol, etc.—over a globally distributed programmable infrastructure. This deployment would be into a *slice* of the infrastructure that is isolated from other slices and incapable of disrupting the experiments running in them. Critically, GENI must enable external end-users to use these experiments. For example, a researcher may deploy a network-level load-balancing service along with CDN servers in a GENI slice and attract end-users to this service. Experimental services may then "graduate" and be deployed over the Internet, or they may remain permanently deployed in GENI's infrastructure.

There are many ways to build such an infrastructure. One approach is to start from scratch—to design and deploy special computing resources and a new virtualized network. Alternatively, the system could be built by combining existing pieces already deployed. GENI takes the latter approach, and aims to integrate heterogeneous computing and network resources, whether they exist today or emerge in the future. Figure 7.1 shows how GENI will have several types of resources distributed across several providers.

This chapter is about the management framework for such an infrastructure. It is about enabling users to create coherent infrastructure slices in which they can deploy their applications across (1) resource types and (2) GENI's IaaS providers. When designing, building, and deploying a management framework for existing and future computing and networking resources, there are many factors to consider, such as:

- **Ownership and Control**: Who will control and maintain each of the components? In PlanetLab, for instance, each server is contributed and maintained by some campus, but they are managed centrally from PlanetLab Central (PLC)—PlanetLab's central authority [133]. Therefore, the entire set of PlanetLab machines is a single IaaS, managed by one provider, PLC. On the other hand, an OpenFlow network can be the production network on a campus, controlled by a local controller (e.g., [85]). It is therefore natural for each campus to be its own OpenFlow IaaS provider, contributing a fraction of the campus network resources to GENI.

- **Policy**: If some resources are managed by each campus, then they will have locally defined policies in place already (e.g. local security, access control, and regulatory). The management framework needs to allow each IaaS provider to define the policies best suited to the resources they are contributing.

- **Trust**: There are many principals involved in GENI: the IaaS provider; the operator of a campus network; an experimenter; and a user who wishes to join an experiment or use a GENI-deployed application. In a heterogeneous environment, what can we assume about the trust between the various principals? Should the trust be centralized (i.e. everyone trusts some central component) or should it be distributed? Should an experimenter establish an account with a central registry, or should she do so with every aggregate?

- **APIs**: It has proven very challenging to design a uniform API for a management framework to manage all of the contributed aggregates. Different aggregates are managed in different ways depending on the types of resources they contain and depending on who owns them. The design of standardized APIs has led to much discussion and debate in the GENI community: there is a tension between the simplicity arising from defining a single uniform rigid API and the flexibility of allowing each IaaS-type to define its own.

The first two factors (Ownership and Control; and Policy) are operational. They dictate which components can be used and how a slice is spread across components in the system. If we assume a central authority that can manage a fairly powerful central component (similar to PlanetLab), then an IaaS deployment can be managed by a simple centralized database or registry and a Web interface. If, on the other hand, we cannot assume a central entity, then the IaaS providers will all need to communicate with each other and with the experimenter to stitch together the slice.

The third factor (Trust) is about security and reliability. It decides how authentication and authorization takes place and which components enforce them. As an example, owners of PlanetLab nodes cede authentication and authorization control over their nodes to PLC *completely*. They do not control who uses their nodes or how. If at any point a node's owner is not satisfied with the way the node is being used, the owner can complain to PLC or simply turn the node off. This trust model simplifies PLC's job and reduces the burden of administering the nodes. On the other hand, a distributed trust model can enable much more flexible policies but may require duplication of effort at each provider.

The fourth factor (APIs) determines development and maintenance effort. A standard uniform API relieves management tools' developers from learning a new API for each resource type. Indeed, there have been several attempts at creating such an API (e.g., SFA [21] and ORCA [55]). These attempts have succeeded at creating a federated authentication and authorization mechanism. Unfortunately, however, these attempts have resulted in incomplete restrictive APIs that side-step the two main challenges in developing a standardized provisioning API: (1) providing a generic description of all types of resources and (2) providing an API that can be used to manage all these types of resources. Instead, these attempts have only enabled a simple generic API and have left resource and reservation descriptions to resource-specific XML strings provided as arguments to the API calls. By doing so, not only have these attempts limited the way management tools and user interfaces can interact with the IaaS provider (i.e., through the simple API), but also pushed all the complexity of understanding resource-specific descriptions back to the user interface. Using resource-specific XML descriptions effectively creates an implicit resource-specific API that is inconvenient for user interface developers (as we have learned ourselves trying to use SFA to create slices in OpenFlow networks). Finally, it is unclear where information on connectivity between IaaS providers would be stored or where it would come from.

In general, the GENI community is converging on a management framework with no centrally owned component, limited policy flexibility, a strongly distributed trust model, and a partial standard API for all IaaS types and providers. This framework is provided by SFA. It is not clear that these choices will be the correct choices for the future.

In this chapter, we explore an alternative design, Expedient. Instead of taking a decisive stand on these issues, Expedient allows most initial design choices to evolve. As with any new complex system, the best way to reach the correct design is through rapid iterative prototyping. Expedient provides this capability. It does so by following two principles:

1. **Reuse**: Whenever possible, Expedient leverages existing technologies to reduce development effort, simplify maintenance, and smooth any learning curves.

2. **Pluggability**: A pluggable system enables new functionality to be added and removed without requiring the rest of the system to change.

Expedient is modeled after common travel websites that allow a user to reserve airline tickets, hotel accomodation, and car rentals for a single trip from one portal. Similarly, Expedient allows a researcher to provision the end-to-end infrastructure for an experiment: it allows the researcher to reserve computing resources and provision the network between them, possibly across IaaS providers, using a single set of credentials. Expedient has the following initial set of requirements:

- Enable a rich communication model between IaaS providers and user interfaces;
- Expose inter-IaaS relationships to users;
- Protect registered IaaS providers from unauthorized access; and,
- Enable delegation and team management.

To meet these requirements and provide a working management platform prototype with minimal development effort, Expedient makes the following initial design choices:

- **Centralized management**: Initially, a centralized Expedient instance is deployed for managing all the GENI infrastructure. With additional plugins, this centralization can be relaxed so that multiple Expedient instances can be federated.
- **Limited centralized trust**: IaaS providers trust Expedient to perform user authentication and authorization, but not completely. Each provider still enforces its own usage policies locally. In fact, Expedient, for the most part, simply acts as an authentication and authorization proxy. The extent to which trust is placed at Expedient depends on how IaaS connector plugins are implemented, and thus can be changed later.
- **Heterogeneous APIs**: Expedient does not require IaaS provisioning and management APIs to change. Each user interface will need to understand the idiosyncracies of the resources it handles. We believe that such a consequence is inevitable. Different resource types will have different properties and an effective user interface will need to understand these resource-specific properties. For instance, a user interface cannot respresent and provision virtual machines the same

way it represents and provisions virtual circuits. At least, by accepting heterogeneous APIs and making them explicit they would be more convenient to work with. And, since these APIs are within plugins, Expedient can also work if a standard uniform API becomes the norm.

To leverage the maturity of Web technologies, Expedient is built as a three-tiered website. The front-end serves a mix of static and dynamically created content, the middle tier processes and creates dynamic content, and the backend is a relational database. Having the management platform as a website eliminates the need for experimenters to download specialized software (they can manage their slices using a web browser), decreases development and maintenance costs since many of the functions needed for Expedient already exist in Web frameworks such as Django [8] (on which Expedient is based), and allows transparent upgrades with new features or functionality without the experimenter worrying about software versioning or compatibility (improving the ability to rapidly prototype new features or design choices).

Some back-of-the-envelope performance calculations suggest this approach is feasible, even if GENI grows to be quite large. Using PlanetLab as an example: PlanetLab currently has around 700 experiments using approximately 1,000 global nodes [15]. If we assume that GENI is approximately 100 times bigger than PlanetLab, then it might need to support about 70k experiments and 100k different resources. Let's assume each experiment is modified on average once per hour. That would be 70k req/hr or 20 req/s, certainly within the capabilities of current webservers [162, 163], and likely within the capabilities of a relational database such as MySQL with 100k rows (Facebook for instance uses MySQL and has a peak load of 13M queries/s [70, 71]). While these numbers need to be validated with a deployment and measurements that can only be obtained after a few years of use, they hint that a website-oriented implementation is feasible. Furthermore, given the large investment in webservers, such a design can ride on the coat-tails of general improvements in Web and database systems.

In the next section (Section 7.2), we give a quick overview of Expedient's components. We describe Expedient's design in Section 7.3, describe how we use Expedient to create end-to-end slices connecting PlanetLab nodes using OpenFlow networks in

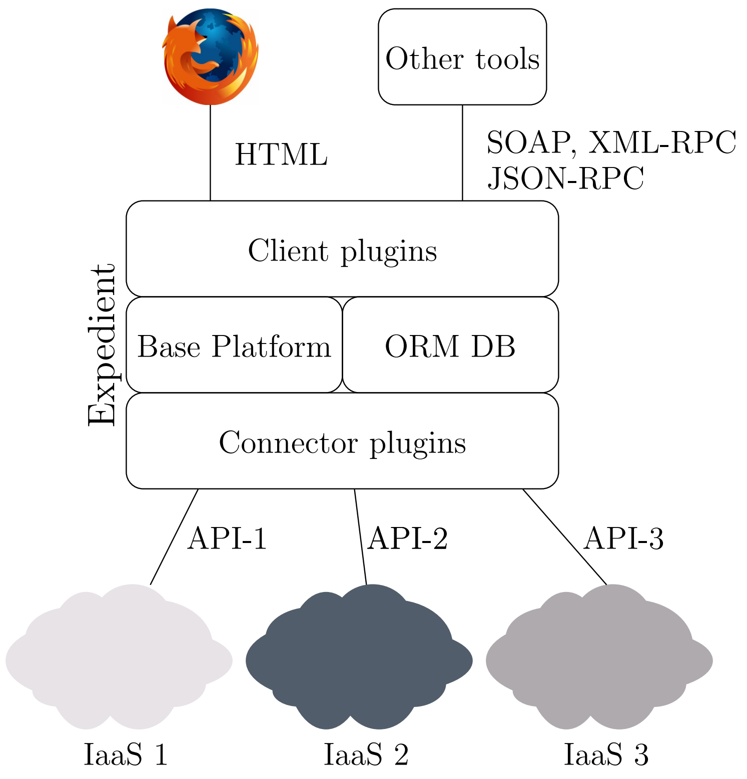

Figure 7.2: Expedient consists of four subsystems: Clients interact with the user through various tools, the base platform provides the basic functionality, the ORM DB store object data in a relational database, and connectors enable all other subsystems to communicate with IaaS providers using IaaS-specific APIs.

Section 7.4, and go over existing solutions in Section 7.5. We conclude the chapter with a discussion of insights learned along the course of Expedient's implementation and deployment and future directions (Section 7.6).

7.2 Overview

Expedient (Figure 7.2) consists of four subsystems: an object relational mapping (ORM) database, a base platform subsystem, and two types of plugin—IaaS connectors and user clients. Connectors and clients cooperate to provide the user interface for creating and managing slices across IaaS providers.

The ORM database provides persistent storage of objects into a relational database and abstracts the underlying SQL through an object-level API. Objects can then be queried and retrieved from the database using relational queries, at the object abstraction level, not at the SQL level. Once retrieved, the objects are available along with class-specific methods. ORMs make it easy to build complex hierarchical models of resources, slices, and IaaS providers.

Expedient's base platform provides basic ORM classes and basic functionality for user management, authentication, authorization, messaging, slice management, and IaaS management. This base functionality is extended by plugins: IaaS connectors and user clients. Plugins extend Expedient's base ORM classes and add new ones.

IaaS connectors extend Expedient's base IaaS provider model. They add provider-specific APIs, provider-specific data fields, and the capability to communicate with the IaaS provider. Connectors expose these new fields and APIs to the platform and to all other plugins. Connectors also describe inter-IaaS relationships and connectivity and are responsible for keeping state in Expedient synchronized with state at the providers.

Clients expose either an API for the consumption of an external tool or a user interface that a user can directly use to manage her provisions. Clients consume the APIs exposed by connectors, and can specialize in particular resource types. For example, a client can show a network topology from which the user can select resources, or it can provide an API, which is consumed by tools external to Expedient.

7.3 Design details

Here we discuss how each of Expedient's four subsystems—the base platform, the ORM, connectors, and clients—helps meet Expedient's requirements.

7.3.1 Base platform

The base platform has several cooperating modules: the authentication module, the authorization module, and other modules that provide messaging capabilities, user management, IaaS provider registration, and other support functions. The base platform also provides the base user interface (UI) that allows users to manage their profiles, register new IaaS providers, and create and manage teams and slices. Below we discuss only authentication and authorization.

Authentication

Expedient's authentication module accepts trusted client SSL certificates and username/password logins. Expedient relies on much of Django's authentication subsystem, but implements its own authorization. Expedient accepts client certificates signed by trusted third-party certificate authorities, allowing Expedient's user repository to be federated. Any user can create a login at Expedient and obtain a user certificate, but actions within Expedient need to be authorized.

Authorization

Expedient implements a fine-grained extensible authorization system to protect IaaS providers from unauthorized access and to protect users information from other potentially malicious users. Expedient's authorization system also helps with federating trust even when deployed centrally.

Authorization is done through Expedient's permission system. Permissions in Expedient can protect many actions within the platform: execution of object methods, URL access, or calling of static functions. To protect an action, the developer specifies a permission check with the following symantics: some entity—the *permission owner*—within Expedient must have some permission to execute the protected action on some object—the *permission target*. The permission target may be an object instance, class, or function, and the permission owner can be any object instance. For example, deleting a slice is protected by permission `can_delete_slice`. The developer informs the permission system that the permission owner is the logged-in user who initiated the delete request, and the target is the slice to be deleted. Thus the logged-in user must have the permission `can_delete_slice` for the slice she is about to delete. Similarly, a slice needs to have the permission `can_use_iaas` for a registered IaaS provider before the slice can access that IaaS provider's resources.

Expedient allows developers to redirect unauthorized users to a developer-specified page. On that page, a developer may ask an unauthorized user for additional information before giving the user access. For example, if an unauthorized user tries to delete a slice, the user gets redirected to a page where the user can request authorization from other users who can give the user permission.

Besides allowing users to *give* each other permissions, Expedient allows users to *delegate* permissions. A given permission cannot be further given to other users; on the other hand, a delegated permission can. Delegation allows Expedient to have more than one manager, each with different authority. So even in a centralized Expedient deployment, it is not necessary to have a centralized point of authority.

Expedient uses *projects* as the unit of team and slice management: slices are created within projects by project members. To create a project, a user must have the `can_create_project` permission for the `Project` class. Before a slice in a project can use an IaaS provider, the project and the logged-in user, like the slice, must have the `can_use_iaas` permission for that IaaS provider. Project members can have multiple roles, such as owner, researcher, technician, etc. A role describes a set of permissions that can be given together to a member. Roles are project-specific, so different projects can create and modify their own roles.

7.3.2 ORM database

The ORM database provides the base classes that plugins extend. Expedient defines the following ORM classes:

- `User`: an Expedient user.
- `IaaS`: the base IaaS provider class. Connectors must extend this class.
- `Resource`: the base IaaS resource class. Connectors extend this class to describe IaaS-specific resource types (e.g., PlanetLab node, OpenFlow switch, etc.).
- `Sliver`: the base class that describes a slice of a resource within an IaaS provider. Connectors extend this class to describe resource provisioning. For example, a virtual machine sliver might describe the CPU percentage and memory allocated

to the virtual machine.

- `Project`: an Expedient project.
- `Slice`: an Expedient slice. A slice is a container of slivers from across various IaaS providers.

7.3.3 Connectors

Connectors enable Expedient to understand new types of IaaS. So far, we have three connectors: a PlanetLab connector and two OpenFlow connectors. The two OpenFlow connectors use two different APIs to connect to IaaS providers. One API uses the GENI API [2] while the other uses the OpenFlow slicing API (Section 7.4).

A connector extends the `IaaS` class with additional data fields and methods for a particular IaaS type. It also overrides the base class's methods for creating and deleting a slice. The connector can create and require new permissions and provide hooks for requesting more information from users before authorizing actions. Each connector can have its own API for interacting with the IaaS. The API that it must override in Expedient is minimal and only imposes that the connector be able to start and stop a slice. All other functionality should be exposed to users through client plugins.

7.3.4 Clients

Clients add new ways for users to interact with resources and IaaS providers. They consume the additional API exposed through IaaS classes and can create slivers or modify existing slivers. We assume that clients are not malicious but may be slightly buggy. As long as they do not attempt to subvert the permission system, IaaS providers are protected against unauthorized access by buggy clients.

Our implementation currently has two clients. The first is a javascript GUI that shows the PlanetLab and OpenFlow network topology of selected IaaS providers and allows users to select a set of servers and switches for their slice. This GUI also creates SSH keys that the user can download and use to access PlanetLab nodes. The second client allows users to download and upload XML descriptions (*RSpec*s) of OpenFlow

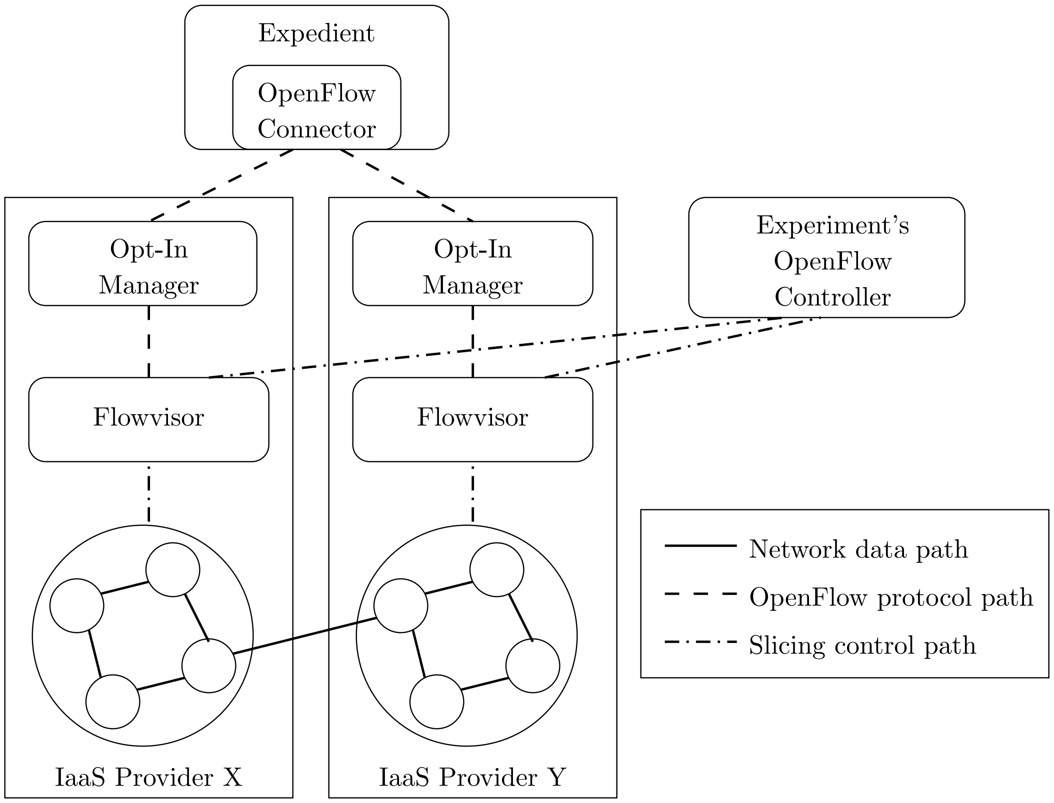

Figure 7.3: The GENI OpenFlow stack consists of Expedient with the OpenFlow connector, per participating provider Opt-In Manager and Flowvisor, and a production OpenFlow network. The experimenter's controller connects to the Flowvisor using the OpenFlow protocol.

resources for their slices.

7.4 Creating end-to-end slices

For GENI, we wanted to demonstrate how to create a complete slice across multiple IaaS providers and types. We wanted the slice to consist of both computing and networking resources distributed across providers, with the network providing end-to-end continuous connectivity.

For the computing resources, we chose MyPLC, a package that allows installation of private PlanetLab systems. Network connectivity is trickier. How can a network be sliced so as to allow multiple slices to control production switches across several providers? How do we enable providers, university campuses in this case, to apply their policies on the traffic and protect their internal traffic from being controlled in a slice? Finally, how do we enable end-users at these campuses to participate in experiments by "opting in" their traffic?

Here we describe how OpenFlow allows us to build the sliceable network infrastructure needed to allow experiments to run next to production traffic, and to enable end-users to opt into experimental slices. We describe how Expedient enables researchers to create such end-to-end experimental slices.

7.4.1 Slicing an OpenFlow network

The OpenFlow slicing stack consists of four layers (Figure 7.3): OpenFlow switches at the bottom, the Flowvisor, the Opt-In Manager, and the OpenFlow Expedient connector on the top.

Sherwood *et al.* describe in [147] how to slice an OpenFlow network using FlowVisor. At a high level, Flowvisor acts as a proxy between OpenFlow switches and OpenFlow controllers for the slices. OpenFlow switches connect to Flowvisor as they would to any other controller, and Flowvisor connects to slice controllers pretending to be the underlying switches. Flowvisor monitors OpenFlow protocol messages from the controllers and ensures that each slice controller only operates on traffic within its slice. OpenFlow slices are defined by the header values they can operate on. Flowvisor's job is to isolate slices from each other and ensure that no flow is controlled by more than one slice.

The job of the Expedient connector and the Opt-In Manager is to allow researchers to create experiments that end-users at the IaaS providers can join or *opt into.* By joining an experiment, a slice of the end-user's traffic will be controlled by the experimenter's OpenFlow controller. That slice of traffic depends on the type of traffic that the experiment is concerned with. For example, a Web traffic load-balancing experiment may operate on port 80 traffic, while a new routing protocol experiment may operate on all IP traffic.

The Expedient connector enables a researcher to select the OpenFlow switch topology that the researcher would like to control. The connector also requires the researcher to specify the type of traffic with which the experiment will be concerned using OpenFlow flow header fields. For example, the experiment may be concerned with IP traffic or just traffic on a particular port.

Multiple slices may request the same traffic type, but none of the slices will get any traffic before flows are opted into into the slice. Owners of traffic at each provider will need to explicitly join an experiment before their traffic is placed in that experiment's slice. The Opt-In Manager's job is to find the particular flows that should go into an experiment's slice once the owner of that traffic joins the experiment.

The Opt-In Manager translates provisioning requests from the connector to Flowvisor configuration commands. A slice creation request at Expedient does not automatically place any traffic under the researcher's control. A user can log into the Opt-In Manager and request to join an experiment. The Opt-In Manager has information on which user owns which traffic. It intersects the header values of the user's traffic with the experiment's requested traffic to find the user's flows that should be controlled by the experiment. A user can join multiple experiments and thus have different slices of her traffic in different experiments.

What happens when a flow in a slice at one IaaS provider crosses over to another IaaS provider's OpenFlow network? Does it stay in the same slice? Not necessarily. This question concerns local security policy. An administrator will want to protect internal traffic and ensure that only flows whose local owner joins an experiment can be controlled by that experiment. For example, a flow sent from one user in one network to another user in another network should not necessarily belong to the same slice: the sender and the receiver might have joined different experiments. Thus, opt-in requests are local to an IaaS provider. If the experimenter wishes to keep the flow originating from one IaaS provider's network in the same slice at a remote IaaS provider's network, the experimenter needs to be able to opt that same flow into her slice at the remote IaaS provider. But the remote Opt-In Manager might not know who the experimenter is. Instead, the remote Opt-In Manager can delegate control of unowned flows in its network to Expedient. If the flow is one of those unowned flows, Expedient can then opt in the flow in question at the remote Opt-In Manager. Otherwise, the remote owner of that flow will need to join the experiment.

7.4.2 Connecting PlanetLab nodes

Expedient has a connector that implements the GENI API [2] to communicate with MyPLC deployments. The GENI API connector allows researchers to create slices at PlanetLab nodes in connected MyPLC IaaS providers.

The OpenFlow connector allows an administrator to describe the connectivity between OpenFlow switches and PlanetLab nodes and between OpenFlow switches across IaaS providers. Researchers can then create end-to-end slices that contain PlanetLab slices at different providers connected by a continuous OpenFlow network stretching between all the PlanetLab nodes.

7.5 Related Work

Much work has been done on GENI control frameworks. A GENI control framework aims to develop a unified API to which IaaS providers adhere. Work such as the Slice Facility Architecture (SFA [21]) builds on SSL authentication and certificates, using Uniform Resource Names (URNs) [148] as global unique identifiers. Under SFA, each IaaS provider, called a *resource aggregate* trusts a number of *clearinghouses*. Resource aggregates expose an XML-RPC API that is accessed by users through a local client. Clearinghouses create user certificates for authentication and signed credentials for authorization that allow a user to execute the XML-RPC calls. The API is minimal, in that it requires resource aggregates to implement only a few types of calls. It pushes the complexity of communicating with different resource aggregate types into an XML document, the *RSpec*, which is provided as a parameter in the API calls. For example, the `CreateSlice` call takes as argument an RSpec that describes what resources are in the slice.

SFA provides a robust scalable way to federate IaaS providers without a single point of authority and without online authentication (certificates can be verified offline). However, SFA suffers from a number of drawbacks. First, the API constrains the mode of interaction between the client and the provider. For example, the API does not allow two-stage commits for slice creation and does not allow additional resource-specific APIs to be exposed to clients. Second, SFA includes the use of tickets to signify available resources, but they are an unnecessary complication that rarely have useful functionality. Third, it is not clear that transportation of complete XML descriptions of the IaaS provider and of a slice is scalable or necessary. Finally, certificate management under SFA places a large burden on IaaS providers, clearinghouses, and users who will need to ensure that certificates are unexpired, that only trusted certificates are used, and that revocations are checked. From a practical standpoint, working with non-standard SSL certificates is not an easy task due to the limited and complex SSL support in standard development libraries.

In contrast with SFA, Expedient does not describe how IaaS providers communicate with external entities nor how they implement authentication and authorization, and thus does not require changes to the IaaS's control infrastructure. Expedient can use SFA as a connector; in fact, it has a connector for the GENI API [2], which is a simplified subset of SFA.

ORCA [55], the Open Resouce Control Architecture, like SFA, attempts to provide a "narrow waist" API through which resource provisioning takes place. ORCA attempts to build a more complete resource provisioning system, where users run controllers that can negotiate resource leases with the underlying infrastructure to obtain resources and control the experiment's scaling. Like SFA, ORCA places constraints on the API between an IaaS provider and the control framework. And although it is highly sophisticated, its complexity may be overwhelming for simpler tasks. Finally, it is not clear that it provides the right level of abstraction needed for a GENI control framework.

Others have also realized the utility of aggregating resources across several IaaS providers, concentrating on just computing resources. SimpleCloud [20] is building an open standard API based on the services offered by AWS, Rackspace, and Azure. Applications will be able to use the API across several providers, thus enabling the application to control its own scaling. However, the API will only be available for PHP applications.

The Distributed Management Task Force (DMTF), the same body that has published the Open Virtualization Format, currently a de facto standard for virtual machine specification, is working on its own API [7]. This API, part of the Open Cloud Standards Incubator, will enable enterprise administrators to manage and audit provisions across several IaaS providers. This API is still under development, and concentrates on commercial computing IaaS offering.

The Open Cloud Computing Interface Working Group (OCCI-WG) has also developed its own API [11] for management of compute resources and services across IaaS providers.

All these proposed APIs require changes at the providers themselves, and may be difficult to implement. Additionally, they only concentrate on computing resources, and are thus not suitable for GENI.

Finally, Yan *et al.* [171] have a system that is conceptually similar to ours, but much more limited in scope. They provide a Web application that is capable of managing virtual machine provisions across several IaaS providers. Like Expedient, they have connectors that describe how to communicate with the provider using the provider's API. However, unlike Expedient, their proxy does not enable new UIs through additional clients and cannot handle heterogeneous resources.

7.6 Discussion

7.6.1 Summary

Expedient attempts to make the jobs of three stakeholders in the IaaS easier: the developers who build the IaaS, the users of the IaaS, and the providers of the IaaS.

For IaaS developers, Expedient does not require a standard API to all connected providers. Thus it relieves developers from the burden of creating this API and then integrating it into their existing systems. It does not require them to update their systems anytime the API changes, and it does not force any particular communication model between the IaaS provider and the IaaS users. Expedient also provides developers with a platform over which they can have rich communication models and a relational database that can be queried for resources. Using these capabilities, developers can build rich UIs and connect them as client plugins.

For IaaS users, Expedient provides all the benefits of a Web application. It does not require downloading specific tools and keeping them up to date (except for a Web browser). It provides users with ubiquitous access and relieves them from having to store their data locally and keeping it backed up. Expedient relieves users from having to manage multiple IaaS provider accounts and provides seamless integration across resource types and provider trust domains.

For IaaS providers, Expedient is transparent. It does not require IaaS providers to treat Expedient users differently from regular users, and does not require them to change their authentication and authorization mechanisms. Their existing users will remain unaffected. It also allows providers to keep their local security policies and enables them to enforce more policies remotely, at Expedient (subject to the IaaS connector's capabilities).

7.6.2 Future work

Expedient is far from being perfect and the final solution to GENI's problems. Expedient needs additional intermediate layers of abstraction to make plugin development easier. And, of course, Expedient will benefit from additional connectors and clients.

The Expedient OpenFlow connector's implementation currently does not have a way to request that traffic be opted into a particular experiment at a remote Opt-In Manager. This functionality will become even more useful when a researcher wishes to reserve a virtual machine and its network traffic simultaneously at Expedient.

Expedient's current authorization system does not have a clean API for connectors and clients. Newer versions of Expedient should simplify the API and make it easier to extend the authorization system.

Expedient is currently being deployed at several campuses around the US as part of the OpenFlow network slicing stack. Additionally, the OFELIA project [12] is developing Expedient and modifying it to suit their particular needs.

7.6.3 Lessons learned

Developing and deploying Expedient has provided much insight into practical issues. Below are a few.

On the Web: Expedient exposed me to the rich world of Web development. I now believe that everyone should build a simple Web application at least once in their lives. The power that Web frameworks make available is uncanny, particularly when combined with "Cloud" services such as Amazon's AWS. However, the Web is rife with security concerns. Too much trust is placed on the developer to be security-conscious when developing a Web application. Future frameworks should protect against security holes due to bugs, and should separate security policy from the rest of the Web application in an architecture that is more akin to a Model/View/Controller/Policy architecture.

On abstraction: Abstraction when building a pluggable Web application is very important. Unfortunately, Expedient's current implementation does not have enough abstraction layers. This problem should be addressed in future versions of Expedient.

On the GENI process: Design by committee will never work well, and a working design will never be accepted by a committee. This is a frustration shared by many in the GENI developer community. The main problem with GENI is the emphasis on pre-meditated design and complete coverage of details, with less importance placed on iterative rapid prototyping and improvements through concise and specific requirements. The GENI community needs to choose one system, and iteratively work on it to get it to an acceptable state rather than developing a large number of systems simultaneously and attempting to combine them.

On centralization: It has become evident during the course of developing and deploying Expedient that there is a strong resistance in the academic community against centralized systems. But centralized trust is only important when that central authority has too much *unchecked* power. For instance, IANA is a centralized entity that has to be trusted. But this fact has only become an issue when the Internet became such a global success, and this problem is now being dealt with. On the other hand, centralization can be a boon to rapid development and deployment. This strong bias against centralization needs to be revisited in light of the complexity and size of today's systems and the importance placed on rapid development cycles.

Bibliography

[1] The 32-bit autonomous system number report. http://www.potaroo.net/tools/asn32/index.html.

[2] The GENI API. http://groups.geni.net/geni/wiki/GeniApi.

[3] Google app engine. http://code.google.com/appengine/docs/whatisgoogleappengine.html.

[4] Internet2 applications overview: eVLBI over advanced networks. http://www.internet2.edu/pubs/200310-WIS-eVLBI.pdf. last accessed 3/5/2011.

[5] Internet2 ION service. http://www.internet2.edu/ion/. last accessed 3/5/2011.

[6] Internet2's Dynamic Circuit Network case study: The LIGO project at Syracuse University. http://www.internet2.edu/pubs/CS-DCN-LIGO.pdf. last accessed 3/5/2011.

[7] Interoperable clouds. http://www.dmtf.org/sites/default/files/standards/documents/DSP-IS0101_1.0.0.pdf.

[8] Meet django. http://www.djangoproject.com/.

[9] Microsoft Azure. http://www.microsoft.com/windowsazure/.

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

[11] Open cloud computing interface. http://occi-wg.org/.

[12] OpenFlow in Europe - linking infrastructure and applications. http://www.fp7-ofelia.eu/.

[13] The OpenFlow switch specification. http://OpenFlowSwitch.org.

[14] Packet traces from wide backbone. http://mawi.wide.ad.jp/mawi/samplepoint-F/2011/201101231400.html. Last accessed on 1/24/2011.

[15] PlanetLab projects. https://www.planet-lab.org/db/pub/slices.php.

[16] Press release – czech scientists demonstrating UltraGrid for HD video. http://www.ces.net/doc/press/2008/pr081029.html. last accessed 3/5/2011.

[17] Protecting your core: Infrastructure protection access control lists. http://www.cisco.com/en/US/tech/tk648/tk361/technologies_white_paper09186a00801a1a55.shtml. last accessed 2/20/2011.

[18] RouteScience PathControl. http://products.enterprisenetworkingplanet.com/networking/ia/RouteScience-Technologies-RouteScience-PathControl.html.

[19] Scec/cme terashake simulations. http://epicenter.usc.edu/cmeportal/TeraShake.html. last accessed 3/5/2011.

[20] Simple cloud API. http://simplecloud.org.

[21] Slice federation architecture. http://groups.geni.net/geni/wiki/SliceFedArch.

[22] SRX getting started - stateless firewall filters (ACLs) use case. http://kb.juniper.net/InfoCenter/index?page=content&id=KB16685. last accessed 2/20/2011.

[23] Sync SRAMs overview. http://www.cypress.com/?id=95&source=header. last accessed 2/28/2011.

[24] Teragrid. http://www.teragrid.com/. last accessed 3/5/2011.

[25] Digital signature standard (DSS). Federal Information Processing Standards Publication, November 2008. DRAFT FIPS PUB 186-3.

[26] William Aiello, John Ioannidis, and Patrick McDaniel. Origin authentication in interdomain routing. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[27] Amazon Web Services. `http://aws.amazon.com`.

[28] David Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable Internet protocol. In *SIGCOMM*, August 2008.

[29] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *SOSP*, October 2001.

[30] Tom Anderson, Timothy Roscoe, and David Wetherall. Preventing Internet denial-of-service with capabilities. In *HotNets*, November 2003.

[31] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS security introduction and requirements. RFC 4033, Network Working Group, Mar 2005.

[32] Katerina Argyraki and David R. Cheriton. Loose source routing as a mechanism for traffic policies. In *Proc. SIGCOMM Workshop on Future Directions in Network Architecture*, September 2004.

[33] Katerina Argyraki and David R. Cheriton. Active Internet traffic filtering: real-time response to denial-of-service attacks. In *USENIX Technical Conference*, 2005.

[34] Katerina Argyraki and David R. Cheriton. Network capabilities: The good, the bad and the ugly. In *HotNets*, November 2005.

[35] Katerina Argyraki, Petros Maniatis, David Cheriton, and Scott Shenker. Providing packet obituaries. In *HotNets*, November 2004.

[36] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[37] Ioannis Avramopoulos, Hisashi Kobayashi, Randolph Wang, and Arvind Krishnamurthy. Amendment to: Highly secure and efficient routing. February 2004. `http://www.cs.washington.edu/homes/arvind/papers/amendment.pdf`.

[38] Ioannis Avramopoulos, Hisashi Kobayashi, Randolph Wang, and Arvind Krishnamurthy. Highly secure and efficient routing. In *INFOCOM*, March 2004.

[39] Baruch Awerbuch, David Holmer, Cristina Nita-Rotaru, and Herbert Rubens. An on-demand secure routing protocol resilient to byzantine failures. September 2002.

[40] Hitesh Ballani, Yatin Chawathe, Sylvia Ratnasamy, Timothy Roscoe, and Scott Shenker. Off by default! In *HotNets*, November 2005.

[41] Boaz Barak, Sharon Goldberg, and David Xiao. Protocols and lower bounds for failure localization in the Internet. In *Proc. EUROCRYPT*, April 2008.

[42] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In vini veritas: realistic and controlled network experimentation. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM, pages 3–14, New York, NY, USA, 2006. ACM.

[43] Bobby Bhattacharjee, Ken Calvert, Jim Griffioen, Neil Spring, and James Sterbenz. FIND proposal. Postmodern internetwork architecture. `http://www.nets-find.net/Funded/PostModern.php`.

[44] John Black and Phillip Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *Proc. EUROCRYPT*, pages 384–397, 2002.

[45] Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. *SIAM J. of Computing*, 32(3):586–615, 2003. extended abstract in Crypto'01.

[46] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205, September 1997.

[47] Robert Braden, Ted Faber, and Mark Handley. From protocol stack to protocol heap – role-based architecture. In *HotNets*, October 2002.

[48] Anat Bremler-barr and Hanoch Levy. Spoofing prevention method. In *In Proc. IEEE INFOCOM*, pages 536–547, 2005.

[49] Kevin Butler, Toni Farley, Patrick Mcdaniel, and Jennifer Rexford. A survey of bgp security issues and solutions. Technical report, AT&T Labs - Research, Florham Park, NJ, 2004.

[50] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *NSDI*, May 2005.

[51] Ken Calvert, Jim Griffioen, and Leonid Poutievski. Separating routing and forwarding: A clean-slate network layer design. In *Proc. IEEE Broadnets*, September 2007.

[52] Martin Casado, Michael Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, August 2007.

[53] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A protection architecture for enterprise networks. In *USENIX SECURITY*, August 2006.

[54] I. Castineyra, N. Chiappa, and M. Steenstrup. The Nimrod routing architecture. RFC 1992, August 1996.

[55] Jeff Chase, Laura Grit, David Irwin, Varun Marupadi, Piyush Shivam, and Aydan Yumerefendi. Beyond virtual data centers: Toward an open resource control architecture. In *International Conference on the Virtual Computing Initiative (ICVCI)*, May 2007.

[56] E. Chen and Y. Rekhter. Outbound route filtering capability for BGP-4. RFC 5291, Network Working Group, August 2008.

[57] E. Chen and S. Sangli. Address-prefix-based outbound route filter for BGP-4. RFC 5292, Network Working Group, August 2008.

[58] Jerry Chou, Bill Lin, Subhabrata Sen, and Oliver Spatscheck. Proactive surge protection: a defense mechanism for bandwidth-based attacks. In *USENIX SECURITY*, July 2008.

[59] Cisco Systems, Inc. Cisco Catalyst 6500/Cisco 7600 Series Supervisor Engine 720 Datasheet. `http://www.cisco.com/en/US/prod/collateral/switches/ps5718/ps708/product_data_sheet09186a0080159856.pdf`.

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

[61] Dave Clark, Bill Lehr, Steve Bauer, Peyman Faratin, Rahul Sami, and John Wroclawsk. Overlay Networks and the future of the Internet. *Communication & Strategies*, 63, 2006.

[62] G. Adam Covington, Glenn Gibb, John W. Lockwood, and Nick McKeown. A packet generator on the NetFPGA platform. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2009.

[63] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[64] Colin Dixon, Thomas Anderson, and Arvind Krishnamurthy. Phalanx: Withstanding multimillion-node botnets. In *NSDI*, April 2008.

[65] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[66] 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.

[67] Deborah Estrin, Jeffrey Mogul, and Gene Tsudik. VISA protocols for controlling inter-organizational datagram flow. *IEEE JSAC*, 7(4), May 1989.

[68] Deborah Estrin, Yakov Rekhter, and Steven Hotz. Scalable inter-domain routing architecture. In *SIGCOMM*, August 1992.

[69] Deborah Estrin and Gene Tsudik. Security issues in policy routing. In *Proc. IEEE Symposium on Security and Privacy*, May 1989.

[70] Facebook MySQL tech talk - 11.2.10. `http://www.livestream.com/facebookevents/video?clipId=flv_cc08bf93-7013-41e3-81c9-bfc906ef8442`.

[71] Facebook at 13 million queries per second recommends: Minimize request variance. `http://highscalability.com/blog/2010/11/4/facebook-at-13-million-queries-per-second-recommends-minimiz.html`.

[72] A. Farrel, A. Ayyangar, and JP. Vasseur. Inter-domain MPLS and GMPLS traffic engineering – resource reservation protocol-traffic engineering (RSVP-TE) extensions. RFC 5151, February 2008.

[73] Nick G. Feamster. *Proactive Techniques for Correct and Predictable Internet Routing*. PhD thesis, M.I.T., September 2005.

[74] P. Ferguson and D.Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. RFC 2827, May 2000.

[75] P. Ferguson and D. Senie. Outbound route filtering capability for BGP-4. RFC 2267, Network Working Group, January 1998.

[76] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. RFC 2827, Network Working Group, May 2000.

[77] Paul Francis. *Addressing in Internetwork Protocols.* PhD thesis, University College London, London, UK, 1994.

[78] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *NSDI*, March 2004.

[79] Zhi (Judy) Fu and S. Felix Wu. Automatic generation of IPSec/VPN security policies in an intra-domain environment. In *12th Intl. Workshop on Distributed Systems: Operations and Management*, October 2001.

[80] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis. A framework for IP based virtual private networks, February 2000. RFC 2764.

[81] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *SIGCOMM*, August 2009.

[82] Sharon Goldberg, David Xiao, Eran Tromer, Boaz Barak, and Jennifer Rexford. Path-quality monitoring in the presence of adversaries. In *SIGMETRICS*, June 2008.

[83] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. *ACM CCR*, 35(5), October 2005.

[84] Adam Greenhalgh, Mark Handley, and Felipe Huici. Using routing and tunneling to combat DoS attacks. In *Proc. USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, July 2005.

[85] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martn Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM CCR*, 38(3):105–110, July 2008.

[86] Saikat Guha and Paul Francis. An end-middle-end approach to connection establishment. In *SIGCOMM*, August 2007.

[87] Krishna P. Gummadi, Harsha V. Madhyastha, Steven D. Gribble, Henry M. Levy, and David Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *OSDI*, December 2004.

[88] Mark Handley and Adam Greenhalgh. Steps towards a DoS-resistant Internet architecture. In *Proc. SIGCOMM Workshop on Future Directions in Network Architecture*, August 2004.

[89] Phil Hawkes and Cameron 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`.

[90] Yih-Chun Hu and Adrian Perrig. A survey of secure wireless ad hoc routing. *IEEE Security and Privacy Magazine*, 2:28–39, 2004.

[91] Yih-Chun Hu, Adrian Perrig, and Dave Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. In *MOBICOM*, September 2002.

[92] Yih-Chun Hu, Adrian Perrig, and Dave Johnson. Efficient security mechansims for routing protocols. In *NDSS*, February 2003.

[93] Yih-Chun Hu, Adrian Perrig, and Marvin Sirbu. SPV: Secure path vector routing for securing BGP. In *SIGCOMM*, September 2004.

[94] J. Ioannidis and S. M. Bellovin. Implementing pushback: Router-based defense against DDoS attacks. In *NDSS*, 2002.

[95] Cheng Jin, Haining Wang, and Kang G. Shin. Hop-count filtering: an effective defense against spoofed DDoS traffic. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[96] D. Johnson, Y. Hu, and D. Maltz. The Dynamic Source Routing protocol (DSR) for mobile ad hoc networks for IPv4. RFC 4728, Network Working Group, February 2007.

[97] David B. Johnson. Routing in ad hoc networks of mobile hosts. In *Proceedings Of The IEEE Workshop On Mobile Computing Systems And Applications*, pages 158–163, 1994.

[98] Jonathan Katz and Andrew Y. Lindell. Aggregate message authentication codes. In *Topics in Cryptology – CT-RSA*, volume 4964 of *Lecture Notes in Computer Science*, pages 155–169, April 2008.

[99] Hema Tahilramani Kaur, Andreas Weiss, Shifalika Kanwar, Shivkumar Kalyanaraman, and Ayesha Gandhi. BANANAS: An evolutionary framework for explicit and multipath routing in the internet. In *Proc. SIGCOMM Workshop on Future Directions in Network Architecture*, August 2004.

[100] Stephen Kent, Charles Lynn, Joanne Mikkelson, and Karen Seo. Secure border gateway protocol (S-BGP). *IEEE JSAC*, 18(4), April 2000.

[101] Angelos D. Keromytis, Vishal Misra, and Dan Rubenstein. SOS: Secure overlay services. In *SIGCOMM*, August 2002.

[102] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. on Computer Systems*, August 2000.

[103] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A data-oriented (and beyond) network architecture. In *SIGCOMM*, August 2007.

[104] Shannon K. Kuntz, Richard C. Murphy, Michael T. Niemier, Jesus Izaguirre, and Peter M. Kogge. Petaflop computing for protein folding. In *In Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, pages 12–14, 2000.

[105] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, February 2007.

[106] Mohit Lad, Dan Massey, Dan Pei, Yiguo Wu, Beichuan Zhang, and Lixia Zhang. PHAS: A prefix hijack alert system. In *USENIX SECURITY*, July 2006.

[107] Jun Li, Jelena Mirkovic, Mengqiu Wang, Peter Reiher, and Lixia Zhang. SAVE: Source address validity enforcement protocol. In *INFOCOM*, June 2002.

[108] Xin Liu, Ang Li, Xiaowei Yang, and David Wetherall. Passport: Secure and adoptable source authentication. In *NSDI*, April 2008.

[109] Xin Liu, Xiaowei Yang, and Yanbin Lu. To filter or to authorize: Network-layer DoS defense against multimillion-node botnets. In *SIGCOMM*, August 2008.

[110] Miloš Liška, Petr Holub, Andrew Lake, and John Vollbrecht. CoUniverse orchestrated collaborative environments with dynamic circuit networks. volume 0, pages 300–305, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[111] Mohsen Machhout, Zied Guitouni, Kholdoun Torki, Lazhar Khriji, and Rached Tourki. Coupled fpga/asic implementation of elliptic curve crypto-processor. In *International Journal of Network Security and Its Applications*, 2010.

[112] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Controlling high bandwidth aggregates in the network. *ACM CCR*, 32(3), July 2002.

[113] Ratul Mahajan, David Wetherall, and Thomas Anderson. Mutually controlled routing with independent ISPs. In *NSDI*, April 2007.

[114] Zhuoqing Morley Mao, Jennifer Rexford, Jia Wang, and Randy H. Katz. Towards an accurate AS-level traceroute tool. In *SIGCOMM*, August 2003.

[115] P. Marques, R. Bonica, L. Fang, L. Martini, R. Raszuk, K. Patel, and J. Guichard. Constrained route distribution for Border Gateway Protocol/MultiProtocol Label Switching (BGP/MPLS) Internet Protocol (IP) Virtual Private Networks (VPNs). RFC 4684, Network Working Group, November 2006.

[116] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. 1st Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[117] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *SOSP*, December 1999.

[118] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks (editorial). *ACM CCR*, April 2008.

[119] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, 2010.

[120] Alper Tugay Mizrak, Yu chung Cheng, Keith Marzullo, and Stefan Savage. Fatih: Detecting and isolating malicious routers. In *IEEE DSN*, June 2005.

[121] J. Moy. OSPF version 2. RFC 2328, Network Working Group, April 1998.

[122] Dalit Naor, Amir Shenhav, and Avishai Wool. One-time signatures revisited: Practical fast signatures using fractal merkle tree traversal. In *IEEE 24th Convention of Electrical and Electronics Engineers*, 2006.

[123] Jad Naous, Arun Seehra, Michael Walfish, David Mazières, Antonio Nicolosi, and Scott Shenker. The design and implementation of a policy framework for the future Internet. *Department of Computer Science, The University of Texas at Austin*, (TR-09-28), September 2009. `http://www.cs.utexas.edu/~mwalfish/icing-tr-09-28.pdf`.

[124] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, December 1998.

[125] NIST. Secure hash standard - fips-180-1. http://www.itl.nist.gov/fipspubs/fip180-1.htm, 1995.

[126] NIST. Advanced encryption standard - fips-197. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[127] Venkata N. Padmanabhan and Daniel R. Simon. Secure traceroute to detect faulty or malicious routing. In *SIGCOMM*, August 2003.

[128] Panagiotis Papadimitratos and Zygmunt J. Haas. Secure routing for mobile ad hoc networks. In *Proc. SCS Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS)*, January 2002.

[129] Panagiotis Papadimitratos and Zygmunt J. Haas. Secure link state routing for mobile ad hoc networks. In *Proc. IEEE Workshop on Security and Assurance in Ad hoc Networks*, January 2003.

[130] Douglas F. Parkhill. *The Challenge of the Computer Utility.* Addison-Wesley, US, 1966.

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

[132] Radia Perlman. Routing with Byzantine robustness. Technical Report TR-2005-146, Sun Microsystems, August 2005.

[133] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building planetlab. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[134] Lucian Popa, Norbert Egi, Sylvia Ratnasamy, and Ion Stoica. Building extensible networks with rule-based forwarding. In *OSDI*, October 2010.

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

[136] Barath Raghavan and Alex C. Snoeren. A system for authenticated policy-compliant routing. In *SIGCOMM*, September 2004.

[137] Venugopalan Ramasubramanian and Emin Gün Sirer. The design and implementation of a next generation name service for the Internet. *ACM CCR*, 34:331–342, August 2004.

[138] Y. Rekhter, Y. Li, and S. Hares. A border gateway protocol 4 (BGP-4). RFC 4271, Network Working Group, January 2006.

[139] Fabio Ricciato, Marco Mellia, and Ernst W. Biersack, editors. *Traffic Monitoring and Analysis, Second International Workshop, TMA 2010, Zurich, Switzerland, April 7, 2010, Proceedings*, volume 6003 of *Lecture Notes in Computer Science*. Springer, 2010.

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

[141] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[142] Kimaya Sanzgiri, Bridget Dahill, Brian Neil Levine, Clay Shields, and Elizabeth M. Belding-Royer. A secure routing protocol for ad hoc networks. In *Proc. IEEE Conference on Network Protocols*, November 2002.

[143] Stefan Savage, Tom Anderson, Amit Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan. Detour: a case for informed internet routing and transport. *IEEE Micro*, 19(1):50–59, Jan 1999.

[144] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Network support for IP traceback. *ACM/IEEE Trans. on Networking*, 9(3), June 2001.

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

[146] Arun Seehra, Jad Naous, Michael Walfish, David Mazières, Antonio Nicolosi, and Scott Shenker. A policy framework for the future Internet. In *HotNets*, October 2009.

[147] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[148] K. Sollins. Architectural principles of uniform resource name resolution. RFC 2276, January 1998.

[149] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet Indirection Infrastructure. In *SIGCOMM*, August 2002.

[150] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Transactions on Networking*, 2002.

[151] Lakshminarayanan Subramanian, Volker Roth, Ion Stoica, Scott Shenker, and Randy H. Katz. Listen and whisper: Security mechanisms for BGP. In *NSDI*, March 2004.

[152] Carl A. Sunshine. Source routing in computer networks. *SIGCOMM Comput. Commun. Rev.*, 7:29–33, January 1977.

[153] Jean sbastien Coron, Marc Joye, David Naccache, Pascal Paillier, and Gemplus Card International. Universal padding schemes for RSA. In *Proc. CRYPTO*, pages 226–241. Springer-Verlag, 2002.

[154] Christian Tschudin and Richard Gold. Network Pointers. In *HotNets*, October 2002.

[155] Jonathan S. Turner, Patrick Crowley, John DeHart, Amy Freestone, Brandon Heller, Fred Kuhns, Sailesh Kumar, John Lockwood, Jing Lu, Michael Wilson, Charles Wiseman, and David Zar. Supercharging PlanetLab: a high performance, multi-application, overlay network platform. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 85–96, New York, NY, USA, 2007. ACM.

[156] Todd Underwood. Peering-the fundamental architecture of the Internet. In *Renesys Blog*, December 2005. `http://www.renesys.com/blog/2005/12/peering_the_fundamental_archit.shtml`.

[157] R. Venkateswaran. Virtual private networks. *IEEE Potentials*, 20:11–15, February 2001.

[158] Michael Walfish, Hari Balakrishnan, and Scott Shenker. Untangling the Web from DNS. In *NSDI*, March 2004.

[159] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *OSDI*, December 2004.

[160] Tao Wan, Evangelos Kranakis, and P. C. van Oorschot. Pretty secure BGP (psBGP). In *NDSS*, 2005.

[161] XiaoFeng Wang and Michael K. Reiter. A multi-layer framework for puzzle-based denial-of-service defense. *International Journal of Information Security*, 2007. Forthcoming and published online, `http://dx.doi.org/10.1007/s10207-007-0042-x`.

[162] Benchmark: Apache2 vs. Lighttpd (static HTML files). `http://www.howtoforge.com/benchmark-apache2-vs-lighttpd-static-html-files`.

[163] Web server performance comparison: LiteSpeed 2.1 vs. `http://www.litespeedtech.com/web-server-performance-comparison-litespeed-2.1-vs.html`.

[164] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[165] Edmund L. Wong, Praveen Balasubramanian, Lorenzo Alvisi, Mohamed G. Gouda, and Vitaly Shmatikov. Truth in advertising: lightweight verification of route integrity. In *PODC*, August 2007.

[166] Wen Xu and Jennifer Rexford. MIRO: Multi-path interdomain routing. In *SIGCOMM*, September 2006.

[167] Abraham Yaar, Adrian Perrig, and Dawn Song. Pi: a path identification mechanism to defend against DDoS attacks. In *Proc. IEEE Symposium on Security and Privacy*, May 2003.

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

[169] Abraham Yaar, Adrian Perrig, and Dawn Song. FIT: fast internet traceback. In *INFOCOM*, March 2005.

[170] Abraham Yaar, Adrian Perrig, and Dawn Song. StackPi: New packet marking and filtering mechanisms for DDoS and IP spoofing defense. *IEEE JSAC*, 24(10):1853–1863, October 2006.

[171] Shixing Yan, Bu Sung Lee, and Sharad Singhal. A model-based proxy for unified IaaS management. In *2010 4th International DMTF Academic Alliance Workshop on Systems and Virtualization Management (SVM)*. IEEE, October 2010.

[172] Xiaowei Yang, David Clark, and Arthur W. Berger. NIRA: A new inter-domain routing architecture. *ACM/IEEE Trans. on Networking*, 15(4), August 2007.

[173] Xiaowei Yang and Xin Liu. Internet Protocol made accountable. In *HotNets*, October 2009.

[174] Xiaowei Yang and David Wetherall. Source selectable path diversity via routing deflections. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 159–170, New York, NY, USA, 2006. ACM.

[175] Xiaowei Yang, David Wetherall, and Thomas Anderson. A DoS-limiting network architecture. In *SIGCOMM*, August 2005.

[176] Manel Guerrero Zapata and N. Asokan. Securing ad-hoc routing protocols. September 2002.

[177] Xin Zhang, Abishek Jain, and Adrian Perrig. Packet-dropping adversary identification for data plane security. December 2008.

[178] Wenchao Zhou, Eric Cronin, and Boon Thau Loo. Provenance-aware secure networks. In *International Workshop on Networking meets Databases (NetDB)*, April 2008.

[179] Earl Zmijewski. You can't get there from here. In *Renesys Blog*, March 2008. `http://www.renesys.com/blog/2008/03/you-cant-get-there-from-here-1.shtml`.

Appendix A

ICING-PVM Cryptographic functions

prf, prf-96, and prf-32. Figure A.1 reports pseudocode for our implementation of PRF(k, d), where k is a 128-bit key and d is a 256-bit data block. PRF(k, d) is based on a two-round CBC-MAC, with optimizations made possible owing to the fixed, short

```
 1: function PRF($key[0:127], block[0:255]$)
 2:     Set $X$ = AES-128-FORWARD($key, block[0:127]$)
 3:     Set $T$ = AES-128-FORWARD($key, X \oplus block[128:255]$)
 4:     return $T$

 5: function PRF-96($key[0:127], block[0:255]$)
 6:     Set $T$ = PRF($key, block$)
 7:     return $T[0:95]$

 8: function PRF-32($key[0:127], block[0:255]$)
 9:     Set $T$ = PRF($key, block$)
10:     return $T[96:127]$
```

Figure A.1: $A[x:y]$ indicates bits x to y of A. PRF is based on two rounds of CBC-MAC and implements a keyed pseudorandom function mapping 256 bits to 128. PRF-96 (resp., PRF-32) truncates the result of PRF to its first 12 (resp., last 4) bytes.

```
1: function NEXT-TAG-KEY2($m_{N_i:t/p}$, $t[0:31]$, $p$, $g$)
2:     $q = p + 2^g$
3:     $block = t[p : q-1] \parallel 1 \parallel 0^{(127-2^g)}$
4:     $m_{N_i:t/q}$ = AES-128-FORWARD($m_{N_i:t/p}$, $block$)
5:     return $m_{N_i:t/q}$
6: function GET-TAG-KEY2($m_{N_i:t/p}$, $t[0:31]$, $p$, $p_{final}$, $g$)
7:     $m_{iter} = m_{N_i:t/p}$
8:     while $p < p_{final}$ do
9:         $m_{iter}$ = NEXT-TAG-KEY2($m_{iter}$, $t$, $p$, $g$)
10:        $p = p + 2^g$
11:    return $m_{iter}$
```

Figure A.2: Pseudocode for tag key calculation. GET-TAG-KEY2 can create prefix keys or tag keys. $m_{N_i:t/p}$ is the prefix key, t is the tag, p is the length of the tag prefix for which $m_{N_i:t/p}$ is the key, p_{final} is the desired length of the resulting prefix for which to get a key, g is a parameter used for performance tweaking and restricts the possible prefix lengths. $32/2^g$ is the number of valid prefix lengths: $p = 2^g, 2 \cdot 2^g, 3 \cdot 2^g \ldots, 32 - 2^g, 32$. Smaller values of g require more rounds of AES.

input size. PRF-96 returns the first 12 bytes of the result of PRF, which we assume to have 96-bit security, suitable for message-authentication purposes. PRF-32 returns the last 4 bytes of the result of PRF. We assume PRF-32 to have 32-bit security; while insufficient to serve as a MAC, this provides enough unpredictability to serve as an effective DoS hardener.

get-tag-key. Calculating $s_{N_i:tag_i}$ requires serial invocations of a pseudorandom function on short inputs (we use AES in its forward direction). Figure A.2 describes a parametrized version of GET-TAG-KEY that allows an ICING-PVM node to trade off flexibility in the choice of allowable tag prefix lengths for a decrease in the number of required serial AES rounds. For example, by using $g = 3$, a node restricts the prefix lengths to multiples of 8: /8, /16, /24, and /32, which means that the provider can delegate tag prefixes only with these lengths. This restriction would enable the node to bound the number of AES rounds for its valid prefix length to 4. When combined with caching of precalculated prefix keys, the number of required AES invocations can be as little as 1.

Jad Naous