One Primitive to Diagnose Them All: Architectural Support for Internet Diagnostics

Ang Chen* Andreas Haeberlen* Wenchao Zhou† Boon Thau Loo*

*University of Pennsylvania †Georgetown University

Today, network operators are increasingly playing the role of part-time detectives: they must routinely diagnose intricate problems and malfunctions, e.g., routing or performance issues, and they must often perform forensic investigations of past misbehavior, e.g., intrusions or cybercrimes. However, the current Internet architecture offers little direct support for them. A variety of solutions have been proposed, but each solution tends to address only one specific problem. Moreover, each solution proposes a different fix that is incompatible with the others, which complicates deployment.

In this paper, we make the observation that most of the existing solutions share a common "functional core", which suggests that it may be possible to add a single primitive to the Internet architecture that can support a wide variety of diagnostic and forensic tasks. We then present one specific candidate that we call *secure packet provenance* (SPP). We show that SPP is easy to add to the current architecture, that it can be implemented efficiently in both software and hardware, and that it can be used to approximate (and sometimes surpass) the capabilities offered by a variety of existing diagnostic and forensic systems.

1. Introduction

Diagnostics and forensics were not among the top priorities for the original design of the Internet [17], and as a result, the architecture offers relatively little direct support for them. At the interdomain level, the only features that are likely to be available are ICMP and a few IP header options, and even these are often disabled [25] or implemented inconsistently [60]. Thus, when an operator encounters a problem that is not limited to her own network (such as bad performance on a given path), there is relatively little tool support; the best option is still often to post a message to a mailing list like NANOG, or to call other operators on the phone.

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064212

Reprinted from EuroSys '17,, [Unknown Proceedings], April 23 - 26, 2017, Belgrade, Serbia, pp. 1–15.

Over the years, a variety of diagnostic and forensic challenges have been identified. These include diagnosing high delay, reordering, or loss [9, 41, 45], identifying the source of attack traffic [43, 62], localizing failures [11, 36, 37], detecting prefix hijacking [71], testing for traffic differentiation [70], topology mapping [60], finding the root cause of routing problems [20, 66], collecting evidence of cybercrimes [42], and verifying SLAs between ISPs [24, 46, 53], and so on. Each of these challenges involves a specific problem *deep within the network*, which is difficult to diagnose without network-level support.

In the absence of direct support from the network, most existing work takes one of the following two approaches. The first is to approximate the missing functionality by creatively "abusing" a feature that exists for some other purpose (such as certain header options [60] or ICMP responses [11]). This is often surprisingly effective, but it typically relies on underspecified behavior and/or idiosyncrasies of certain router implementations, which can diminish data quality and require an enormous amount of ingenuity to work around (e.g., [36, 60]). The second approach is to extend the architecture with a new feature of some kind (e.g., [7, 53]), such as a new protocol, header field, etc. Such extensions provide a "clean" solution for the problem at hand, but deploying new features in the entire network is extremely difficult – so difficult, in fact, that hardly any of the proposed solutions have been widely deployed so far. To make matters worse, existing proposals typically focus on solving one particular problem and do not help with any of the other diagnostic and forensic problems that have been identified; thus, a comprehensive solution would require deploying *all* of the proposed extensions *in combination*. Given the ISPs' reluctance to make major changes to the network, this seems unrealistic.

In this paper, we ask the following question: If ISPs were willing to deploy *only one* new primitive in the network to help with diagnostics and forensics, what should that primitive be? Our key observation is that, while the existing solutions seem very different at first glance, *they all essentially answer variants of the same question: "What were the causes and/or effects of a given past event in the network?"*. If the network could remember recent events (such as packet

transmissions) and the corresponding causes and effects, even for a short amount of time, many forensic problems would be easy to solve. For instance, reverse traceroute [36] could locate the source of packet drops simply by following packets on their way from the sender to the receiver (and potentially back), and note the point at which they no longer made progress. Other forensic problems would require some post-processing: for instance, WhyHigh [41] could find the source of high latencies by inspecting the differences between packet transmission and arrival times, and Netdiff [46] could calculate the throughput on each path segment to find the bottleneck. However, this processing could be done at the edge without further changes to the core.

There is one additional feature that some forensic systems require: the ability to *prove* the correctness of a given answer [7, 24, 43, 53, 72]. This is necessary because attackers may falsify evidence to cover their tracks [40]. Although the strength of proofs and the properties being proven sometimes vary, in essence they are all concerned with the presence or absence of particular entries in our hypothetical ledger: for instance, the PoPs in ICING [53] essentially correspond to a chain of entries that connects a packet to a particular sender, the signatures in Passport [43] and AIP [7] correspond to the beginning of this chain, and the logs in SNP [72] correspond to causal connections along the chain.

This commonality suggests that it may be possible to deploy a *single* primitive in the network, once and for all, and then re-implement the previously proposed diagnostic and forensic systems as "applications" on top of it, without further changes to the network core. In this paper, we propose one specific candidate for such a primitive that we call *secure packet provenance* (SPP). SPP is based on the concept of data provenance from the database literature [14], which has already been used for diagnostics and forensics in other contexts, such as operating systems [31] and distributed systems [67, 72, 75]. However, as we show experimentally in Section 7, existing solutions would be completely overwhelmed with the high data rates at the network data plane. SPP solves this problem by avoiding cryptographic operations on the fast path and by relying mostly on ephemeral state; as a result, it outperforms the state-of-the-art secure provenance system by several orders of magnitude.

While the key insights of this paper are architectural (that there can be a single shared primitive, and that SPP is a good candidate), we have also designed and implemented a concrete protocol that could provide SPP in the Internet. Other than a small link-layer header, our protocol does not require any changes to the current data plane and can be implemented efficiently in hardware. (We demonstrate this with a NetFPGA prototype that runs at 10 Gbps.) We also used SPP to approximate six different diagnostic primitives from the literature, and we show that with SPP, each primitive can be implemented with just a few lines of code. Our main contributions are:

- Two architectural insights: that a single shared primitive can support a wide variety of diagnostic and forensic tasks, and that provenance is a good candidate for such a primitive (Section 2);
- the definition of a secure provenance model for the Internet's data plane (Section 3);
- SPP, a concrete protocol for maintaining secure provenance (Section 4);
- case studies showing that SPP can approximate a number of existing diagnostic systems (Section 5);
- software and hardware prototypes (Section 6); and
- an experimental evaluation, as well as proof-of-concept implementations of six diagnostic primitives on SPP (Section 7).

We discuss deployment strategies and their implications in Section 8, and we present related work in Section 9.

2. Overview

Diagnostics and forensics were not among the top priorities for the original Internet [17], as it was small in scale and experimental in nature. But today's Internet, with its broad range of applications, has attracted problems of all kinds [4, 30, 34, 40, 41, 57], which sometimes cause losses of millions of dollars [23]. But the available tools are far from adequate: packet traces, IP addresses [5], and even thumbnail images [2] are serving as evidence; due to the lack of reliable forensics, innocent users have been falsely accused of wrongdoings [2, 5, 65]. We believe that there is a need for better support for diagnostics and forensics.

2.1 Goal: A single primitive

There is a rich body of work on diagnostic and forensic systems that solve specific variants of this problem, typically by extending the Internet in one way or another [7, 10, 11, 18, 19, 26, 29, 36, 41, 43, 45, 51, 53, 58, 59, 62, 64, 70]. However, the resulting variety of problem-specific, mutually incompatible extensions represents a challenge for widespread deployment. Hence, rather than trying to improve any individual one of these systems, we ask: *Is there a single primitive that could be added to the Internet to solve a wide range of diagnostic and forensic challenges?* Such a primitive would not necessarily match the efficiency of the more specialized solutions, since a shared primitive would need to provide a strict superset of the functionalities of the individual primitives; but it could certainly be more efficient than deploying all of them together. Moreover, if so many existing diagnostic and forensic systems are based on some variant of this primitive, we have good reasons to believe that it will be useful for solving future, as-yet-unknown forensic challenges as well – which is a key requirement for any possible addition to the network architecture.

In this paper, we propose *secure packet provenance* as a candidate for such a primitive. At a high level, prove-

nance [14] tracks how data flows through the network by recording each event, e.g., the transmission of a packet, or the installation of a new route, and its direct causes and effects. With this information, any event of interest can be explained by recursively looking up the causes of the event until a set of "root causes" (such as the transmission of a new packet at an end host, or the origination of a new route) is reached. Additionally, our proposed primitive collects cryptographic evidence of network-level events; this can be used to authenticate the provenance even in adversarial settings.

2.2 Challenges

Intuitively, maintaining a secure provenance graph for the Internet would be sufficient for diagnostics and forensics, since it contains a complete and accurate description of what happened, why, when, and where. However, two key challenges need to be solved to make this approach practical.

Challenge #1: Overhead. A complete provenance graph of the entire Internet data plane would quickly consume any amount of space that could realistically be provided. We propose to solve this by keeping the full provenance only very briefly, and by offering a way to save (and later authenticate) any parts of the graph that are relevant for ongoing diagnostic and forensic tasks. To keep the computational overhead low, our proposed solution relies mostly on cryptographic primitives that can work at high speeds, such as hashing, and it applies several optimizations, such as batching.

Challenge #2: Privacy. Collecting all the provenance in a central location would be a privacy disaster. Our proposed solution avoids this by distributing the graph, and by allowing each network-level component to keep the part of the graph that pertains to itself. Also, we do not allow "global" queries of the form "show me all the packets that Bob sent" – users can only explore the provenance graph hop by hop, starting from a vertex they already know about. In effect, users can only query the provenance of packets they have already seen in their entirety. Moreover, we allow ISPs to restrict the visibility of their own subgraph; for instance, an ISP might permit its local admins to see its complete provenance, including routing policies and link statuses, but it might limit queriers from other domains to only forwarding-related information, e.g., the path that packets were sent on.

3. The provenance graph

We begin by defining the data model for the provenance information we wish to provide. A common way to represent the provenance is as a *provenance graph* [75] – a DAG in which each vertex represents an event and edges connect causes to effects. In this graph, the explanation of an event is simply the tree that is rooted as the corresponding vertex.

3.1 What is the right layer?

For a single provenance model to work for heterogeneous networks, it needs to be detailed enough to encode useful debugging information, but also general enough to abstract away hardware-specific features. We observe that this challenge resembles that of the original Internet, which needed to interconnect a variety of different network types and protocols. The answer in the original design was IP's "narrow waist", which was itself universal but permitted diversity at layers above and below. Thus, if our provenance model captures the network's operation at the IP layer, it will form a basis that different networks could agree on. As we will show in Section 3.4, operations on other layers can still be encoded as extensions to the IP-level provenance graph.

Network model: We model the network as a graph whose nodes are IP-capable devices. Each node has a number of ports, which can be connected to ports on other nodes using links. Nodes can transmit packets on their ports to some or all of the nodes that are connected to the corresponding (unicast or multicast) link. Therefore, this model not only includes routers and middleboxes, but also end hosts. Packets can be lost or corrupted in transmission, and nodes can mutate, duplicate, or drop any packet. Moreover, each node has a set of rules that decide how packets should be processed, and an increment-only local timer to obtain timestamps.

3.2 The provenance graph G

For clarity, we define the *provenance graph* $G := (V, E)$ from the perspective of a hypothetical global observer that can observe every single event in the network – i.e., every time a packet is sent or received, a link goes up or down, and a rule is inserted or deleted. G is a DAG and contains one vertex for each event, as well as a directed edge (v_1, v_2) whenever v_2 causally depends on v_1. Vertexes can have multiple incoming edges; for instance, a node might send a packet on a particular port because a) it received the packet earlier, b) it had a rule that matched the packet and specified this port, and c) the link on that port was up. Specifically, we define six vertex types, using a provenance model similar to the one from DTaP [74]:

- When a link l goes up/down on node N at time t, add a vertex LINKUP(N,t,l) / LINKDOWN(N,t,l).
- When a node N adds/removes a rule r at time t, add a vertex RULEADD(N,t,r) / RULEDEL(N,t,r).
- When a node N receives a packet p on port P at time t, insert a vertex v:=RECEIVE(N,p,P,t) to V; also, find the vertex v_1:=LINKUP(N,t,l) for the link l that is currently connected to P, and add an edge (v_1, v) to E.
- When a node N sends a packet p on port P at time t, add a vertex v:=SEND(N,p,P,t) to V. If p is sent because a packet p' was previously received by N at time t' on port P' and is forwarded to port P because of a rule r, find the vertexes v_2:=RULEADD(N,t'',r) and v_3 :=RECEIVE(N,p',P',t') in V and add edges (v_2, v) and (v_3, v) to E.

G is, in effect, a complete chronicle of everything that happened in the network: in principle, it is possible to "replay" the entire execution of the network in simulation. Thus, if a question can be answered in this very detailed simulation, it must also be possible to answer it using the information in G. In particular, to explain why an event e has occurred, we can simply find the corresponding vertex v in G and look at the subtree that is rooted at it, the leaves of which are the "root causes" that, in conjunction, have caused e to occur. Later, we will describe a *distributed* algorithm that maintains a close approximation of G without assuming a central entity. This is based on the observation that each vertex $v \in G$ has a natural "home": the node N that appears as the first entry will store the vertex v.

3.3 Querying and evidence

We allow users to examine the graph G with a query primitive QUERY(v) that returns v's adjacent vertexes in G. Thus, users could start with a vertex they know (say, the transmission or arrival of a packet at their local node) and explore the graph by invoking QUERY recursively. However, recall that G is distributed, and that each node stores the vertexes that pertain to it. So a malfunctioning or compromised node could fabricate or destroy vertexes that it stores locally. To prevent this, nodes are required to store not only the vertexes themselves, but also *evidence* to prove that the adjacent vertexes exist. The evidence e_v of a vertex v can be thought of as a statement that is signed by the "home" node of v saying that v is a part of G. Conceptually, nodes exchange evidence whenever they add an edge to G between two of their vertexes. Thus, each node can use the evidence to prove to any third-party that the other end of the edge must exist in G.

Hence, we augment the query primitive with evidence. QUERY(v,e_v) returns two sets of vertexes: all the predecessors and successors of v in G. Each returned vertex v' is accompanied with evidence that 1) v' is in V, and that 2) the relevant edge ((v', v) for predecessors and (v, v') for successors) is in E. As before, users can use QUERY to explore a larger portion of G by invoking QUERY recursively, starting from some vertex they know and have evidence for.

3.4 Extensions

The above data model only captures IP-level provenance. But as we discussed in Section 3.1, operations above and below the IP layer can be encoded as extensions to this basic provenance graph to support richer diagnostic and forensic capabilities. Below, we briefly sketch two examples.

Control-plane diagnostics: In the basic provenance model from Section 3.2, changes to link statuses and rules are "root causes" that cannot be explained further. However, it would be easy to add more entries to the TELs to further explain the provenance of these events. For instance, NetReview [24] already records a type of secure provenance for the BGP control plane; this provenance could be integrated with the IP-level model to further explain the RULE vertexes. Provenance tools that do not understand the new vertex types in the TEL could simply ignore them and continue to treat the RULE vertexes as basic events in the provenance graph.

Summarizations: To enable longer-term forensic queries, it could be useful to have a less detailed but smaller version of the IP-level provenance graph (say, a flow-level version); thus, the detailed version could be discarded after a few seconds, while the aggregated version could remain available for hours, or even days. Our basic model can accommodate such extensions by having routers commit to the basic graph and its summarizations simultaneously. As long as both endpoints of a link generate the summarizations in the same way (e.g., by using the same sampling technique), they can verify correctness exactly as in the basic IP-level version.

Visualization: To help operators to better understand the diagnostic results, the evidence can be displayed using provenance visualizers such as NetTrails [73].

3.5 Does G reveal too much information?

End users might be concerned that QUERY could be used to spy on their traffic. But our design prevents this: to query a vertex in G, the querier must already have evidence for an adjacent vertex. So, in order to access the provenance of a packet p that was sent from A to B, the querier must have some evidence of p's existence – which is only available at the sender A, the recipient B, and the ASes along the path, all of whom have already seen p in its entirety. Thus, there are only two cases: 1) the querier already knows that p exists, and what exactly it contains; in this case, QUERY will reveal where p came from, where it went, and what exactly happened along the way. Or 2) the querier does not yet know that p exists; in this case, the querier learns nothing from QUERY because the invocation will fail.

ISPs could have similar concerns about the topology and the configuration of their own infrastructure. But the Internet's topology can already be learned in great detail today [60], so G does not reveal much additional information – it merely reduces the effort that is needed to obtain it. Moreover, networks can protect policy-related information by hiding certain vertexes: each node can implement its own policy to decide which vertexes should be hidden. For instance, a network may want to reveal RULE and LINK vertexes only to its own admins, and hide them from users in other domains. Thus, each querier is presented with a *view* of the provenance graph, and can explore only the parts that are visible to her. To preserve usability, our provenance model prescribes that the SEND and RECEIVE vertexes be included in any view. Therefore, queries with a restricted view, e.g., inter-domain queries, can only trace packet paths from the returned SEND and RECEIVE vertexes; queries with an admin's view, e.g., intra-domain queries, can additionally learn why, i.e., from the RULE and LINK vertexes.

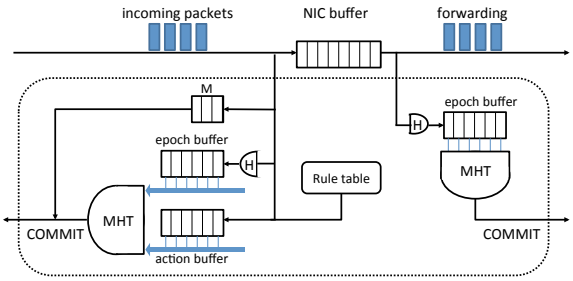

Figure 1: Data flow in the commitment protocol.

4. The SPP protocol

We now describe a distributed algorithm called SPP that implements the proposed provenance graph.

4.1 Assumptions and threat model

We design SPP based on the following assumptions:

- There is a hash function $H(\cdot)$ that is pre-image resistant and collision resistant.
- Each node i has a key pair π_i/σ_i that can be used to sign messages. A node i's signatures cannot be forged without knowing i's private key σ_i.
- If a link $i \rightarrow j$ exists, then j has a back channel for sending a small number of messages back to i.

The first assumption could be satisfied, e.g., by SHA-256. The second assumption could be satisfied with a small extension to the RPKI. The third assumption holds trivially for all bidirectional links; for other links, it could be satisfied by using a different link for the back channel.

Threat model: We assume that nodes can fail or be compromised by a *Byzantine* attacker, i.e., we make no special assumptions about the affected nodes, other than that they cannot break cryptographic keys. In particular, these nodes can drop, alter, or fabricate packets, they can destroy or tamper with any local state, and/or collude with each other.

4.2 Commitment protocol

The purpose of the commitment protocol (illustrated in Figure 1) is to generate evidence for the provenance graph. The protocol runs between the two endpoints A and B of each link $A \rightarrow B$; bidirectional links run two separate instances of the protocol, and nodes with multiple ports run separate instances for each local port. By this protocol, A would be able to prove that the packets it has sent have been received by B, and B would be able to prove that the packets it has received were indeed sent by A. This is done as follows.

Sender: A uses its local timer to divide time into *epochs* of some fixed length, e.g., 100ms, and both A and B maintain a small number of *epoch buffers* in which they record information about the packets they have sent or received. A begins a new epoch E_i by sending a message NEWEPOCH(i) to B. After that, whenever A sends a packet p to B, A appends the hash $H(p)$ to the buffer and then prepends an index j of packet in the epoch buffer as a small extra header before p. A ends E_i by sending message ENDEPOCH(i,n) to B, where n is the total number of packets it has sent to B in this epoch.

Receiver: B has meanwhile forwarded each packet as usual, but it has also recorded in its own epoch buffer the hashes of all the packets it has received correctly – i.e., without link-layer errors or CRC mismatches – from A; moreover, B has identified any missing index numbers (by looking for gaps in the sequence of numbers) and has recorded these in a small separate buffer M, so it can later report them to A. When A's ENDEPOCH message arrives, B computes a Merkle Hash Tree [49] (MHT) over the hashes in the epoch buffer, extracts the top-level hash h_0, writes an entry $s_B :=$ EPOCHLOCAL$(A \rightarrow B, i, h_0)$ to its tamper-evident log (see Section 4.3), and returns a message COMMIT(i, a_B, CHAIN(a_B, s_B), M) back to A. a_B is an *authenticator* (defined in Section 4.3), and CHAIN(a_B, s_B) is a hash chain that connects a_B to s_B. (This is used to enable audits later on.) By sending this message, B commits to having received the packets in its epoch buffer.

Agreement: While B is working on its commitments, A continues to forward packets, and it records the corresponding hashes in other epoch buffers to avoid being stalled. However, once the COMMIT message arrives, A locates the corresponding buffer, removes the packets with sequence numbers in M, and then computes a MHT over it in the same way as B, which should yield the same top-level hash h_0. A then records an entry $s_A :=$ EPOCHREMOTE$(A \rightarrow B, i, h_0, a_B,$ CHAIN$(a_B, s_B))$ in its local tamper-evident log and returns a CONFIRM(i, a_A, CHAIN(a_A, s_A)) message to B, which records a FINAL(i, a_A, CHAIN(a_A, s_A)) entry in its log. At this point, A and B have agreed on the set of packets that have been sent over the link in this epoch, and they both hold evidence of this fact (the authenticators and hash chains) in their respective logs. Note that this does not attempt to make packet transmissions reliable, but merely enables the endpoints to agree on the set of correctly transmitted packets.

Actions: Nodes not only have to remember each packet they received, but also what happened to it, so that it can be tracked down a path. SPP represents this information as 1) a time offset to the beginning of the epoch to indicate the time when the packet was received; 2) a set of links to which the packet was forwarded (i.e., to capture both unicast and multicast protocols), if any; and 3) for each such link, a list of rule identifiers and mutations that were applied. Such information is collected in *action buffers* that are "parallel" to the epoch buffers. The nodes commit to their actions by building an MHT over the action buffer, just as it does for the epoch buffers; and the top-level hash h_0^a is recorded in an entry ACTION$(A \rightarrow B, i, h_0^a)$ in its tamper-evident log, just after the EPOCHLOCAL entry.

SPP uses the action buffers to produce the RULE vertexes that link a RECEIVE(p) vertex to its SEND(p_i') vertex. This

is crucial because some nodes can apply mutations to packets in transit: for instance, a NAT will change the port numbers and IP addresses in the header, and many routers will decrement the TTL field. In these cases, the hash $H(p'_i)$ of the forwarded packet will differ from the hash $H(p)$ of the packet that was received. But given the recorded actions, an auditor whose view includes the RULE vertexes can reapply them to p and verify whether $H(p'_i)$ is the correct hash.

Epoch faults: If any of the required messages does not arrive, or if A and B compute different top-level hashes, they report this as an *epoch fault* to their local administrator, e.g., by incrementing an SNMP counter. Absent link failures and attacks, epoch faults can only occur due to undetected packet corruption that has not been handled at the MAC layer, or due to loss of control packets (which could be avoided with extra FEC on these packets, or by sending control packets multiple times.) Therefore, a non-trivial number of epoch faults suggests either a link failure or an attack, and should be investigated immediately by an administrator.

4.3 Tamper-evident log

To prevent nodes from "changing history" and from presenting different views of their history to different auditors, each node maintains an append-only *tamper-evident log* (TEL) [28]. The TEL consists of entries of the form $s_i := (t, h_i, \mathcal{E}, c)$, where t is a timestamp, $\mathcal{E}$ is an entry type, c is the content of the entry, and $h_i := H(h_{i-1} \,||\, t \,||\, \mathcal{E} \,||\, H(c))$ forms a hash chain of the entries. SPP has nine entry types:

- RULEADD(r, R) / RULEDEL(r): A rule R with rule ID r was added or deleted;
- LINKUP(l) / LINKDOWN(l): Link l went up or down;
- EPOCHLOCAL(l, E, h_0) / EPOCHREMOTE(l, E, h_0, a, c): The top-level hash of the local/remote MHT for an epoch E on link l was h_0. a and c are the remote node's authenticator and hash chain.
- FINAL(l, E, a, c): The final authenticator and hash chain for epoch E on link l were a and c, respectively.
- ACTION(l, E, h_0^a): The top-level hash of the action buffer for an epoch E on link l was h_0^a.
- CHECKPOINT(C): C contains a snapshot of the node's current link statuses and rules.

The TEL can be used to authenticate past entries as follows. Recall from Section 4.2, any node A can commit to the contents of its TEL up to some entry s_k by sending an *authenticator* $a_k := (k, h_k, \sigma_A(k \,||\, h_k))$ to another node. If A ever tampers with a previously recorded entry s_j, $j \leq k$, this change will invalidate the hash values of all subsequent entries and be inconsistent with a_k, as well as any other authenticators that the node has sent since s_j. Therefore, suppose that A wants to prove to B that an entry s_j was part of the log that was authenticated by a_k, $k \geq j$. Then A can provide a hash chain CHAIN(a_k, s_j) that consists of $(t_x, \mathcal{E}_x, H(s_x)), j < x \leq k$; using this information, B can recompute $h_j, h_{j+1}, \ldots, h_k$; if h_k matches the value in a_k and a_k is properly signed with A's secret key, B can be sure that the claim is valid [28].

The TEL has two other uses. First, it can be used to reconstruct previous states, e.g., a rule that was used in some past epoch, by loading the most recent checkpoint before that epoch and replaying all the subsequent actions until the epoch of interest has been reached. Second, SPP does not require synchronized clocks across the network; the EPOCHLOCAL and EPOCHREMOTE entries in the TEL provide a form of timeline entanglement [47], which limits how much a compromised node can distort the timing of events to the length of a single epoch.

Notice that the data in the TEL is needed to respond to queries; if a node's TEL is lost or corrupted, that node will no longer be able to respond and thus will (appropriately) register as faulty. However, the loss of the TEL will also prevent a more detailed diagnosis. If this is undesirable, the system can maintain replicas of the TEL.

4.4 Query processing protocol

We now describe how to query the evidence in the TELs.

Querier: A can query the fate of some packet p it has previously sent to B as follows. A scans its epoch buffers for the hash $H(p)$ and identifies 1) the epoch i in which p was sent, 2) its index j, and 3) the commitment $c := \sigma_B(A \rightarrow B \,||\, i \,||\, h_0)$ with which B has acknowledged p's receipt. A then constructs a containment proof PROOF$(c, H(p))$, which shows that $H(p)$ is a leaf node in the MHT rooted at h_0, and invokes QUERY(p) on B with the tuple $u := (c, \text{PROOF}(c, H(p))$.

Responder: When B receives the query, it first verifies that the provided commitment is genuine. If so, it uses the epoch number and the link identifier in c to locate the corresponding action buffer, which will tell B the rule that it has applied to p, and which link(s) p was forwarded to. Finally, B provides the following response: 1) for each link $B \rightarrow C$ to which p was forwarded, the commitment $c_C := \sigma_C(B \rightarrow C \,||\, j \,||\, h_0^{C,j})$; 2) the new hash $H(p')$; 3) a containment proof PROOF$(c_C, H(p'))$; 4) the relevant entry s_p in the action buffer, and 5) a containment proof PROOF$(c_C, H(s_p))$. 1)–3) give A all it needs to invoke QUERY on the next hop C (or, if the packet was cloned, on each next hop), and to generate the SEND and RECEIVE vertexes for the next-hop link(s); 4) and 5) allow A to apply the mutations in s_p and verify that p' is the same packet as p.

4.5 Retroactive freezing protocol

So far, we have explained SPP as if each node kept all of its epoch buffers forever. In practice, SPP allows each node to expire old epoch buffers after some time T_E, while ensuring that malicious nodes cannot discard their buffers freely to cover their tracks, and that normal queries are given enough time to complete.

System	Goal	Information offered	Capabilities				
			Secure evidence	Supports forensics	Covers entire Internet	Fine-grained entities	Fine-grained traces
Tulip [45]	Fault localization	Loss, delay, reordering	×	×	✓	✓(Routers)	✓(Packets)
NetPolice [70]	Traffic differentiation detection	Loss	×	×	✓	× (ISPs)	× (Flows)
SPIE [62]	IP traceback	Backward routes	×	✓	✓	✓(Routers)	✓(Packets)
NetSight [29]	Network debugging	Packet histories	×	✓	×	✓(Routers)	✓(Packets)
Netdiff [46]	ISP performance benchmarking	Delay	×	×	✓	× (ISPs)	✓(Packets)
Paris-traceroute [11]	Load-balancer detection	Load-balanced routes	×	×	✓	✓(Routers)	✓(Packets)
HAL [26]	Packet attestation	Packet transmissions	✓	✓	×	✓(Links)	✓(Packets)
AudIt [9]	Performance accountability	Loss, delay	×	×	✓	× (ISPs)	✓(Both)
SPP	Single network-level primitive	All of the above	✓	✓	✓	✓(Routers)	✓(Packets)

Table 1: Comparison between SPP and some existing diagnostic and forensic primitives.

SPP uses a retroactive freezing protocol, where a node A can request that the evidence for a packet p be frozen into stable storage, so that it can be inspected on human timescales. A does so by sending a special *freeze packet* $p' = \text{FREEZE}(H(p))$ on the same port as p before T_E elapses. p' and p maintain the same header so that they will likely take the same path. But if path divergence happens or p' gets dropped, SPP can be recursively applied to p' to investigate such instances. Moreover, the freeze packet is sent *retroactively*, up to several seconds after the packet was originally sent, so that a compromised node cannot predict which packets will be frozen, and then treat these packets differently to avoid detection; by the time the freeze packet arrive, the nodes will have forwarded the packets and committed to their actions. The end users can choose which packets to freeze according to their needs, or randomly freeze a subset of their traffic. To prevent the freeze primitive from being abused (e.g., for DoS attacks), nodes can limit the rate at which they are willing to freeze packets: if a node receives too many freeze requests from a neighbor, it can record the requests and the corresponding commitment, and then deny the excess request. If that node is challenged later because it did not respond to a request, it can show the saved requests to prove its innocence.

4.6 Properties

Next, we discuss the properties of SPP. In the presence of Byzantine nodes, the provenance graph G_e constructed from the collected evidence e is only an approximation of the "actual" provenance graph G; for example, a faulty node may refuse to provide an explanation consistent to e in response to a QUERY request. However, G_e is a close enough approximation of G, providing the following guarantees:

- G_e is **accurate**. G_e faithfully reproduces all the vertexes on correct nodes, that is, 1) if a vertex v on a correct node exists in G_e, then v must also exist in G, with the same predecessors and successors; and 2) a correct node will never be accused as faulty.
- G_e is **complete**. Given evidence e from correct nodes, 1) each vertex in G on a correct node also appears in G_e, and 2) when some node is detectably faulty, recursive QUERY invocations will identify at least one faulty node.

In other words, although we cannot force faulty nodes to cooperate, SPP will always generate provenance that reflects the actual execution of all correct nodes, and SPP can correctly expose at least one faulty node with non-repudiable evidence.

In terms of privacy guarantees, a node is not allowed to audit or otherwise learn about packets it has not processed:

- G_e is **private**. Given an evidence e collected through recursive QUERY, G_e constructed by node v contains only SEND and RECEIVE vertexes for packets that are *visible* to v. We say a packet p is *visible* to a node v, if 1) p is received or sent by v, 2) p is mutated and forwarded as p' that is visible to v, or 3) a visible packet p' is mutated and forwarded as p.

4.7 Limitations

SPP is designed for diagnostics and forensics on the Internet's data plane, and there are at least three classes of problems that SPP cannot diagnose directly: a) faults of a remote node that do not affect any external messages, such as CPU overload; b) faults that happen outside of the Internet data plane, such as BGP prefix hijacking; and c) faults that need aggregate information about the packets, such as high per-flow latencies. Next, we explain these categories in more detail, and discuss potential ways of addressing some of them.

Non-observable faults: Not all problems on a node can be detected from only its externally visible inputs and outputs. For instance, if a bit flips in a node's memory, its CPU load is high, or its disk has failed, the network packets that the nodes sends may not be affected initially (or ever). Even if the problem does affect a network packet, detection may still be impossible if the nodes that receives the packets is also faulty. This limitation is inherent [27] and also affects other systems that attempt to detect or diagnose faults based on network events.

Control-plane diagnostics: SPP, as described here, generates an IP-level provenance graph on the Internet's *data plane*; it does not provide visibility into control-plane events. Thus, SPP's QUERY primitive cannot detect faults that manifest entirely on the control plane, such as BGP prefix hijacking, routing policy violations, and the like. This limitation is not inherent, and it should be possible to remove it

by extending the provenance model to capture control-plane events, as discussed in Section 3.4.

Aggregate information: SPP's QUERY primitive returns information about individual packets, so it cannot directly diagnose problems that are related to aggregate properties of multiple packets or entire flows. For instance, if a flow is experiencing low throughput, this cannot be detected based on what happened to individual packets in that flow. One way to get around this would be to implement the summarization extension from Section 3.4; an even simpler way would be to query multiple packets and to do the analysis as a post-processing step. The precision of the second approach would be limited by the accuracy of the nodes' timestamps (recall that SPP does not assume synchronized clocks); however, previous work has shown [10] that useful performance measurements are possible even when the clocks are only loosely synchronized.

5. Case studies

Next, we describe four classes of common diagnostic and forensic tasks for which specialized solutions already exist. We explain how SPP can approximate these solutions, and how they could be re-implemented on top of SPP. Table 1 provides a summary.

Traceback: Traceback is the process of identifying the sender of a (potentially spoofed) packet. This is difficult in the current Internet because packets contain no secure data about their source or the paths they have traversed. Source authentication systems like AIP [7] and Passport [43] aim to prevent spoofing using cryptographic signatures. Path verification systems aim to reconstruct a packet's path, e.g., by securely recording it in a header [53], by probabilistically marking packets [59], or by keeping digests of packets at each router [62]. Both types of traceback essentially require access to the *path a received packet has taken*, which is a part of the packet-level provenance that SPP offers (although SPP does not proactively prevent spoofing).

Routing and performance problems: A common diagnostic task is to determine why a particular path has unusually high packet loss, delay, or has become unavailable [11, 41, 64]. To overcome the limited visibility deep within the network, proposals have been made to extract more diagnostic information, e.g., by using more vantage points [36], adding network extensions [10, 45, 56], or using historical data [19, 62]. NetSight [29] even remembers packet "histories" that are similar to the provenance in SPP (though in an intra-domain setting). In essence, those systems want to know the *path a transmitted packet has taken*, along with some timing information for each hop. SPP exposes a superset of the information needed: packet-level properties are visible directly; flow-level properties can be extracted by some post-processing on a set of frozen packets.

Intrusions and misbehavior: Internet-related evidence is appearing in many court cases, but forged packets and IP addresses can lead to judicial errors [26] and bogus actions [55]. One possible solution is to enable the use of packet traces as secure *evidence* using source or packet authentication. AIP [7] and Passport [43] provide the former, and HAL [26] provides the latter; ICING [53], Clue [6], and DRKey [38] support both. In some cases, the ISPs themselves have an incentive to manipulate unwanted traffic [18] or to inject advertisements [58]. Systems to detect such misbehaviors include, e.g., Glasnost [18] and NetPolice [70] that detect traffic differentiation, and Web Tripwires [58] that detects in-flight packet modifications. However, to ultimately resolve such situations, one also needs *evidence*: since the recipient of the packet (the victim) is usually different from the entity that takes action (e.g., a judge), it is necessary to verify that a particular evidence is authentic. SPP's authenticators are designed for this purpose.

Topology discovery: Topology mapping is useful for latency prediction and modeling [60]. However, in the absence of direct support, people must generally rely on low-quality data, e.g., from traceroutes or IP record-route options, which require great ingenuity to collect and clean up. It would be much easier if the network provided *explicit and unambiguous information*, so that there would be no need for "guesswork" based on subtle idiosyncrasies of network hardware.

6. Implementation

We have implemented two prototypes of SPP: a software-only implementation of the entire system, and a NetFPGA prototype of the parts that would need to run at line speed.

Software prototype: Our software prototype is written in C/C++. It can be configured to run 1) as a Click router module [39], or 2) as a standalone program. In Click mode, SPP runs with live traffic forwarding; we performed functionality checks and prototyped six common diagnostic routines in this mode. In standalone mode, SPP still runs the entire protocol but disables traffic forwarding; we used this mode to evaluate SPP's protocol overhead as a very conservative lower-bound. Both modes are trace-based, so they are not limited by the speed of our physical NICs. We used SHA-1 for the hash function[1] and RSA-2048 for the cryptographic signatures, as implemented in the OpenSSL library v1.0.1.f. Our Click mode implementation is based on Click v2.1.

NetFPGA prototype: As we will show in Section 7, the dominating cost in SPP comes from packet hashing and MHT construction. To evaluate its performance in realistic deployment, we have built an additional implementation of those two components in hardware, on a NetFPGA-10G [13] platform. Our platform contains a Xilinx Virtex 5 (65nm) FPGA (xc5vtx240tffg1759-2 [32]), as well as four SFP+ modules that can each support 10 Gbps traffic. We have implemented SPP as part of the Output Port Lookup module

[1] After the recent discovery of a collision [63], SHA-1 is no longer considered secure. Future implementations should use a more recent hash function, such as SHA-256.

(somewhat analogous to the design in [52]), so that it can run in parallel with the traffic forwarding path. Our logic is divided into 13 fully pipelined stages. The first stage contains a state machine that parses packets from NetFPGA's AXI4-Stream interconnects; the second stage computes per-packet hashes; and the remaining 11 stages construct the MHTs. Our implementation builds on NetFPGA and open-source hashing libraries, and consists of 2,588 lines of Verilog code.

The first stage routes 64 bits of packet data per cycle from the AXI4-Stream interconnects, so it can send a minimum-sized packet to the hashing stage every eight cycles. Our hasher also accepts 64 bits per cycle, but it incurs a 14-cycle delay after the packet's last-bit signal is asserted. To nevertheless keep up with the incoming data rate, the hashing stage contains four separate instances of the hasher and uses them in a round-robin fashion. Each of the MHT stages consists of a buffering phase and a hashing phase: the buffering phase uses a fallthrough FIFO in SRAM to hold the hashes produced by the previous stage, and the hashing phase dequeues hashes from its FIFO, hashes them in pairs, and then enqueues the new hash at the next FIFO. The last stage's hasher produces the MHT roots. Since the data rate decreases as hashes pass through the MHT stages, we are able to do rate matching using 15 hashers: four for the first MHT stage, two for the second MHT stage, and one per each of the remaining stages. We have used SHA-3 (Keccak) in the hardware implementation for its good performance; to make the results comparable to those from the software prototype, we use only the last 160 bits to match the length of SHA-1.

7. Evaluation

In this section, we evaluate SPP's performance overhead and demonstrate how common diagnostic functionalities can easily be built with it. We first evaluate SPP's protocol overhead with our software prototype, including storage, bandwidth, and computation costs, both with *real high-speed traffic*, and in *worst-case scenarios*; in addition, we report our hardware microbenchmarks to show that the seemingly high computation cost in software could be easily handled by off-the-shelf hardware technology. (Note that the storage and bandwidth overheads, unlike the computation cost, would not differ across hardware and software platforms.)

We obtained our real traffic from CAIDA's live capture on a 10 Gbps OC-192 link on Jan. 19, 2012, in which 4.6 million packets were sent with an average rate of 2.46 Gbps. For the worst-case scenarios, we synthesized traffic at 100 Mbps, 1 Gbps, and 10 Gbps in which all packets have the minimum size, and thus the traffic has the maximum packet rate (which is unlikely to occur with real traffic). We also used an epoch length of $T = 100$ms, and 10-bit sequence numbers in the link-layer headers, allowing the numbers in the header to wrap: the full sequence number can be reconstructed as long as loss bursts are below 2^{10}. Our software experiments were run on a Dell OptiPlex 9020 workstation, which has a 3.40 GHz Intel i7-4770 CPU (with 8 cores), 16 GB of RAM, and a 500 GB hard disk. The OS was Ubuntu 14.04 with kernel version 3.8.0.

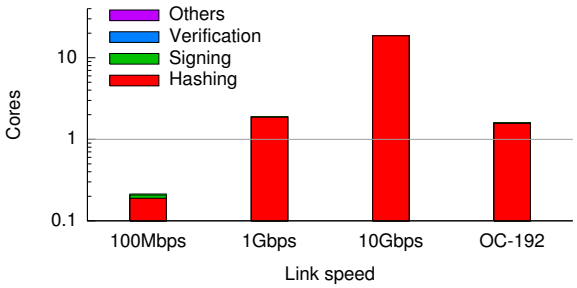

Figure 2: Computation cost of SPP's commitment protocol, normalized to the power of one core. The cost of hashing dominates. (The other bars are too low to see.)

7.1 Recording: Computation cost

SPP requires each network component to regularly generate commitments for the traffic it sends and receives, and to verify its neighbors' commitments. We first used our software prototype to quantify this cost. We generated synthetic traces that consisted entirely of 40-byte packets (the smallest valid TCP packet, 84 bytes on the wire [35], and thus the worst case for SPP) with rates of 100 Mbps, 1 Gbps, and 10 Gbps. We then ran all four traces through our software prototype, measured the computation time to generate and verify the commitments, and normalized the cost to the performance of an individual core. For instance, if one core took 2 seconds to process the commitments for packets sent in 1 second, we report this as 2 cores. (SPP trivially scales to multiple cores, as the cores can work on different epochs independently.) We report a decomposition of the cost of hashing, signature generation, and signature verification.

Figure 2 shows that the dominant computational cost of SPP is hashing, especially at higher link speeds. This is good news because hashing is easy to do in hardware [22, 54], and it is also the reason why our NetFPGA prototype focuses on hashing: the remaining computations have a moderate cost, so routers should be able to perform them in software. Figure 3 shows results from a similar experiment where the two highest-cost traces still maintain the same rates, but have different packet sizes (and 14-byte Ethernet headers). The figure shows that the overhead drops quickly as the packet size increases. This is because the number of internal hashes in the MHT depends only on the number of packets, but not on their size. At a more typical packet size of 300 bytes [61], the cost is 54% lower.

Hardware prototype: For our NetFPGA prototype, computation cost is not a good metric; instead, we quantify the maximum supported bitrate and the number of hardware elements that it requires. Our NetFPGA prototype can be synthesized to run at 200 MHz (5 ns per clock cycle), which achieves a theoretical throughput of 12.8 Gbps, and an effective throughput of 10 Gbps with the existing SFP+ mod-

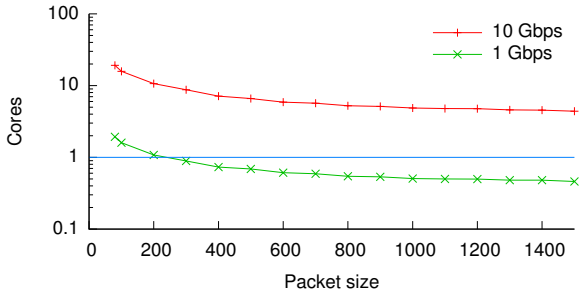

Figure 3: Computation cost for different packet sizes in the two traces with the highest costs.

Resource	Used	Total available	Utilization
Slice registers	53,964	149,760	36%
Slice LUTs	109,040	149,760	72%
Block RAMs	28	324	8%

Table 2: Hardware cost for hashing and building MHTs.

ules. We note that these results are consistent with other benchmarking efforts [12, 21, 48]. We have listed the hardware utilization in Table 2, in terms of LUTs (LookUp Tables), registers, and Block RAMs used. They are well within the resources available on Virtex-5 FPGAs, and would be only a fraction if mapped on more recent FPGAs: for instance, NetFPGA-SUME's Virtex-7 has nearly three times as many logical elements [76]. We also note that 10 Gbps is not the limit: for 100 Gbps routers, there are optimized hashers that could achieve 34.27 Gbps per hasher on Virtex-5 FPGAs [50], which is about ten times faster than the hash module we have used. The performance of our hardware prototype represents a lower bound on the performance that a hardware implementation of SPP can achieve; in a real-world deployment, the packet processing would be performed on ASICs in high-speed routers, which are much faster than FPGAs.

7.2 Recording: Bandwidth cost

Since SPP's bandwidth and storage overheads do not vary with the underlying hardware or software platforms - unlike the computation speed - we evaluated them on our software prototype. SPP requires an extra link-layer header, as well as some new control messages for exchanging commitments. Both consume some fraction of the raw link capacity that is no longer available for sending traffic. To quantify this effect, we measured the fraction of the raw link capacity that was used by SPP. We sent $R = 3$ replicas of each control message, to conservatively account for message loss, and we assumed a link-level packet loss rate of 1%, which is orders of magnitude above typical rates today [1, 3]. We show results for 40-byte packets (the worst case) and a more typical packet size of 300 bytes.

Figure 4 shows our results. For the 100 Mbps trace with 40-byte packets, SPP only consumes about 2.06% of the available link capacity. Moreover, the overhead drops with

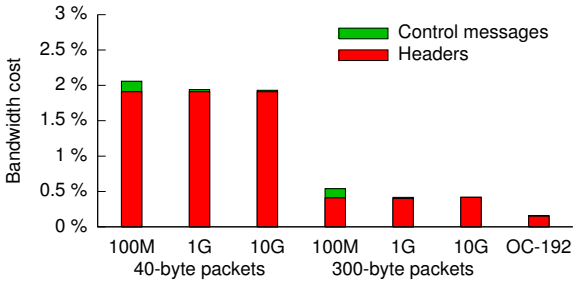

Figure 4: Bandwidth consumption of SPP's commitment protocol, as a fraction of the raw link capacity.

increasing link speeds and increasing packet size. This is because the overhead has two components: one consists of three fixed-size messages (NEWEPOCH, ENDEPOCH, and CONFIRM) that are sent once per epoch, regardless of the link speed and the number of messages, and the other consists of the link-layer headers and the entries in the missing packet list (COMMIT), which are both proportional to the number of packets. At 1 Gbps and with the more typical 300-byte packets, the overhead is only 0.42%. For the most realistic case of the OC-192 link, the overhead is only 0.16%.

7.3 Recording: Memory

SPP requires a certain amount of RAM for epoch buffers, action buffers, and the list of lost packets. Next, we quantify how much memory these data structures require.

In our implementation, an entry in the epoch buffer requires 20 bytes (the size of a hash value), an entry in the action buffer requires 30 bytes (the size of a timestamp, a destination port number, and up to 3 mutation records), and an entry in the loss buffer requires 10 bits (the size of a sequence number). For each received packet, SPP adds one entry to each of the first two buffers, and for each missing sequence number, it adds an entry to the third buffer. We also dimension the buffers for the worst case. If we assume a 1 Gbps link, an epoch length of $T = 100$ms, and a minimum packet size of 40 bytes (i.e., up to 312,500 packets per epoch), the epoch and action buffers would require 6.25MB and 9.38MB of memory, respectively; with a link-level loss rate of 1%, the loss buffer would require 3.91 kB. Since all sizes are proportional to the link speed, a 10 Gbps link would require ten times as much.

The number of buffers depends on the number of ports the node has, and on the latency that is needed to finish the commitment protocol, which depends on the link's RTT. (Recall that the sender must retain the hashes until the receivers' COMMIT message arrives.) If we conservatively assume a per-link RTT of up to 100ms, $2*(100/T) = 2$ buffers would be needed per port, so a node with twenty 1 Gbps ports would need 625 MB of RAM. Note that the hashes are written at much lower rates than the links' bitrates, so expensive SRAM is *not* required – commodity DRAM is enough, e.g., NetFPGA-SUME has 8 GB of DDR3 synchronous DRAM with a 238.8 Gbps peak memory throughput [76].

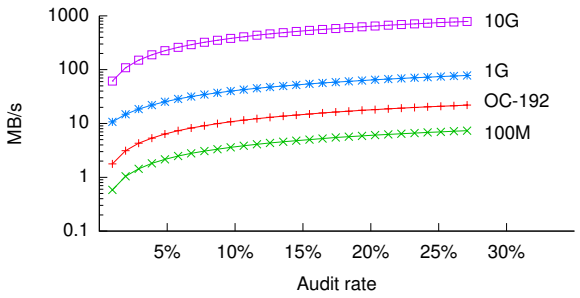

Figure 5: Data rates for different auditing rates ϕ.

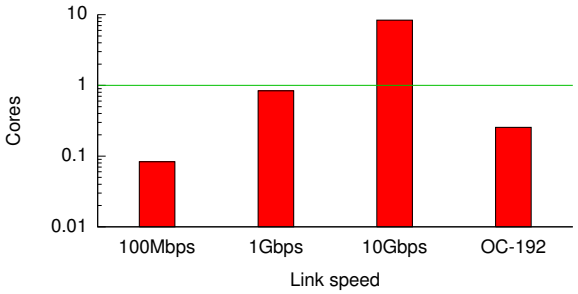

Figure 6: Computation cost for answering queries.

7.4 Recording: Disk space

SPP requires disk space to store the packet-level evidence that has been "frozen" by queriers, as described in Section 4.5. To quantify how much storage is needed, we ran the traces through SPP and randomly froze a certain fraction ϕ of the packets. Figure 5 shows the amount of data written to disk due to audits. It is expected that the amount of frozen evidence increases with ϕ. However, the increase is not linear: for small ϕ, SPP must store not only the hash of each frozen packet but also the hashes of internal nodes along the path to the root. But, as ϕ increases, there is more and more overlap between the paths, which reduces the number of additional hashes that need to be stored for each new frozen packet. Note that the next-hop authenticator needs to be stored only once per epoch, so the necessary space is comparatively small. From the figure we can see that, an auditing rate of $\phi = 1\%$ can be well supported by the throughput of a hard disk, and $\phi = 15\%$ with a commodity SSD. Summarizations (Section 3.4) can further reduce the storage consumption: with flow-level summarization (see [15]), a 100 GB disk can store the summaries from the OC-192 trace for the most recent 25.3 hours, i.e., for more than a day. In order to store even more data, the granularity of the summarization could be reduced further.

7.5 Querying

Computation: Upon a query, SPP must freeze and retrieve the evidence that is needed to answer it. The evidence can be constructed by building a MHT with packets in the queried epoch, and tracing the relevant paths from the root to the queried packets. Querying multiple packets in the same epoch only costs marginally more than querying only one packet from that epoch, because queries for packets in the same epoch can be buffered until the end of the epoch and answered altogether. Therefore, the worst-case cost is when MHTs for all epochs need to be reconstructed. Note that this is a simple repetition of the MHT construction at recording time (Section 7.1), only this time we do not need to hash the packets again. We show the computation cost for different link speeds in Figure 6, and note that our NetFPGA prototype could achieve this at a 10 Gbps rate.

Bandwidth: The bandwidth needed for freezing is low (a single 40-byte packet), so we focus on the bandwidth for retrieving the evidence. Recall that to query the provenance of a packet p on a node n, the querier provide n with p's hash, the number of the epoch p was sent in and the corresponding authenticator, p's index in the epoch buffer, and a containment proof that links the authenticator to the hash; the response contains the same information for the next hop, along with the relevant entry from the action buffer. For a given choice of hash function and signature algorithm, the size of all fields is fixed, except for the containment proof, which grows with the height of the per-epoch MHT. For a 1Gbps link with $T = 100$ms, the size of a single query in our implementation is 680 bytes; for a 10Gbps link, the MHT grows by four levels, and thus the size of a query grows to 760 bytes. Responses are 30 bytes larger because of the additional action buffer entry. Both queries and responses are small enough to fit into a single packet.

We now estimate the *worst-case bandwidth cost* of querying. The cost for a single query (with freeze packet and headers for request and response) is 40+680+710+2×28=1486 bytes, or 4.95 times the average packet size of 300 bytes. Thus, if a node allows up to 1% of its traffic to be queried, query-related packets would account for 4.7% of its traffic.

Core functionality		LoC
Trace a transmitted packet's path	[33]	8
Trace a received packet's traversed path	[62]	8
Identify node on path that drops a packet	[45]	8
Attest to the transmission of a packet	[26]	9
Identify link on path with highest delay	[41]	24
Compute a link's average throughput	[46]	26

Table 3: Several applications we built with SPP, and the lines of code (LoC) they required.

7.6 Comparison with SNP

We next compare SPP with SNP [72], the state-of-the-art system for secure provenance. SNP has been applied to BGP, Chord, and Hadoop, but it is not designed to handle the high data rates on the network data plane. To demonstrate this, we ran SNP and SPP side by side, and we streamed packets through both of them; we report the results we have obtained on the 1Gbps trace with 40B packets.

Disk space: At an auditing rate of 1%, SPP wrote 65% less evidence on disk than SNP. This is because SPP's evidence

```
void tracert(Packet *p, Evidence *e) {
  IP *nextHop = gatewayIP;
  Packet *p0 = p;
  do {
    query(&p, &e, nextHop);
    print(nextHop+" "+(e.time-p0.time));
  } while (nextHop != END_OF_PATH);
}
```

Figure 7: Code for tracing a packet's traversed path.

mostly consists of 20-byte hashes and not the longer RSA signatures that SNP requires.

Bandwidth: SPP consumes 98.2% less bandwidth than SNP. This is because SPP's can commit to a batch of packets using a single root hash, whereas SNP has to commit to each packet one by one.

Computation: On the same trace, SPP runs 1378.5 times faster than SNP. This is because SPP only performs two hashes per packet and one RSA signature per batch, whereas SNP needs to sign every single packet. At this speed, SNP would require the equivalent of more than a thousand CPU cores to process 1 Gbps of traffic in software, whereas SPP can do the same with a single core.

7.7 Building applications with SPP

To determine whether SPP can fulfill its key promise of supporting a wide variety of diagnostic and forensic tasks, we implemented the core functionalities of six diagnostic and forensic systems from the literature on top of the QUERY primitive that SPP provides. Table 3 shows a list of the six systems, along with the lines of code (LoC) in our implementations. The LoC numbers are very low: our most complex application consists of 26 LoC, and four of the six applications have less than 10 LoC. For concreteness, Figure 7 shows the slightly simplified code for tracing the path a transmitted packet p has traversed; the code simply iterates through the sequence of hops, starting with the evidence it received when p was originally sent, and outputs the IPs and latencies it encountered along the way.

The low number of LoC may seem surprising, but the reason is that most of the complexity in the original applications was in the special-purpose network primitives they proposed, or in smart techniques for leveraging and working around existing primitives (such as ICMP TTL Exceeded) that were originally introduced for some other purpose. With SPP in place, the applications we tried reduce mostly to gathering the relevant evidence and/or performing some simple post-processing. Therefore, SPP does deliver its key benefit: a *single* primitive that can handle most existing – and hopefully future – diagnostic and forensic tasks.

8. Deployment

As with most extensions to the Internet architecture, getting SPP deployed at scale would not be easy. However, SPP has a number of properties that could help to facilitate its deployment. We summarize these properties below; for more details, please refer to our full paper [15, Appendix B].

One primitive, many applications: As we have argued, it should be possible to implement a variety of existing diagnostic and forensic systems on top of SPP. Thus, although deploying any new feature in the data plane would not be cheap, at least this effort would have to be spent only once (rather than once for each specialized solution), and it would yield a solution for a wide range of problems.

Few changes to the protocol stack: SPP leaves the current protocol stack (almost) untouched; it mostly "sits on the sidelines" and collects information about the traffic it observes. (The one change it does require is the additional link-layer header for the commitment protocol, which is only visible to the routers on that particular link.) Thus, SPP requires much fewer changes than a design that introduces a new kind of addresses [7] or major packet header changes [53].

Using existing routers: Deploying SPP does not necessarily require new routers. If a link does not have SPP-enabled routers, it can still use SPP by attaching a separate machine of FPGA (i.e., an SPP proxy) to each endpoint of the link, and mirroring all traffic to them. The proxies can make commitments and enable audits, while the routers themselves forward traffic as usual. If a link has an SPP-enabled router on one side and a conventional router on the other, a similar proxy could be used to pair with the SPP-enabled router.

Incremental deployments are useful: SPP can be usefully deployed at an individual ISP to diagnose ISP-local problems, so there is no need for a global "flag day". Its benefits increase gradually with the size of the deployment: This is very different from a protocol like S-BGP, which must be deployed almost universally to be useful.

Efficient hardware implementation: Like most secure diagnostic primitives, SPP requires at least some cryptographic operations. However, its fast path mostly requires hashing, which can be implemented efficiently in hardware.

8.1 Incentives

We believe that ISPs have at least two incentives to deploy SPP. First, SPP can handle troubleshooting tasks that are directly useful to an individual ISP, even without considering the rest of the Internet. Thus, each ISP would initially have at least some incentive to deploy SPP in its own network, independent of what everyone else is doing. SPP is perhaps more heavyweight than a specialized troubleshooting tool that serves a single purpose, but SPP has a wide variety of uses, so an ISP might be willing to shoulder some additional cost for the extra flexibility.

Second, SPP can help with cross-domain fault localization, which is notoriously difficult. For instance, suppose two adjacent ISPs cannot agree whether a path performance problem lies in one ISP or the other. Today, this situation might involve long phone calls between the ISP operator teams, and potentially some customer dissatisfaction on both

sides. This creates a triple incentive to deploy SPP: 1) ISPs might prefer to peer with networks that support SPP, to better diagnose problems in these networks; 2) ISPs might adopt SPP in their own network to distinguish themselves from competitors and to highlight their own reliability; and 3) ISPs might adopt SPP to quickly establish that the problem is *not* on their side.

Ideally, these incentives would initially lead to the formation of "deployment islands", which would then slowly grow until they start to merge. In the process, the fraction of a typical path that would be covered by SPP would slowly increase, leading to better and better end-to-end diagnostics.

9. Related Work

Specialized primitives: As discussed in Section 2, there is a rich literature on systems that solve a *particular* diagnostic or forensic problem [7, 10, 11, 18, 19, 26, 29, 36, 41, 43, 45, 51, 53, 58, 59, 62, 64, 70]. To address all of the underlying problems, it would be necessary to deploy all of these systems in combination. In contrast, SPP aims to provide a single primitive that can be used for a broad variety of tasks.

Packet-level diagnostics: SPIE [62], HAL [26], and NetSight [29] resemble SPP in that they all "remember" every single network packet. However, SPIE cannot reliably identify a specific packet due to the use of Bloom filters, and HAL only collects per-packet evidence but does not perform diagnosis. NetSight [29] is closest to SPP: it assembles a "history" of each packet for SDNs. However, NetSight provides no security guarantees in the presence of compromised nodes, and it is designed for a data-center setting, where packet traces can be recorded without privacy concerns and data does not need to be shared with other domains. UnivMon [44] and OpenSketch [68] are recent proposals for flow-level monitoring counters based on sketches. These approaches are useful for gathering traffic statistics, but, unlike SPP, they do not provide packet-level provenance data.

Provenance: The concept of data provenance originated in the database literature [14], and has since found a number applications in the networking domain [16, 67, 72, 75]. Among these, SNP [72] is the only one that is suitable for adversarial settings. However, SNP's overhead would be enormous if it were applied to high-speed data, and it also lacks the privacy protections that are needed for wide-area diagnostics across multiple domains.

Accountability: SPP is similar in spirit to previous proposals for network-level accountability, e.g., "packet obituary" [8] that reports packet drops, or AudIt [9] that provides secure records of delay and loss rates. Network Confessional [10] uses a similar retroactive sampling approach to prevent special treatment of the sampled packets; however, it focuses on forwarding performance verification, not direct support for diagnostics or forensics. PAAI [69] also uses retroactive sampling to track lost packets, but it assumed that end hosts are always honest.

10. Conclusion

As the large number of proposed extensions shows, the current Internet architecture does not support diagnostics and forensics very well. However, most existing proposals are specialized solutions; thus, a comprehensive solution would require deploying several of them concurrently, at a substantial cost – in terms of both overhead and complexity. In this paper, we have made a case for a network-level primitive that can support a variety of different diagnostic and forensic applications, and we have also presented SPP as a concrete proposal. Our evaluation shows that SPP can be implemented efficiently in hardware and can approximate a variety of common diagnostic and forensic tasks.

Acknowledgments

We thank our shepherd, Dejan Kostić, and the anonymous reviewers for their thoughtful comments and suggestions. We are also grateful for the feedback we received from André DeHon, Albert Kwon, Christopher W. Fletcher, Ling Ren, and Sizhuo Zhang on the FPGA prototype. This work was supported in part by NSF grants CNS-1054229, CNS-1065130, CNS-1218066, CNS-1453392, CNS-1513679, and CNS-1513734.

References

[1] NTT SLA. http://www.ntt.net/english/service/sla_ts.html.

[2] Police face £750k bill for false Operation Ore charges. http://www.telegraph.co.uk/technology/news/8422200/Police-face-750k-bill-for-false-Operation-Ore-charges.html.

[3] Sprint SLA. https://www.sprint.net/sla_performance.php.

[4] Symantec says hackers tried extortion. http://bits.blogs.nytimes.com/2012/02/07/symantec-says-hackers-tried-extortion/.

[5] Techie lands in jail due to Airtel, sues it. http://ibnlive.in.com/news/techie-lands-in-jail-due-to-airtel-sues-it/101343-3.html.

[6] M. Afanasyev, T. Kohno, J. Ma, N. Murphy, S. Savage, A. C. Snoeren, and G. M. Voelker. Privacy-preserving network forensics. *Commun. ACM*, 54(5):78–87, May 2011.

[7] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet protocol (AIP). In *Proc. SIGCOMM*, 2008.

[8] K. Argyraki, P. Maniatis, D. R. Cheriton, and S. Shenker. Providing packet obituaries. In *Proc. HotNets*, 2004.

[9] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker. Loss and delay accountability for the Internet. In *Proc. ICNP*, 2007.

[10] K. Argyraki, P. Maniatis, and A. Singla. Verifiable network-performance measurements. In *Proc. CoNEXT*, 2010.

[11] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proc. IMC*, 2006.

[12] B. Baldwin, N. Hanley, M. Hamilton, L. Lu, A. Byrne, M. Neill, and W. P. Marnane. FPGA implementations of the round two SHA-3 candidates. In *Proc. Second SHA-3 Candidate Conference*, 2010.

[13] M. Blott, J. Ellithorpe, N. McKeown, K. Visssers, and H. Zeng. FPGA research design platform fuels network advances. *Xilinx Xcell Journal*, 2010.

[14] P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *Proc. ICDT*, 2001.

[15] A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. One primitive to diagnose them all: Architectural support for Internet diagnostics. Technical Report MS-CIS-17-04, University of Pennsylvania, 2017.

[16] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proc. SIGCOMM*, 2016.

[17] D. Clark. The design philosophy of the DARPA Internet protocols. *ACM Computer Communication Review*, 18(4):106–114, 1988.

[18] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling end users to detect traffic differentiation. In *Proc. NSDI*, 2010.

[19] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Netw.*, 9(13):280–292, 2001.

[20] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proc. SIGCOMM*, 2004.

[21] K. Gaj, E. Homsirikamol, and M. Rogawski. Comprehensive comparison of hardware performance of fourteen round 2 SHA-3 candidates with 512-bit outputs using field programmable gate arrays. In *Proc. Second SHA-3 Candidate Conference*, 2010.

[22] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif. Comprehensive evaluation of high-speed and medium-speed implementations of five SHA-3 finalists using Xilinx and Altera FPGAs. https://eprint.iacr.org/2012/368.pdf.

[23] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *ACM Computer Communication Review*, 39(1):68–73, Dec. 2008.

[24] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proc. NSDI*, 2009.

[25] A. Haeberlen, M. Dischinger, K. P. Gummadi, and S. Saroiu. Monarch: A tool to emulate transport protocol flows over the Internet at large. In *Proc. IMC*, 2006.

[26] A. Haeberlen, P. Fonseca, R. Rodrigues, and P. Druschel. Fighting cybercrime with packet attestation. Technical Report MPI-SWS-2011-002, Max Planck Institute for Software Systems, July 2011.

[27] A. Haeberlen and P. Kuznetsov. The fault detection problem. In *Proc. OPODIS*, 2009.

[28] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, 2007.

[29] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. NSDI*, 2014.

[30] S. Hao, M. Thomas, V. Paxson, N. Feamster, C. Kreibich, C. Grier, and S. Hollenbeck. Understanding the domain registration behavior of spammers. In *Proc. IMC*, 2013.

[31] R. Hasan, R. Sion, and M. Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In *Proc. FAST*, 2009.

[32] X. Inc. Virtex-5 family overview. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, Feb. 2009.

[33] V. Jacobson. Traceroute. ftp://ftp.ee.lbl.gov/traceroute.tar.gz.

[34] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proc. NDSS*, 1999.

[35] Juniper Networks. Packets per second. http://kb.juniper.net/InfoCenter/index?page=content&id=KB14737.

[36] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. Van Wesep, T. Anderson, and A. Krishnamurthy. Reverse traceroute. In *Proc. NSDI*, 2010.

[37] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *Proc. NSDI*, 2008.

[38] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig. Lightweight source authentication and path validation. In *Proc. SIGCOMM*, 2014.

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

[40] M. Kotadia. Trojan horse found responsible for child porn. ZDNet, 8/1/2003.

[41] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao. Moving beyond end-to-end path information to optimize CDN performance. In *Proc. IMC*, 2009.

[42] M. Liberatore, B. N. Levine, and C. Shields. Strengthening forensic investigations of child pornography on P2P networks. In *Proc. CoNEXT*, 2010.

[43] X. Liu, A. Li, X. Yang, and D. Wetherall. Passport: Secure and adoptable source authentication. In *Proc. NSDI*, 2008.

[44] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *Proc. SIGCOMM*, 2016.

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

[46] R. Mahajan, M. Zhang, L. Poole, and V. Pai. Uncovering performance differences among backbone ISPs with Netdiff. In *Proc. NSDI*, 2008.

[47] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proc. USENIX Security*, 2002.

[48] S. Matsuo, M. Knezevic, P. Schaumont, I. Verbauwhede, A. Satoh, K. Sakiyama, and K. Ota. How can we conduct fair and consistent hardware evaluation for SHA-3 candidate? In *Proc. Second SHA-3 Candidate Conference*, 2010.

[49] R. Merkle. Protocols for public key cryptosystems. In *Proc. IEEE S&P*, 1980.

[50] H. E. Michail, L. Ioannou, and A. G. Voyiatzis. Pipelined SHA-3 implementations on FPGA: Architecture and performance analysis. In *Proc. CS2*, 2015.

[51] A. Mizrak, S. Savage, and K. Marzullo. Detecting compromised routers via packet forwarding behavior. *IEEE Network*, 22(2):34–39, 2008.

[52] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the NetFPGA platform. In *Proc. ANCS*, 2008.

[53] J. Naous, M. Walfish, A. Nicolosi, D. Mazières, M. Miller, and A. Seehra. Verifying and enforcing network paths with ICING. In *Proc. CoNEXT*, 2011.

[54] D. Naylor, M. K. Mukerjee, and P. Steenkiste. Balancing accountability and privacy in the network. In *Proc. SIGCOMM*, 2014.

[55] M. Piatek, T. Kohno, and A. Krishnamurthy. Challenges and directions for monitoring P2P file sharing networks. In *Proc. HotSec*, 2008.

[56] A. Ramachandran, K. Bhandankar, M. B. Tariq, and N. Feamster. Packets with provenance. In *Proc. SIGCOMM Poster*, 2008.

[57] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proc. SIGCOMM*, 2006.

[58] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *Proc. NSDI*, 2008.

[59] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proc. SIGCOMM*, 2000.

[60] R. Sherwood, A. Bender, and N. Spring. DisCarte: A disjunctive Internet cartographer. In *Proc. SIGCOMM*, 2008.

[61] R. Sinha, C. Papadopoulos, and J. Heidemann. Internet packet size distributions: Some observations. Technical Report ISI-TR-2007-643, USC ISI, 2007.

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

[63] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full SHA-1. Cryptology ePrint Archive, Report 2017/190, 2017.

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

[65] R. Sylvester. IP address typo leads to a false arrest in Kansas. The Wichita Eagle, http://www.kansas.com/mld/eagle/news/local/crime_courts/12620843.htm.

[66] R. Teixeira and J. Rexford. A measurement framework for pin-pointing routing changes. In *Proc. NetT*, 2004.

[67] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proc. SIGCOMM*, 2014.

[68] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *Proc. NSDI*, 2013.

[69] X. Zhang, A. Jain, and A. Perrig. Packet-dropping adversary identification for data plane security. In *CoNEXT*, 2008.

[70] Y. Zhang, Z. M. Mao, and M. Zhang. Detecting traffic differentiation in backbone ISPs with NetPolice. In *Proc. IMC*, 2009.

[71] Z. Zhang, Y. Zhang, Y. C. Hu, Z. M. Mao, and R. Bush. iSPY: Detecting IP prefix hijacking on my own. *IEEE/ACM Trans. Netw.*, 18(6):1815–1828, Dec. 2010.

[72] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proc. SOSP*, 2011.

[73] W. Zhou, Q. Fei, S. Sun, T. Tao, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. NetTrails: A declarative platform for provenance maintenance and querying in distributed systems. In *Proc. SIGMOD Demo*, 2011.

[74] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *Proc. VLDB*, 2013.

[75] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *Proc. SIGMOD*, 2010.

[76] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, 2014.