PoCoMON*
A POlicy-COMpliant Overlay Network

Michael Miller

May 6, 2011

Abstract

On today's commodity internet, senders have no control over the path that their packets take beyond the first hop. This aspect of today's internet prevents many potential network policies from being implemented. Source routing has been proposed as a mechanism to solve this problem, in which the sender chooses the path the traffic will take. However, previous approaches have either not enforced policies specified by the sender, or have been prohibitively expensive in terms of computing power. ICING-PVM addresses these concerns by efficiently implementing source-based routing in a manner which allows path preferences to be enforced. PoComON builds on top of ICING-PVM, and provides a transitional overlay network to deploy ICING-PVM on today's internet supporting legacy applications.

1 Intro

1.1 → ICING-PVM

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 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

*Some of this document is borrowed (nearly) verbatim from the ICING-PVM USENIX Security submission [25]. Sections/paragraphs that are borrowed are denoted with a → .

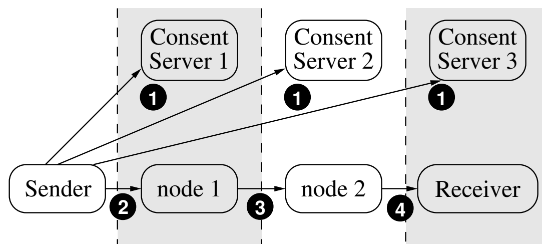

Figure 1: ICING-PVM's components and forwarding steps. ❶ In the general case, the sender gets PoCs from the consent servers of all nodes on the path (in practice, a consent server can delegate PoC-issuing, making this step lightweight). ❷ The sender creates and sends the packet to the first ICING-PVM 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.

fine-grained control over which providers carry which traffic between its branch offices, yet the network paths must respect the providers' pairwise business relationships.

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 extremely expensive).

ICING-PVM tries to fill that void. We describe a new networking primitive that we call a PVM (Path Verification Mechanism). A PVM ensures that the paths that packets actually take through the network respect the policies of those handling the packets.

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 ascertain (1) that it or its delegate had approved the packet's purported path, and (2) that the packet has followed that path so far.

To validate our design, Jad Naous implemented ICING-PVM in hardware, on the NetFPGA platform [1]. This implementation achieves 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. The hardware component to ICING-PVM is not a critical part of this paper, but it is mentioned to give a basis for comparison with the software implementation.

1.2 Contributions

This thesis makes two primary contributions: implementing PoComON, and revitalizing the source code base to a working state. These contributions are detailed below.

1.2.1 Revitalization

A large component of the work I did on the project was revitalization of the ICING-PVM source code to a working state, and getting a working experiment setup. When I encountered the project for the first time, the code base was undocumented, and there was no presently working experimental setup. I had to figure out what each application in the repository did, and how all the applications fit together. As part of my work on ICING-PVM, I figured out how to get a working experiment, and documented the process. When I first started to work on ICING-PVM, there was an overlay forwarder implemented, and basic send/receive programs implemented which did no encapsulation/deencapsulation, and did not support bidirectional communication.

In later sections, I discuss some of the issues I encountered while getting ICING-PVM's overlay functionality working, and getting a working experiment setup.

1.2.2 PoComON

One potential issue with ICING-PVM's deployment is that it requires custom hardware, and would not be usable on today's internet without significant architectural changes. To overcome this challenge, we implemented PoComON, an overlay network that runs on top of UDP, allowing ICING-PVM traffic to pass over the commodity internet. PoComON is compatible with existing legacy applications.

PoComON is built on top of the overlay functionality present in ICING-PVM. We adapted the overlay network to support receiving traffic from legacy applications and delivering traffic to legacy applications. We also implemented a path selection policy to demonstrate ICING-PVM's functionality.

We demonstrate an unmodified *ssh* communicating with a remote host over PoComON. We anticipate that PoComON will ease the deployment of ICING-PVM, allowing for a gradual transition.

1.3 Outline

Section 2 describes the ICING-PVM data plane, drawing from our previous USENIX security submission. In Section 3, we give a brief overview of the ICING-PVM control plane, the mechanism by which ICING-PVM determines paths. Section 4 gives an overview of PoComON, an overlay network developed to ease the deployment of ICING-PVM. Section 5 provides an overview of the work needed to revitalize ICING-PVM into a known good codebase with a working system.

2 → Overview of ICING-PVM

We now describe ICING-PVM at a high level, including its threat model.

2.1 Architecture and components

ICING-PVM can be the forwarding mechanism of a new network layer (analogous to IP) or an overlay network ([5, 29, 19, 28, 30, 15, 10]). In both cases, a ICING-PVM network comprises ICING-PVM ***nodes***, which enforce Path Compliance. In the network layer case, ICING-PVM nodes would be deployed by network providers at the ingress boundaries of their networks; internally, forwarders need not implement ICING-PVM. This scenario is default-off, which necessitates careful bootstrapping; the details are beyond this thesis's scope, but similar problems have been treated elsewhere [6, 31, 23]. In the overlay case, the ICING-PVM nodes are ***waypoints*** interconnected by the underlying network (IP) and deployed by providers. This section is agnostic about the deployment scenario; the ICING-PVM nodes are the machines that participate in ICING-PVM, possibly including end-hosts.

To communicate with a receiver, the sender assembles a ***path*** of nodes. ICING-PVM is concerned with forwarding and is orthogonal to path retrieval (routing). How senders retrieve paths depends on the scenario; the options include querying DNS to get a path (instead of an IP address), purchasing access to a remote ISP via its Web site, static configuration , using a gateway that participates in a routing protocol, etc.

This thesis mostly assumes that the sender has candidate paths in hand. Figure 1 summarizes ICING-PVM's forwarding. 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 implemented on general-purpose servers; this allows policies to be flexible, fine-grained, and evolvable [13, 8, 9, 14, 28]. At present, the consent server is merged with the path server. In our current implementation, the path server is hardcoded with each node's private credentials, and thus is is able to mint PoCs on behalf of any host on our sample network. This limitation could be fixed by implementing key delegation, something we intend to do in the future.

The ICING-PVM nodes downstream of the sender ensure that packets follow approved paths. This job decomposes into three tasks: (1) the node checks that a path is approved; (2) it checks that the path is being followed; and (3) it proves to downstream ICING-PVM nodes that it has seen the packet. Later, we describe in (sometimes annoying) detail how ICING-PVM nodes perform these functions. The high-level construction is depicted in Figure 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

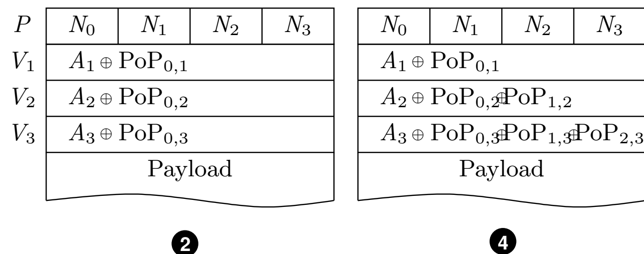

Figure 2: Simplified ICING-PVM packet at steps ❷ and ❹ from Figure 1. Two crucial header fields are the path (P) and the verifiers (V_j's). The sender (N_0) initializes the verifiers with path authenticators (A_j's) derived from the PoCs and the packet content. Each node N_i checks its verifier (V_i) and updates those for downstream nodes (V_j for $j > i$) to prove that it passed the packet. $\text{PoP}_{i,j}$ is a proof to N_j that N_i has carried the packet.

nodes derive the keys from their IDs.

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 this model in the next subsection and now list some 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. It seems extremely hard to *prevent* such misbehavior in our environment; what ICING-PVM can do is *detect* it. Indeed, 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. This can be caused by a number of different issues. For example, either node(or an intermediary node) could have violated the ICING-PVM protocol. However, it is also possible that each node was obeying the protocol, and a packet was dropped. Thus, a difference in counts does not necessarily imply that deviant behavior has occurred.

Similarly, an ICING-PVM node can copy packets elsewhere, or pass packets through a hidden node. These, too, seem very hard to prevent in a federated environment. However, unlike in the status quo, ICING-PVM senders and receivers can *choose* their path—and can include only nodes that they trust not to leak their packets. This choice and encryption are complementary: 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. It also does not provide information about whether a packet received any contracted-for services at a node.

In its current form, 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 [31] has shown how to perform such accounting with minimal forwarder state, and ICING-PVM could be extended to incorporate this technique.

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 [22] 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), for instance trying to abuse the delegation mechanism, or trying to make an honest node N_i accept a packet whose path was not approved by (a delegate of) N_i's consent server 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 [27] is a secure keyed pseudorandom function with full 128-bit security, that PMAC [7] is a fully secure deterministic MAC that is also a secure keyed pseudorandom function, and that CHI [16] is a secure hash function even if the hash is truncated to 248 bits.

2.4 Naming

Each ICING-PVM node assigns itself an identifier, called a ***node ID***, that is a unique public key. The node retains the corresponding private key. With such self-certifying names [24, 4], 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 node IDs, each associated with a path-specific tag. A ***tag*** is an identifier that has local meaning to the node and its provider; it describes specific handling or a service to apply to the packet. For example, a tag can describe a priority level for queuing, identify a customer to bill, select an output link, request virus-scanning services, or specify a combination of these. It can be thought of as a generalized MPLS label [11] (and shares some functionality with the vnode mechanism in [12]). 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.

2.5 Proofs of consent (PoCs)

When presented with a path, a node's consent server checks that the path complies with the provider's policy, perhaps by incorporating external information (billing, authentication, etc.). If so, the consent server creates a PoC and returns it to the sender. A PoC permits the sender to transit the node only using the given tag. It includes a cryptographic token, specific to the path and computed under a ***tag key*** that is unique to the given node ID 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 disclosing all of its tag keys.

2.6 Packet creation and proofs of provenance (PoPs)

As mentioned above, the 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 a node N, given a packet, 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, a 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 derivation uses a non-interactive Diffie-Hellman key exchange, and a node caches the results.

2.7 Packet processing: Verification and forwarding

Each node that receives the packet:

1. computes the PoC from the packet header and the tag key that the node shares with its consent server;
2. derives the PoP keys that it shares with the upstream nodes (which it can do from their IDs in the header);
3. computes the MACs of the packet (PoPs) under those PoP keys; and
4. checks that the PoC and PoPs are correct.

The PoPs computed in step 3 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:

5. derives the PoP keys that it shares with the downstream nodes (again from IDs in the header);
6. computes additional PoPs under those PoP keys;
7. inserts the PoPs into the header; and
8. 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, when the node receives a packet, the PoC and the PoPs that it inspects in steps 1–4 above are XORed together in an aggregate MAC; this approach reduces space while protecting the security of each of the components [18].

3 Overview of control plane

3.1 Overview

The data plane, described in the previous section, details how traffic passes through forwarders. At each forwarder, the data plane's role is to verify that traffic is following its prescribed path. By contrast, the control plane's role is enumerating and deciding between paths. The control plane consists of three pieces:

1. The client, which requests possible paths to a host, and determine which path to send data over, based on whatever policies it implements.
2. The path server, which returns a list of possible paths to clients.
3. The sIRP routers, which propagate routing information about paths to path servers. sIRP is not currently implemented, and is described in the future work section.

3.2 Client

The sender of traffic is responsible for choosing between paths returned by the path server. It can do so based on whatever policies it wants. For example, the client might decide that it doesn't want paths to pass through a non-friendly government (for example, the US DoD may decide that it does not want traffic to pass through a disliked nation).

3.3 Path server

The path server is responsible for returning possible paths to a given host. A client requests paths by sending an RPC to the path server containing two arguments. First, it provides the name of the host it wants to reach. Second, it provides list of partial paths which the client has consent to use. Upon receiving the RPC, the path server performs a 'join,' concatenating paths that have an intersecting suffix and prefix, using the partial paths provided by the client and the paths contained in the path server's path database. For example, client A can specify that it can connect to node B, and may ask for a path to node C. The path server knows of a path from B to C, and joins the partial path supplied by the client with a second partial path supplied by the path server, completing the path. The path server then returns this list of concatenated paths to the client. Communication between the client and the path server happens over vanilla UDP, out of band from ICING-PVM.

The path server's path database is currently hardcoded, because of the fact that sIRP is presently unimplemented.

The path server implements two RPC calls:

- **find_pathto_ps**(host, partialpaths), used for bootstrapping a path to the path server. This is unused in our current implementation of ICING-PVM.
- **find_pathto_host**(host, partialpaths), which a client invokes to get an ICING-PVM path to a host. The partialpaths argument specifies the paths which the client is aware of. The path server performs a 'join' between the partialpaths and the pathserver's path database (as described above), resulting in a list of paths to the host. The path server then returns the list of paths to the client. In our current implementation, PoCs are also minted by the path server on behalf of nodes. In addition to returning a path, this call also mints PoCs for the path.

3.4 Consent server

A ***consent server*** is responsible for minting PoCs and distributing them to clients. The criteria for deciding whether to mint PoCs for a client is undefined in the ICING-PVM protocol; the maintainer of the consent server can choose to implement whatever policies are desired. In the simplest implementation, each node could have a consent server, but this is not practical in a production environment; it would mean that for each communication path, the client would

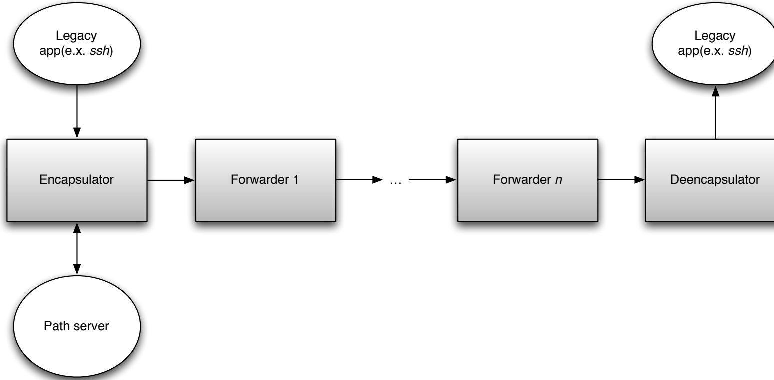

Figure 3: A high level overview of how the components of PoComON fit together. The legacy application sends traffic to the encapsulator, which hands traffic to a forwarder, using the path server to find an appropriate path. The forwarders keep handing off traffic to each other using the prescribed path until the packet reaches the destination, at which point the deencapsulator sends the encapsulated IP traffic to the local host.

have to ask each node along the path for PoCs. To mitigate this issue, the ICING-PVM protocol supports tag prefix delegation. The details of tag prefix delegation are beyond the scope of the paper, but can be found in [25].

Currently, the consent server is unimplemented. The path server fills its void, minting PoCs on behalf of all nodes in the network.

4 PoComON

4.1 Overview

One of the limitations to deployment for ICING-PVM is that it requires support from every node along the path. Each node must be able to issue PoCs and verify their presence. This need for network support limits the ability of ICING-PVM to be deployed on today's internet.

As mentioned in Section 1.2, PoComON solves this roadblock to deployment by implementing an overlay network [17], on top of which ICING-PVM runs. The overlay network runs at the Application Layer, on top of UDP(as opposed to ICING-PVM, which runs at the Internet Layer, on top of Ethernet). PoComON could conceivably run at the Transport Layer on top of IP, but we chose to run it over UDP for convenience.

A high-level picture of the architecture of PoComON can be seen in Figure 3. Bidirectional communication follows a similar flow, except in the reverse direction.

4.2 Design

As shown in Figure 3, PoComON consists of three pieces: the ***Encapsulator***, the ***Forwarder***, and the ***Deencapsulator***. The Encapsulator is responsible for packaging up legacy IP traffic into an ICING-PVM packet, and sending it over a UDP socket to a Forwarder. The Forwarder is responsible for passing traffic to the next ICING-PVM node in the path, either over UDP or through custom hardware. The Deencapsulator is responsible for routing the traffic to its final destination, either over the commodity internet, or by delivering it to the machine the Deencapsulator is running on. PoComON is designed on top of Click [21], making use of prebuilt Click element to ease development.

4.3 Encapsulator

The Encapsulator is built as a combination of a new Click element, several pre existing click elements, and a Click configuration file. Its operation can be summarized with the following steps:

1. Receive IP packet over `tun` interface running on local host.
2. Determine hostname of destination ICING-PVM overlay node.
3. Find path to destination ICING-PVM overlay node, in terms of ICING-PVM nodes.
4. Get PoCs for nodes along the path.
5. Encapsulate IP packet in ICING-PVM packet with PoCs generated in previous step.
6. Send ICING-PVM packet over UDP socket to next hop in path.

There are a number of subtleties to the above steps. In step (1), the OS kernel must route the packets to the `tun` interface setup by Click. To accomplish this, we added in entries to Linux's routing table to force traffic destined for specific IP addresses through the `tun` interface. Specifically, we ran: `sudo route add -host` ***REMOTE-IP*** `gw 1.0.0.1 tun0`. ***REMOTE-IP*** is the destination IP address, on the commodity Internet, of the host to which we want to send ICING-PVM traffic. `1.0.0.1` is used as the gateway, since it is the address of the `tun0` interface. Routing packets through this IP sends them to Click.

To accomplish step (2), the sender does a destination IP prefix $\mapsto$ ICING-PVM hostname mapping. In ICING-PVM, the hostname maps to a node ID, similar to how, on DNS and IP, a hostname maps to an IP address. In our implementation of PoComON, the mapping is hardcoded, but it could easily be made dynamic by using a dynamic lookup table(for example, a hash table), instead of hardcoding the mappings.

In step (3), the sender issues an RPC to the path server, which returns a list of possible paths to the destination host. The RPC is done using sfslite's [2] RPC facilities over a TCP connection with the path server. The connection happens

completely out of band, separate from ICING-PVM. The encapsulator then makes a decision about which path to take, based on whatever policies it decides to implement. In our PoComON implementation, we tested two policies for path selection: minimum length path, and maximum length path, and verified that traffic passed through the appropriate nodes.

In step (4), the sending machine mints PoCs; it is hardcoded with all of the information necessary to mint PoCs for any hosts it might encounter on any path. This is something we wish to fix in the future by creating a consent server, but we do not anticipate it affecting performance significantly.

In step (6), the next hop is determined by hardcoded values in the Click configuration file. We believe this represents a realistic deployment scenario: a customer likely always sends a packet directly to its ISP as the first hop. It would not be hard to have the encapsulator send packets to one of multiple forwarders, based on a path.

4.4 Forwarder

The Forwarder's job is to route the ICING-PVM packet to its next hop in the overlay path, either over the commodity Internet using a UDP socket, or over custom ICING-PVM hardware directly connecting the two nodes. The forwarder is implemented as a combination of a Click configuration file and several Click elements.

The Forwarder's actions can be summarized in the following steps:

1. Receive ICING-PVM packet, either over a UDP socket, or over custom ICING-PVM hardware.
2. Verify past nodes' PoPs are correct.
3. Verify that the current node's PoC in the packet is correct.
4. Attach a PoP to the packet, to show future nodes that the packet has passed through the forwarder.
5. Advance the hop ID in the packet.
6. Send the packet to the next forwarder or deencapsulator, either over a UDP socket, or over custom ICING-PVM hardware.

If a packet fails any of the verification steps, either for PoPs of past nodes, or the PoC of the present node, the packet is dropped. In the future, we hope to be able to notify the sender that its packet was dropped, but we do not support this at present.

The forwarder currently does not support ICING-PVM hardware, since there are still several bugs relating to hardware-software interaction.

4.5 Deencapsulator

The Deencapsulator, as with the Encapsulator and Forwarder, consists of several Click [20] elements and a Click configuration file. Its operation can be summarized in the following steps:

1. Receive packet from UDP socket or ICING-PVM hardware.
2. Verify the PoPs of past realms.
3. Verify the PoC of current realm.
4. Deencapsulate the IP packet contained inside the ICING-PVM packet
5. Rewrite the destination IP address to 1.0.0.1(which is the `tun0` interface's address)
6. Send the packet over the `tun` interface.

As with the forwarder, ICING-PVM hardware is not currently supported, for the aforementioned reasons.

The current implementation of the Deencapsulator (as described by the steps above) sends the packet to the local machine. We have implemented a version of a Deencapsulator which sends the packet over the `eth0` interface, but found that it was difficult to deploy due to the fact that the gateway on the local network inspects the source IP to see that the packet came from a local computer. We tried modifying the source IP, but with this modification, the destination host had no way to reply to the packet, since it didn't know who sent it. This limitation could be lifted if we had control over the router, but we did not, due to the fact that we were running our evaluation experiments on Amazon's EC2 service.

5 Experimental Evaluation of ICING-PVM

5.1 Experimental setup

We ran our experiments on Amazon's EC2 cloud computing service [3]. We chose to do this because it was easy to get up and running, and because if needed, we could scale up to many nodes quickly and inexpensively.

We used five medium 32-bit EC2 instances (six in some experiments), medium being the fastest 32-bit instance type we could use. We chose 32-bit instances because we have only qualified ICING-PVM to run on 32-bit machines. In the future, we hope to qualify ICING-PVM against 64-bit instances.

The five instances communicate over a private (to Amazon) gigabit network in Northern California. In order to provide a more realistic deployment scenario, we hope to spread our instances geographically in the future.

A diagram of the layout of the instances can be seen in Figure 4.

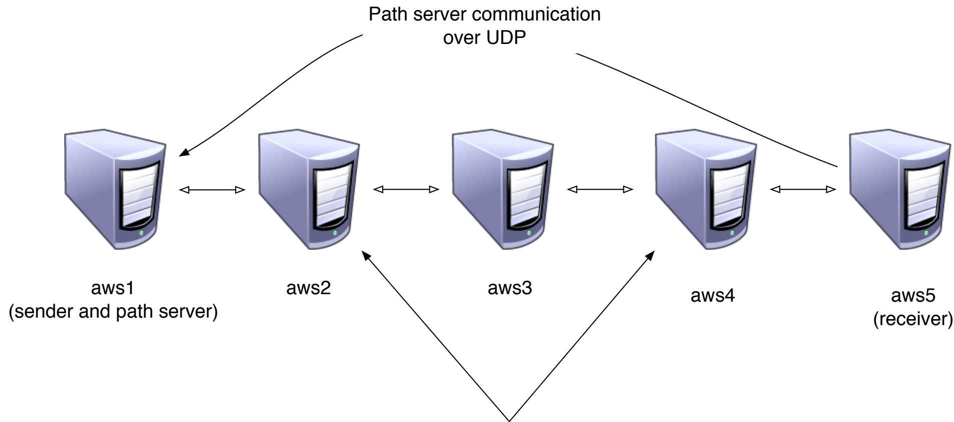

Figure 4: Experimental setup for our baseline experiment. The arrow in the triangular shape represents a direct bididirectional link between `aws2` and `aws4`.

5.2 Performance

Performance was not a primary objective of our first implementation of PoComON, but we report it anyway, to show how a (relatively) unoptimized implementation performs. We did several experiments to evaluate performance, which are detailed below. All of our performance data has been gathered in 50-packet intervals. We discard the first throughput measurement, and use the next 1000 for data. The rest of the data is discarded.

5.2.1 Baseline

Our "baseline" setup was 5 Amazon medium EC2 instances. The instances communicated in a linear fashion: `aws1` talked with `aws2`(bidirectionally); `aws2` talked with `aws3` (bidirectionally), etc.

We also constructed a path directly from `aws2` to `aws4`. This created a secondary path for traffic from the sender to receiver, and enabled us to test path selection.

We ran the path server on the first instance(named `aws1`), the forwarder ran on `aws`{2–4}, and the deencapsulator on `aws5`. We also ran a "receive" program on `aws5` which counts the packets received and reports throughput. We executed a "send" program on `aws1`, which simply sent out 4-byte UDP packets to the receiving host.

The experimental setup can be seen in Figure 4.

We received a mean throughput of 2628.62 packets per second($\bar{x} = 2408.91$, $\sigma = 1243.16$). The distribution of the data can be seen in Figure 5.

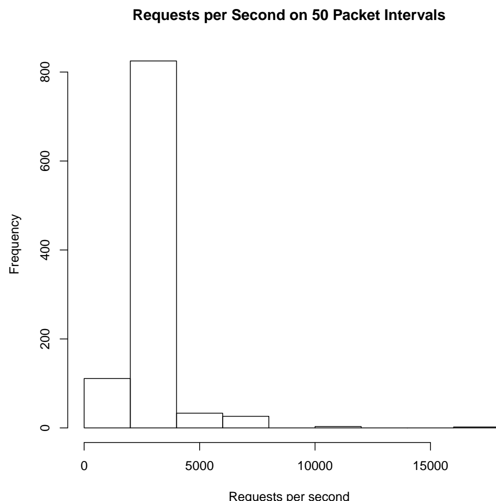

Figure 5: Data distribution for our baseline experiment.

5.2.2 Dedicated path server

Our next experiment consisted of running our baseline experiment, with the exception of having the path server be on a dedicated machine, `aws6`. We show the experimental setup in Figure 6. We expected throughput to go up, considering that the path server was a bottleneck in the baseline experiment.

Surprisingly, the throughput decreased when we made the path server dedicated. Our data had a mean throughput of 1271.08 packets per second($\bar{x}$ = 1258.91, σ = 223.69). We suspect this is due to the fact that the communication cost between the two servers was significantly higher than the computation cost to generate PoCs. The standard deviation in this experiment was much lower than the baseline. We suspect that Click and the path server competing for scheduler time caused the large standard deviation. The distribution of the data can be seen in Figure 7.

5.2.3 Path caching

Our following experiment tested throughput with path caching. In this experiment, we used the same setup as the previous experiment, but modified the sender to only request path information from the path server once; after the initial request, the information would be reused. This had the effect of eliminating the control plane's performance from the results; effectively, only the data plane was measured.

Unsurprisingly, our throughput went up enormously with this experiment. Our mean throughput was 6439.66 packets per second, ($\bar{x}$ = 5725.09, σ = 3056.92). The standard deviation was relatively high, suggesting that the data

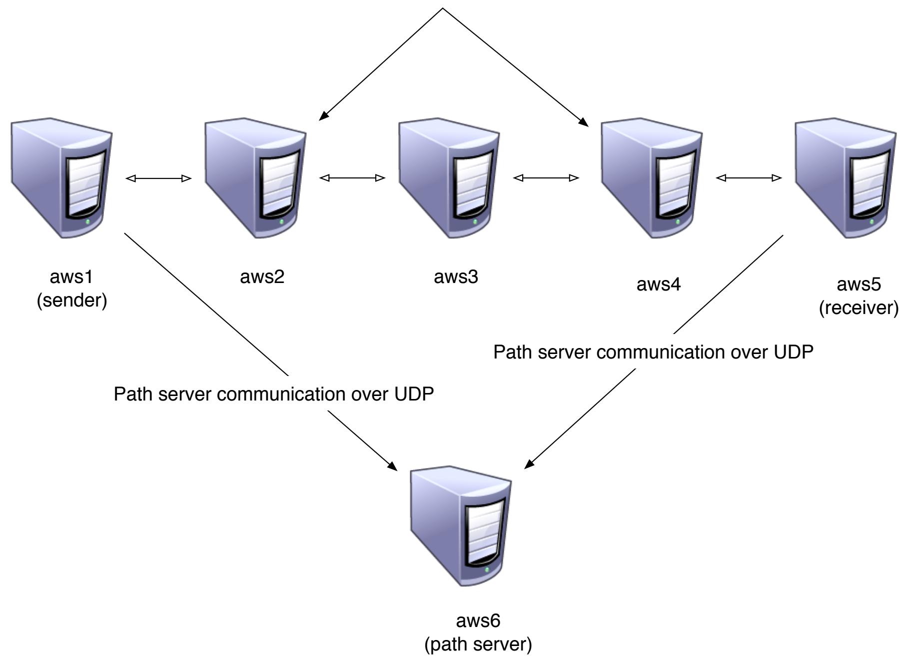

Figure 6: Experimental setup for our dedicated path server experiment. The arrow in the triangular shape represents a direct bididrectional link between `aws2` and `aws4`.

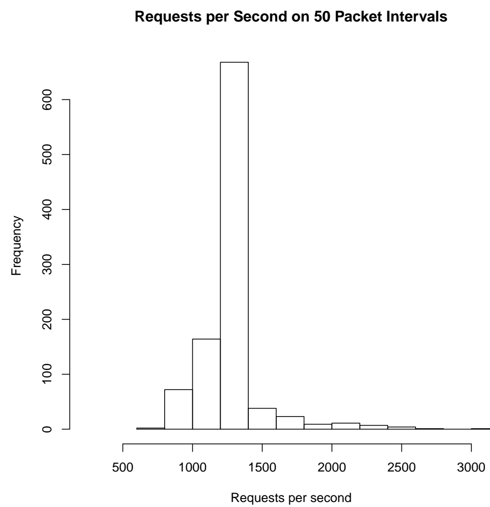

Figure 7: Data distribution for our dedicated path server experiment.

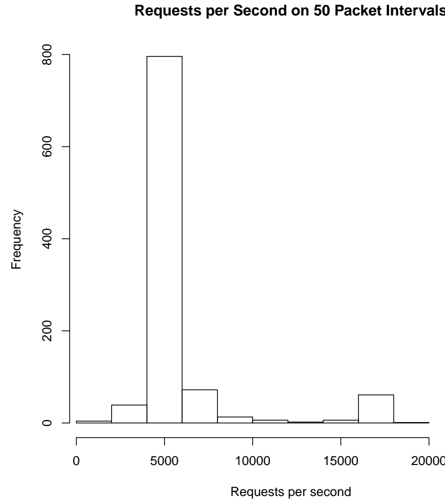

Figure 8: Data distribution for our path caching experiment.

plane had a high variance in throughput. The distribution of throughput for this experiment can be seen in Figure 8.

5.3 Path selection

We modified the path server, and forwarders to create a path that bypassed `aws3`, to simulate path selection. We then modified the encapsulator to choose either the path with the minimum length (the path bypassing `aws3`) or the maximum length (the path going through `aws3`). We verified that the correct path was taken with tcpdump, by running *tcpdump* on `aws3`.

5.4 Attacks

One attack which we wanted to consider in our evaluation was a malicious party sending a packet without valid PoCs. To test this, we configured the path server to mint bogus PoCs for `aws3`. We verified with *tcpdump* that packets which entered `aws3` were not forwarded to the next machine. This showed that our implementation correctly dropped packets with malformed PoCs.

6 Revitalization

A large part of the project ended up being revitalization. I started the project with nearly no documentation on how the system worked. Also, there was no working experiment setup available to me at the time I joined the project. As

a result, I had to do a lot of investigation to figure out how the build system worked, and how the execution environment was setup.

6.1 Execution environment

Figuring out the execution environment was particularly challenging. To start, there were two path servers in the repository, one of which was working, and one which was old and incompatible with the latest version of ICING-PVM (I didn't know that to begin with). Second, there was no obvious program to talk with the path server. I eventually discovered the programs `i2tx` and `i2rx`, which sent and received packets over ICING-PVM respectively.

6.2 Build environment

Perhaps the most challenging aspect of getting up to speed with ICING-PVM was the build environment. There were several arbitrary things that needed to be done to get the build working. For example, the environment variable `DO_B163_LIB=1` needed to be set for MIRACL's build to complete successfully. This was not obvious, and I had to go searching through the Makefile to find this out. I also had to figure out, by brute force, that I could only use GCC 4.3 or earlier to build the control plane, due to build errors on later versions of GCC.

Another example of the build system's lack of user friendliness is that MIRACL's headers needed to be patched **after `make install`** was run. The build would not complete if the file was patched beforehand. This is due to a class name conflict.

6.3 Bugs

Throughout the course of getting a working ICING-PVM setup working, I encountered a number of bugs in the code. In particular I encountered a heisenbug which manifested itself when I added the STL to the project. I also encountered a linker error that only manifested itself at -O0, and was masked by the fact that builds were done at -O2 by default. These bugs took a nontrivial amount of time to diagnose and fix, and were part of the revitalization I did with ICING-PVM.

6.4 Documentation and automation

After dealing with the horrors of an undocumented system, I resolved that no one would ever again experience the painful process of figuring out how to get ICING-PVM working.

To do this, I wrote a 42-step, 69-line document explaining how to get the control plane built. I also wrote up a set of Ruby scripts to automate getting a working experiment setup. I plan to write a comprehensive document describing how all of these pieces fit together in the near future.

7 Future work

7.1 sIRP

One limitation of the current ICING-PVM and PoComON implementation is that paths are hardcoded into the path server. This does not represent a realistic deployment scenario, where paths may change as hosts enter and leave the network. Today's Internet has solved the problem of path determination with routing protocols such as BGP.

A similar mechanism has been proposed for ICING-PVM: sIRP (Simple Icing Routing Protocol). The protocol is relatively simple: in essence, each (backbone) node broadcasts nodes it can reach to other neighbors, and the propagation continues until each node has a view of the topology of the network. Node departures are handled in a similar way: when a node loses a connection to a neighbor, it broadcasts this information to neighbors, who in turn broadcast this information to their neighbors, until the network is aware of the node's departure.

sIRP ensures valley free routing by assigning neighbors to one of three categories: either a node is a ***provider***, a ***peer***, or a ***customer***. Two connected neighbors must agree upon their relationship before path information will propagate between the neighbors. For example, Node A must agree that it is a provider for node B in order for node B to claim it is a customer of node A. As a rule, a customer will not propagate paths to peers, ensuring valley free routing.

sIRP has been partially implemented. The future work would entail completing it and testing it.

7.2 DNS hackery

An annoying aspect to working with PoComON is that routes must be preset in Linux's routing table for each possible destination host. We propose a mechanism [26] by which users could connect to hosts of the form `tag.realm.pocomon.net`, and have the requests automatically routed over PoComON. This would entail building a DNS server to create fake IP addresses(in the 1.x.x.x subnet) for each tag/realm pair, and keep track of the mappings. The DNS server also needs to communicate the mappings to the Encapsulator, so that the encapsulator knows which node to send incoming traffic to. Adding this DNS hackery would make using PoComON a much easier experience for users.

8 Conclusion

Today's routing policies limit the amount of control sends have over the path that traffic takes. ICING-PVM introduces a new protocol that enables more flexible routing policies, allowing senders to specify paths. However, ICING-PVM is relatively hard to implement over an existing network. PoComON makes it feasible to deploy ICING-PVM on today's internet, by implementing ICING-PVM

on top of an overlay network running over UDP. As a whole, for my thesis, I developed PoComON, got a working setup of ICING-PVM, and documented the system. PoComON is now in a working state, carrying legacy *ssh* traffic with no modifications.

References

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

[2] sfslite. `http://http://www.okws.org/doku.php?id=sfslite`.

[3] Amazon Web Services. Amazon elastic compute cloud (Amazon EC2). `http://aws.amazon.com/ec2`, February 2011.

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

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *ACM CCR*, 38(3):105–110, July 2008.

[15] 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.

[16] 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`.

[17] D. Joseph, J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. *OCALA: An architecture for supporting legacy applications over overlays*. Computer Science Division, University of California, 2005.

[18] 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.

[19] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *SIGCOMM*, August 2002.

[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[21] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(4):263–297, November 2000.

[22] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

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

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

[25] Jad Naous, Arun Seehra, Michael Walfish, David Mazières, Antonio Nicolosi, and Michael Miller. Panda: A mechanism for policy-compliant packet forwarding. Submitted to the 20th USENIX Security Symposium.

[26] 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. Submitted to the 2010 USENIX Symposium on Networked Systems Design and Implementation.

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

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

[29] 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.

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

[31] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, August 2005.