usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Don’t Yank My Chain: Auditable NF Service Chaining

Guyue Liu and Hugo Sadok, *Carnegie Mellon University;*
Anne Kohlbrenner, *Princeton University;* Bryan Parno, Vyas Sekar,
and Justine Sherry, *Carnegie Mellon University*

https://www.usenix.org/conference/nsdi21/presentation/liu-guyue

This paper is included in the
Proceedings of the 18th USENIX Symposium on
Networked Systems Design and Implementation.
April 12–14, 2021

978-1-939133-21-2

Don't Yank My Chain: Auditable NF Service Chaining

Guyue Liu, Hugo Sadok, Anne Kohlbrenner, Bryan Parno, Vyas Sekar, Justine Sherry*

Carnegie Mellon University **Princeton University*

Abstract

Auditing is a crucial component of network security practices in organizations with sensitive information, such as banks and hospitals. Unfortunately, network function virtualization (NFV) is viewed as incompatible with auditing practices which verify that security functions operate correctly. In this paper, we bring the benefits of NFV to security-sensitive environments with the design and implementation of AuditBox.

AuditBox not only makes NFV compatible with auditing, but also provides stronger guarantees than traditional auditing procedures. In traditional auditing, administrators test the system for correctness on a schedule, e.g., once per month. In contrast, AuditBox continuously self-monitors for correct behavior, proving runtime guarantees that the system remains in compliance with policy goals. Furthermore, AuditBox remains compatible with traditional auditing practices by providing sampled logs which still allow auditors to inspect system behavior manually. AuditBox achieves its goals by combining trusted execution environments with a lightweight verified routing protocol (VRP). Despite the complexity of routing policies for service-function chains relative to traditional routing, AuditBox's protocol introduces 72-80% fewer bytes of overhead per packet (in a 5-hop service chain) and provides 61-67% higher goodput than prior work on VRPs designed for the Internet.

1 Introduction

Modern networks contain a myriad of *network functions* (NFs) such as firewalls, intrusion detection systems, normalizers, exfiltration detectors, and proxies. Beyond security and performance benefits, a key driving factor for NFs is that they are mandated by legal and policy requirements; *e.g.,* HIPAA [8], FERPA [7], and PCI [13], among others.

While *network functions virtualization* (NFV) promises potential benefits in cost, elasticity, and richer policies [32, 58, 63], there is significant resistance to adoption of NFV [6] for such regulatory use cases. Conversations with industry experts suggest that this reluctance stems from the inability to audit NFV deployments to demonstrate *compliance* as mandated by standards [15, 41]; *i.e.,* show that the service function chains (SFC) are correctly implemented and packets traverse the intended sequences of NFs in the right order.

With legacy hardware NF deployments, administrators and auditors can simply look at static wiring and hardware placement to intuitively verify ('what you see is what you get') if the network meets intended requirements.[1] In contrast, NFV introduces new dimensions of dynamism, virtualization, and multiplexing in the environment. For instance, VMs running NFs may be ephemeral, virtual switches may multiplex several services, and servers may host multiple services. Enhanced dynamism and a larger attack surface make NFV systems harder to reason about and as such, regulators do not have suitable tools for auditing. This lack of auditing is a fundamental stumbling block for NFV adoption.

To this end, we propose (1) *formal models* of correct SFC routing which clarify 'correctness' in the context of dynamic NF scheduling and routing; and (2) a protocol which *provably* provides *continuous* assurance that packets follow the (formally specified) policy-mandated paths.

Realizing this vision in a practical system, however, is challenging. To see why, consider traditional *verified routing protocols* [46, 55] (VRPs). At a high level, VPRs cryptographically ensure that packets are not modified in flight and do not deviate from a traversal of a prespecified sequence of routers. Unfortunately, VPRs fail to provide the required capabilities for our setting. First, VRPs assume that packets will traverse their path unmodified, but NFs can legitimately modify packets. Second, VRPs assume that the correct route for packets is fixed and known a priori, but in SFC the correct route for a packet may only be revealed mid-flight. Third, these approaches focus on per-packet behavior, whereas NFV often involves stateful NFs whose semantics depend on cross-packet state. Furthermore, VRPs have prohibitively high performance overhead; *e.g.,* OPT [46] increases min-sized packets by 3× for a 4-NF chain, and even the most recent work EPIC [47] incurs 1.69× overhead for a strong attack model.

In this paper, we present the design and implementation of AuditBox, which (1) re-enables status-quo 'what you see is what you get' auditing practices, and (2) raises the bar by enforcing *at runtime* that the system operates correctly. AuditBox builds on four key ideas:

- *Using secure enclaves:* To ensure that the correct NF software is running, AuditBox runs them atop hardware enclaves (*e.g.,* Intel's SGX [25]). By changing the trust model, we reformulate verified routing to audit actions between *trusted* NFs, with an untrusted network in between.
- *Trusted PacketIDs:* To tackle dynamic packet modifications, *immutable packetIDs* are carried by an AuditBox packet trailer. This enables us to logically bind modified packets to incoming packets when creating audit trails.
- *NF-hop-by-hop protocols:* Given trusted NFs, we devise a simplified path attestation protocol that focuses on the packets at individual "NF hops". This hop-by-hop attestation has the dual benefit of addressing dynamic paths and reducing the size of attestation headers.

[1] As we argue later, this is indeed a weak guarantee, but today's NFV deployments lack tools even for this weak property.

- *Lightweight simplified operations:* The use of trusted NFs inside enclaves enables a number of simple-yet-effective cryptographic optimizations such as the use of symmetric keys and *updatable MAC* computations to significantly improve data plane performance.

To realize these ideas with minimal modifications to NF implementations, AuditBox embeds a trusted SFC routing shim in the enclave alongside the NF. The shim receives inbound/outbound packets to/from the NF and is simultaneously responsible for generating *audit trails* for traditional auditing practices and for our new goal of enforcing *runtime checks* that the untrusted components of the system behave as expected. Relative to verified routing protocols for the Internet, AuditBox introduces 72-80% less per-packet header overhead (assuming a 5-hop service chain) and offers 62-67% higher goodput in the dataplane.

Nonetheless, auditability in AuditBox (or any such framework) does come at a cost relative to an uninstrumented NFV cluster (*e.g.,* 3%-38% overhead for s single NF as shown in Figure 16). That said, for security-sensitive settings, regulatory compliance is a fundamental requirement for NFV deployment. Hence, AuditBox's essential advantage is in bringing the benefits of ease of management, lower-cost equipment, faster upgrades and security patches, and flexibility that are associated with NFV to new markets where it would not have been viable previously. Indeed, we estimate that AuditBox, despite its overheads relative to uninstrumented NFV, can still result in capital savings of 1.9-60× (depending on the appliance) relative to traditional hardware middlebox solutions.

2 Background and Motivation

We begin by discussing network compliance today (§2.1), our problem statement (§2.2) and threat model (§2.3), and finally discuss why compliance for NFV is challenging (§2.4).

2.1 Tussle Between Compliance and NFV

Modern organizations need to satisfy a number of security standards for compliance with government and industrial regulations (*e.g.,* HIPAA, FERPA, FISMA, GDPR, and PCI, among others) [66]. In this paper, we focus on requirements or *controls* related to network security under NIST 800-53 [41] in the United States:[2]

- *Middleboxes must be deployed to protect sensitive data and systems:* Control SC-7 mandates the need for protection devices (*e.g.,* proxies, gateways, firewalls, guards, encrypted tunnels) arranged in an effective architecture.

- *Administrators must periodically test that security infrastructure is running properly:* Control SI-6 demands that these inspections be performed periodically, with 'once a month' as an example acceptable frequency.

- *Systems must provide logs of anomalies and past behavior:* Control AU-2 requires systems to keep such records for later analysis for auditing.

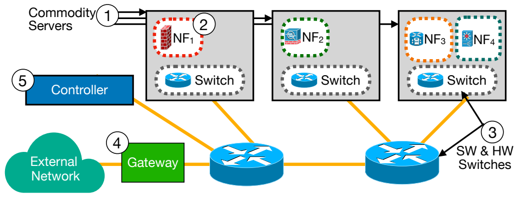


Figure 1: Components of a basic NFV cluster.

- *Independent 'auditors' must certify that security mechanisms are in place and running correctly:* Control CA-7 mandates that organizations be 'certified' by outside auditors (*e.g.,* third-party IT consulting companies) that the above requirements (among others) are being met.

Reluctance to adopt NFV: To meet the above compliance requirements, there is a large regulatory technology ('RegTech') industry [66] (expected to surpass $55.28B by 2025). One might expect then that this RegTech market would be an early adopter of NFV to reduce capital and operating expenses. However, our conversations with representatives from NIST and a RegTech firm revealed that this industry is *hesitant* to adopt NFV. The key reason is that while NFV lowers the bar for some aspects of compliance (*e.g.,* SC-7), it makes other requirements such as SI-6, AU-2, and CA-7 difficult, if not impossible.

In hindsight, this reluctance is not surprising. NFV introduces new dimensions of dynamism and multiplexing (*e.g.,* shared hosts running VMs, dynamic overlay routing, dynamic load balancing). This makes it harder to reason about the deployment and introduces an increased attack surface for threats (and misconfigurations). In contrast, a simple statically-wired NF deployment with hardware boxes seems intuitively easy to test, audit, and demonstrate compliance to external auditors.

2.2 Problem Setup

We consider an NFV cluster managed by a framework such as E2 [58], AT&T Domain 2.0 [2], or Blue Planet [3]. At a high level, NFV clusters consist of five basic components (Figure 1): (1) *Commodity servers* on which containerized or virtualized NFs run; (2) *Network Functions* such as firewall, proxy, or IDS; (3) *Software and Hardware Switches* that steer traffic between a sequence of NFs; (4) *Gateways* where cluster traffic enters; and (5) a *Controller* responsible for provisioning NFs on the servers and defining routes through them. NF instances are composed to create *service function chains* (SFC) policies for specific traffic classes (*e.g.,* Gateway → Firewall → Proxy → IDS → Gateway).

To make these systems *auditable*, we need to verify that: (1) the correct and untampered NFs are running correctly and (2) packets traverse these NFs according to policy. We formally define what it means for traffic to traverse NFs 'according to policy' in §4. Furthermore, to meet the compliance requirements, operators must be able to inspect and demonstrate that the correct behavior is happening; *e.g.,* trigger tests to observe that packets follow policies (SI-6) and

[2] ISO 270001 [15] has similar international standards.

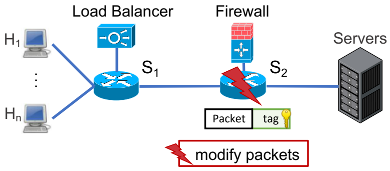


Figure 2: MACs for integrity are invalidated by rewritten packets.

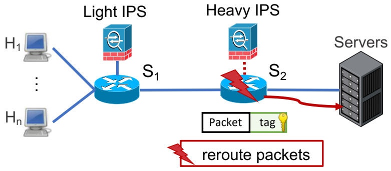


Figure 3: Rerouting traffic and bypassing the heavy IPS.

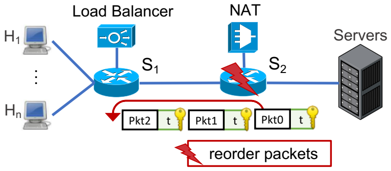


Figure 4: Reordering packets effects NF state and future packets.

produce *audit trails* for external inspectors.

To 'raise the bar' for auditability, we add two additional requirements. First, rather than merely enabling administrators to test on demand or post-facto whether or not the system is or was running correctly, we also want to enable the system to *audit itself, continuously, at runtime* for deviations from correct behavior, and to alert administrators if such a deviation is detected. Second, we also want to support rich *dynamic* SFC policies as opposed to traditional static service-chain policies; *e.g.,* steering packets tagged as suspicious by earlier NFs for deeper inspection [32].

In this context, we note that while auditability may seem related to *network verification* (*e.g.,* [44, 45, 60, 74], the requirements are fundamentally different on two fronts. First, most existing network verification efforts simply look at the configurations of the network elements such as NFs/switches and formally verify if the configuration meets intended policies. It does not typically provide any runtime guarantees about data plane actions. Second, network verification *can* be used to provide evidence of correct operation [45] but it does not provide 'what you see is what you get' audit trails; in this regard, verification could be coupled with audit trails to provide additional evidence of compliance.

2.3 Threat Model

Although most operators are primarily concerned with cluster misconfiguration rather than outright attacks, we target a stronger threat model as follows. First, what is trusted: the controller is the arbiter of correct policy and how NFs should be scheduled; we assume that the controller is trusted, and we do not consider attacks where a 'lead' administrator (that is, an administrator charged with configuring policies at the controller) provides invalid or malicious policies to the controller. In the case of the gateway and the NFs, we assume there exists vendor-certified code which is digitally signed. This code is privileged to drop or rewrite packets, and we exclude NFs that can inject packets.

Most other components of the network are untrusted. An attacker may attempt to corrupt server software (including the operating system and/or VMM), NF and gateway software, and the software and hardware of the switches. The attacker can cause one or more corrupted components to inject, drop, or rewrite packets.

Our solution builds on the 'abstract enclave assumption' defined by prior research [17, 18, 61]: the attacker cannot observe or modify any data or program code running within an enclave, and the enclave is trusted to attest to the integrity of the code running therein. While existing enclave solutions – such as SGX, which we build on – fall short of meeting the abstract enclave assumption perfectly [22, 24, 36, 54, 73], fixing the shortcomings of current enclave solutions is out of scope for this work, as such fixes are an active area of research in their own right [26, 34].

2.4 Challenges

At first glance, it would appear that we can borrow from prior work on *verified routing protocols* (VRPs) that cryptographically guarantee that a packet takes a pre-specified intended path and is not modified in flight (*e.g.,* [46,49,55,77,80–83]). We use OPT [46] as an exemplar state-of-art solution from this class. Specifically, OPT extends each packet with: (a) a cryptographic hash of the packet contents and (b) its expected switch-level path. Every router along the path verifies that the packet's current hash matches the header and also adds attestations to ensure the packet has matched the expected sequence. As we will see next, our NFV auditability problem introduces new dimensions outside the scope of these prior efforts.

Mutable Packets: Figure 2 shows two NFs: a load balancer that modifies the destination IP to distribute the load across multiple backend servers and a firewall configured to block packets from malicious IPs. Consider the scenario where switch S_2 (either adversarially or via misconfigurations) modifies the IP header to bypass the firewall. Because NFs can legitimately modify packet headers, it is difficult to distinguish whether this action was malicious or an intended NF action. OPT-like VRPs assume that most packet fields are immutable and perform crypto operations by excluding a few mutable packet fields (*e.g.,* TTL), and hence would generate a large number of false positives by flagging all legitimate NF modifications.

Dynamic Paths: Consider a dynamic SFC scenario in Figure 3, where a lightweight IPS performs basic detections and then routes suspicious packets to the heavy IPS for further processing. Again, an adversarial or misconfigured device could reroute all suspicious packets to bypass the heavy IPS, and it is hard to tell whether this was the result of the light IPS's action or a malicious switch. Because the intended path cannot be determined until the light IPS finishes processing, OPT-like VRPs – which must pre-specify the end-to-end route of the packet – are not applicable to this network.

Stateful Behavior: NFs' stateful semantics mean that per-packet auditability may not be sufficient. We may also need to ensure that all packets in a given flow follow the same path and that they arrive in order if we wish to ensure the stateful semantics are not compromised. To see why, consider Figure 4, which shows two stateful NFs: a layer 4 load balancer to distribute packets based on source IP and port and a NAT that maps public ports to private ports. Consider an adversary (or misconfiguration) that reorders packets and sends the `FIN` packet before the data packets. This could cause all following data packets to be discarded by the load balancer. This highlights a fundamental limitation of VRPs: because they reason about correctness of each packet independent of the others, there may be cross-packet policy violations which they cannot capture.

3 AuditBox Overview

Our goal in designing AuditBox is to provide auditing capabilities for NFV deployments. In practice, we also want: R1) *minimal modifications* to existing NFs and R2) *low overhead* on the data/control paths. In this section, we discuss some of the key ideas in AuditBox and its overall architecture.

3.1 Key Ideas

We first discuss the main ideas that enable AuditBox to tackle the challenges of packet modifications (C1), dynamic paths (C2), and stateful actions (C3) while meeting the practical requirements of minimal NF modifications (R1) and low overhead (R2).

A) Running NFs in enclaves atop a shim: Inspired by the trust guarantees provided by prior work [39,61,70], AuditBox runs NFs in trusted enclaves. This enables AuditBox to trust those modifications that are validly introduced by NFs (C1). To avoid modifying existing NFs (R1), we introduce a trusted shim in each enclave to perform auditing (§5.4). This shim also ensures that the next-hop NFs are chosen based on the intended policy (C2).

B) Trusted Packet ID: Mutable packets (C1) make it hard to generate audit trails as we cannot causally relate NF-modified packets to their inputs (§4). To tackle this, we introduce an *immutable packet ID*, carried by the packet header (§5). We envision a *trusted gateway* (running in an enclave) that generates and assigns this ID when the packet first arrives in the NFV cluster; the packet ID is carried through NFs even if the packet itself is modified or rewritten by NFs.

C) NF-Hop-by-hop updated attestations: We leverage the trusted shim in each enclave to develop a hop-by-hop attestation protocol in which each *pair* of shims attest that the packet was delivered, without improper modification between an NF and its policy-compliant successor. Compared to the end-to-end approach taken by traditional VRPs, hop-by-hop attestation has the dual benefit of supporting *dynamic paths* and also *reducing packet overheads*.

D) Efficient crypto mechanisms: By using trusted enclaves, we can use *one symmetric key* for all NFs in the same policy pipelet (§3.3), to simplify the cryptographic operations and also introduce new opportunities for efficiency. For example, we implement an efficient *updatable MAC* algorithm to improve the performance of repeated attestations to the packet at each hop (§6.1).

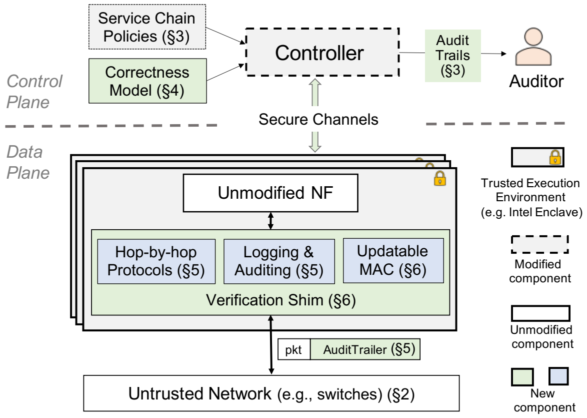


Figure 5: AuditBox Architecture.

3.2 AuditBox Architecture: Data Plane

The key components of the AuditBox architecture are illustrated in Figure 5.

In the data plane, AuditBox runs unmodified NFs in trusted execution environments (TEEs) to isolate them from other untrusted network components (*e.g.,* switches, OSes). Although our current implementation (§6) uses Intel SGX enclaves, our design in principle can be realized using other TEE technologies such as Arm TrustZone [1]. The key capabilities we leverage are attested memory isolation and integrity during program execution. In each enclave, we add a shim which intercepts the traffic entering/exiting the NF to serve three purposes. (1) The shim determines the correct next-hop NF according to a policy it received from the controller; this is no different than any other NFV policy manager such as FlowTags [32] or E2 [58].

The second two tasks for the shim are novel to AuditBox and form the entirety of sections 4, 5, and 6 and so we only introduce them briefly here. (2) The shim checks each incoming packet to verify that the packet has not been improperly routed or modified while traversing the untrusted network between enclaves; on egress the shim attaches a custom trailer (called AuditTrailer) along with a MAC attesting to the contents of the packet and trailer so that the next-hop NF can similarly verify the packet was routed correctly. (3) The shim logs any packets to local storage which *either* appear to violate policy, *or* are tagged via a secret 'log bit' for recording.

3.3 AuditBox Architecture: Control Plane

The controller serves two key purposes: NF deployment and management, and serving as an interface for an administrator to inspect logs and audit trails.

NF Deployment: NF deployment includes scheduling NFs

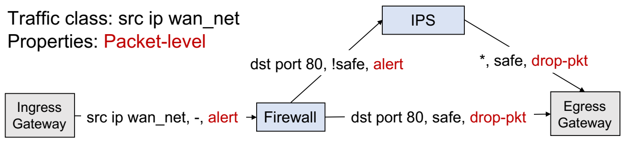


Figure 6: An annotated policy graph.

on servers, verifying enclave attestations that the correct NF software is running at each NF, distributing and updating the global symmetric key used for dataplane verification, and installing the correct *next-hop policies* at each enclave.

Policies are roughly similar to other DAG-based policy languages [31, 32, 58]: network operators specify *policy pipelets* where nodes represent NFs and edges are annotated with traffic classes and where they should be routed according to policy (illustrated in Figure 6). AuditBox augments this traditional policy with two new parameters: a *correctness model* which defines what behavior in the network constitutes a violation (presented in §4) and *violation annotations* on each edge to determine whether packets which violate the correctness model should be dropped or raise an alert. From this policy, the operator can define a function which, for a given packet egressing an NF, can determine what the correct next-hop NF to deliver the packet to is; this function is installed in the shim at each enclave along with the assigned violation action to take in case, *e.g.,* a packet is corrupted inflight between enclaves.

Logging and Audit Trails: Auditors and administrators can use the controller to query for *logs* (records of any anomalies or policy violations) and *audit trails* (the end to end path a packet takes through the cluster, and all intermediate rewritten states of the packet between NFs). By default, each enclave stores logs and audit trail records to local storage, encrypted with a symmetric key that is local to that NF; the controller sets up pairwise keys with each NF for log and audit storage. Because it is infeasible to log every packet through the system, the administrator configures a sampling policy; we discuss the sampling policy, the logging mechanism, and the security of the logs in §5.4.

4 Formalizing Correctness

We begin by formally defining what it means for a system to **(a)** obey correct routing, and **(b)** support auditing.

Correct Routing: Since networked forwarding elements are untrusted, we rely on a trusted shim in each SGX enclave to verify at runtime that the network has not deviated from correct behavior. §4.2 defines 'correctness' with regard to network forwarding behaviors, *i.e.,* those verified by the shim.

Auditability: Our verified routing approach operates on a hop-by-hop basis; as we discuss in §5 this simplifies our protocol and limits its size overhead. However, auditors expect *end-to-end* evidence (upon inspection) that the system is operating correctly. An 'audit trail' consists of a recorded sequence of *causally connected* operations across devices – *e.g.,* packet p entered at the gateway, was processed by NF_i resulting in p', which was processed by NF_j, and resulted in p'', which was released through the gateway. Due to space constraints, we defer our formal definition of an 'audit trail' to §B.2.

Prior to introducing our formal definitions (§4.2), we define our model of the network (§4.1). We make a few assumptions for the sake of simplifying our presentation. All can be relaxed at the cost of additional notational complexity. First, we assume that the cluster has a single ingress/egress 'gateway' where packets transit to/from the primary network. We similarly assume that each NF has only one ingress/egress port. Finally, we assume that forwarding within the cluster is performed at L2, and hence it is not necessary for network switches to update any TTL values or checksums, *i.e.,* there is no need for the network to modify packets.

4.1 Definitions

Time: We model time as an ordered sequence E, the set of all *events* in the system. E is initialized to $[]$, that is, empty.

As the (modeled) system runs, NFs in the system populate E with 4-tuples containing the following named values:

- $\mathbf{pkt}_{\mathrm{in}}$: a packet received ($\in P$, defined below).
- $\mathbf{pkt}_{\mathrm{out}}$: a packet sent ($\in P$).
- **NF**: the network function ($\in F$, defined below).
- **t**: the logical 'time' the packet was received or sent (*i.e.,* the index in E); represented as a positive integer ($\in \mathbb{N}$).

We may refer to members of $e \in E$ as $e.\mathrm{pkt}_{\mathrm{in}}$, $e.\mathrm{pkt}_{\mathrm{out}}$, $e.\mathrm{NF}$, or $e.\mathrm{t}$. We *index* E using array notation *e.g.,* $\mathrm{E}[43] = (*,*,*,43)$. We may also search E for events matching the specified value at a specified field using the function $\text{get-event}_E(\text{field, value}) \to [\mathrm{E}]$, for example, $\text{get-event}_E(\mathrm{pkt}_{\mathrm{in}}, p_i)$ returns the sequence of all events in E where p_i was received as input at an NF.

Note that, although AuditBox provides some logging mechanisms, *there is no globally ordered event log in the system implementation* – the event sequence defined here is merely an abstraction to help reason about time while modeling.

Packets: P is the set of all 64-9000 byte binary strings, that is, P is the set of all (up to jumbo framed) Ethernet packets. Each packet p contains an Ethernet header and an IP header. Depending upon the NFV framework, the packet may also contain a collection of 'metadata' fields such as FlowTags [32] or a Network Service Header (NSH) [64]. If the packet represents a TCP or UDP packet, we represent the classic flow 5-tuple (source IP address, destination IP address, source port, destination port, protocol) through the function $\textsc{flow}(p)$.

As a packet p traverses the network, it may be transformed into some p' or p'' through modifications to the payload or metadata fields. If the data *anywhere* in the packet – that is, the 64-9000 byte binary string it represents – has been changed, then $p \neq p'$. In the event log, $e.\mathrm{pkt}_{\mathrm{in}}, e.\mathrm{pkt}_{\mathrm{out}} \in P$.

NFs: There is a set of Network Functions F. Each $\mathrm{NF}_i \in F$ is a function,[3] $\mathrm{NF}_i : P \to P \cup \{\bot\}$. That is, it takes in a packet and produces another packet (or null).

[3] Called a 'transfer function' elsewhere in the literature [44, 60].

In our model, we assume NF_i encloses some NF_i-specific function $f_i : P \to P \cup \{\perp\}$, which may keep state, modify packet contents, etc. When NF_i takes in a packet and f_i returns null, this represents a drop. We exclude NFs that could inject new packets.

With respect to the event log E, we log each operation at NF_i as follows:

Algorithm 1 NF Model

```
function $\text{NF}_i$(input)
    output $\leftarrow f_i$(input)
    if output $\neq \perp \vee$ input $\neq \perp$ then
        $E$.append(input, output, $\text{NF}_i$, $E$.length + 1)
    return output
```

Switches: Like prior work in SDN and formal modeling [44], we consider the network as 'one big switch' or a 'fabric' and we model its behavior with a single function $\Phi : P \cup \{\perp\} \to P \cup \{\perp\}$. In our threat model, Φ is the untrusted part of the system. Φ is not considered an NF ($\Phi \notin F$), and Φ does not append to E.

Gateway: Packets enter and exit the cluster – and hence enter and exit the model – via a dedicated NF, GW which, like other NFs, also appends to the event sequence. We model the Gateway as GW_{in} when a packet enters the cluster via the gateway as follows:

Algorithm 2 Model of Packets Entering the Cluster

```
function $GW_{\text{IN}}$(input)
    if input $\neq \perp$ then
        $E$.append($\perp$, input, $\text{GW}_{\text{in}}$, $E$.length + 1)
    return $f_{in}(input)$
```

Like normal NFs, the gateway applies a gateway-specific function f_{in} to the packet before transmitting it. We define GW_{out}, for when a packet exits the cluster similarly (§B.1).

Path: Packet processing occurs via traversal of a sequence of NFs and switches. Whenever an NF sends a packet, it is passed to the network which is expected to steer the packet to the next NF specified by the service-chain policy. When a new packet arrives at GW_{in}, we model traversal of the network via a nested function of Φ and elements of F, for example: $\text{GW}_{\text{out}} \circ \Phi \circ \text{NF}_i \circ \Phi \circ \text{NF}_j \circ \Phi \circ \ldots \circ \Phi \circ \text{GW}_{\text{in}}$.

Policy: There are many languages [32, 58] and services for specifying service-chain policy. Here we assume that for a packet and an NF which just produced that packet, that there exists some *policy function* ($\text{policy} : P \times F \to F$) that can determine the NF that should next process the packet. Importantly, we assume that the information necessary to determine the right next hop relies only in the packet fields (*e.g.,* IP header, metadata) and the departing NF.

4.2 Correctness

We give two possible definitions of "correct" forwarding below, based on properties of the modeled event log E.

4.2.1 Packet Correctness

Our first definition, packet correctness, is based on the UDP service model of packet delivery. Under the packet-correctness definition, the system is 'correct' even if packets are reordered, dropped, or duplicated between NFs. The behaviors which are incorrect under this model are limited to **(a)** injecting packets which were not sent by an end host, or **(b)** modifying/corrupting packets between sender and receiver.

Packet correctness holds iff Property 1 holds over E.

$$\forall e_b \in E \text{ s.t.}: e_b.\text{pkt}_{\text{in}} \neq \perp, \tag{1}$$

$$\exists e_a \in E \text{ s.t.}: e_a.\text{t} < e_b.\text{t} \wedge e_a.\text{pkt}_{\text{out}} = e_b.\text{pkt}_{\text{in}} \tag{2}$$

$$\wedge \text{policy}(e_a.\text{pkt}_{\text{out}}, e_a.\text{NF}) = e_b.\text{NF} \tag{3}$$

Property 1: No Injection or Modification

To summarize the above: for all events e_b in which an NF e_b.NF receives and processes a packet, there exists a prior event e_a in which e_a.NF sends the same packet to e_b.NF.

4.2.2 Flow Correctness

Flow correctness is based on the TCP service model. Like packet correctness, a network where packets are injected or corrupted by the network is not correct. To meet flow correctness, the network also must not drop packets, reorder packets within a flow, or duplicate them. Hence, the network obeys Flow Correctness iff it respects Property 1 and the following three properties.

The first additional property aims to verify that the network has not dropped any packets in flight. An absolute guarantee that no packets are dropped is impossible, as the NFs cannot force hostile network elements to deliver packets. Hence, we instead define our 'no drops' property to state that packets in a given flow that *do arrive* at their own destination may only be accepted if all packets previously transmitted from that flow to that destination have already arrived. To achieve this, Property 2 ('No Drops') says in English that if a packet $e_{a2}.\text{pkt}_{\text{in}}$ is sent by e_{a2}.NF and received at e_{b2}.NF, then any other earlier packet $e_{a1}.\text{pkt}_{\text{out}}$ belonging to the same flow and also destined for e_{b2}.NF should have already been received by e_{b2}.NF prior to $e_{b2}.\text{pkt}_{\text{in}}$.

$$\forall \text{NF}_a, \text{NF}_b \in F, \forall e_{a1}, e_{a2}, e_{b2} \in E \text{ s.t.}: \tag{4}$$

$$(e_{a1}.\text{t} < e_{a2}.\text{t} \wedge e_{a1}.\text{pkt}_{\text{out}} \neq \perp \wedge e_{a2}.\text{pkt}_{\text{out}} \neq \perp \tag{5}$$

$$\wedge e_{a2}.\text{pkt}_{\text{out}} = e_{b2}.\text{pkt}_{\text{in}} \tag{6}$$

$$\wedge \text{policy}(e_{a1}.\text{pkt}, e_{a1}.\text{NF}) = \text{policy}(e_{a2}.\text{pkt}, e_{a2}.\text{NF}) \tag{7}$$

$$\wedge e_{a1}.\text{NF} = e_{a2}.\text{NF} = \text{NF}_a \wedge e_{b2}.\text{NF} = \text{NF}_b \tag{8}$$

$$\wedge \text{flow}(e_{a1}) = \text{flow}(e_{a2})) \implies \tag{9}$$

$$(\exists e_{b1} \in E \text{ s.t.}: \tag{10}$$

$$(e_{b1}.\text{t} < e_{b2}.\text{t} \wedge e_{b1}.\text{NF} = \text{NF}_b \wedge e_{b1}.\text{pkt}_{\text{in}} = e_{a1}.\text{pkt}_{\text{out}})) \tag{11}$$

Property 2: No Drops

Property 3 ensures that packets within a flow are not reordered between NFs. In English, the property specifies that if an NF_a transmits a packet $e_{a1}.\text{pkt}_{\text{out}}$ before a packet

$e_{a2}.\text{pkt}_{\text{out}}$ to the same NF_b, then $e_{a1}.\text{pkt}_{\text{out}}$ should also *be received* by NF_b before $e_{a2}.\text{pkt}_{\text{out}}$.

$$\forall \text{NF}_a, \text{NF}_b \in F, \tag{12}$$
$$\forall e_{a1}, e_{b1}, e_{a2}, e_{b2} \in E \text{ s.t. :} \tag{13}$$
$$(e_{a1}.\text{pkt}_{\text{out}} = e_{b1}.\text{pkt}_{\text{in}} \land e_{a2}.\text{pkt}_{\text{out}} = e_{b2}.\text{pkt}_{\text{in}} \tag{14}$$
$$\land e_{a1}.\text{pkt}_{\text{out}} \neq \perp \land e_{a2}.\text{pkt}_{\text{out}} \neq \perp \tag{15}$$
$$\land e_{a1}.\text{NF} = e_{a2}.\text{NF} = \text{NF}_a \land e_{b1}.\text{NF} = e_{b2}.\text{NF} = \text{NF}_b \tag{16}$$
$$\land \text{flow}(e_{a1}) = \text{flow}(e_{a2})) \implies (e_{a1}.\text{t} < e_{a2}.\text{t} \iff e_{b1}.\text{t} < e_{b2}.\text{t}) \tag{17}$$

Property 3: No Reordering Within the Same Flow

$$\forall e_a \in E, \nexists e_b \in E \setminus \{e_a\} \text{ s.t. : } e_a.\text{pkt}_{\text{in}} = e_b.\text{pkt}_{\text{in}} \tag{18}$$

Property 4: No Duplication

Finally, we ensure that packets are not duplicated in the networks ('replay attacks') with Property 4. We note that the Property 4 formalization assumes that the same packet is never sent by any NF more than once. This seems to be a reasonable assumption. As we are forwarding under L2, an NF_A and an NF_B will always have different Ethernet headers even if the IP, TCP/UDP, and payload are identical. And it seems reasonable to assume that an NF_A will never transmit the same packet twice either – even a re-transmitted TCP packet will come with a different IPID value. However, we discuss how to revise this formalism to allow NFs to duplicate packets in §A.

5 AuditBox Protocol

In this section, we focus on the dataplane protocol. We begin by providing a high-level view of the actions that each AuditBox-enabled node performs at each hop (§5.1). Then we discuss the detailed construction of the attestation check and update logic we use for packet- (§5.2) and flow-level (§5.3) correctness. We envision different NFV deployments can flexibly choose one of these levels of correctness as desired. Finally, in §5.4 we describe the secure logging mechanisms which allow auditors to observe audit trails demonstrating that the system is operating correctly.

5.1 High-level workflow

AuditTrailer: AuditBox adds an AuditTrailer, carried by the packet, to verify integrity. We discuss the contents of AuditTrailer in packet-level and flow-level form in detail in §5.2 and §5.3 respectively; here we briefly present the two foundational fields for auditing and runtime correctness. These fields are present in both the packet-level and flow-level versions of the AuditTrailer.

pktID is a unique, immutable ID assigned to a packet; when a packet enters an NF and is modified or rewritten, the *pktID* allows us to identify for each input packet which output packet (if any) is a result of its processing. This tracing is the key mechanism which enables us to generate audit trails – the causal sequence of events we present in §B.2.

tag is a message authentication code (MAC) which allows us to identify if a packet has been improperly modified while in flight between two (trusted) shims. Operationally, the tag is a Galois Message Authentication Code (GMAC), computed using a symmetric key over various packet fields (discussed below in §5.2 and §5.3). The *tag* field enables us to perform runtime correctness checks while running with untrusted operating systems, switches, *etc.*.

Data Plane Actions (Shim and Gateway): The AuditBox dataplane protocol is implemented at the gateway to/from the cluster and at the shims inserted into each NF enclave.

At the Gateway: For each incoming packet, the ingress gateway generates an AuditTrailer (including a unique pktID field) and appends it to the packet, and then forwards it to the next NF. At the end of the service chain, as packets leave the cluster, the egress gateway validates the AuditTrailer (assuming the validation passes), and then removes the AuditTrailer and forwards the packet out of the cluster.

At the Shim: At each hop, the shim receives the incoming packet and performs the following four operations: Check, Process, Update, and Log. *Check* validates the received packet, including verifying the MAC stored in the *tag* field, to verify that the received packet was not incorrectly modified by the network. *Process* hands the packet to the actual NF code; when the packet is returned from the NF, the shim then *Updates* the AuditTrailer (*e.g.,* computing the new *attestation* field). Finally, in certain cases the shim *Logs* the packet and certain metadata to produce an audit trail; the packet is then released to its next hop in the service chain.

5.2 Packet Correctness Protocol

As presented in §4.2.1, the goal of our packet correctness protocol is to ensure that packets are not injected or modified by the network, *i.e.,* that all packets received and processed by an NF were transmitted *to* that NF by another NF or the ingress gateway.

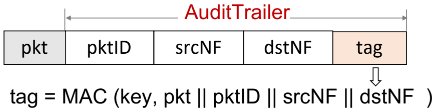


Figure 7: The AuditTrailer for packet-level integrity.

For packet correctness, the AuditTrailer (Figure 7) contains the following fields: pktID (6 bytes), tag (16 bytes), srcNF (2 bytes), dstNF (2 bytes). As discussed above, the pktID is required for generating audit trails so we do not discuss it further here other than to require that it be transmitted alongside other packet fields (IP header, payloads, *etc.*) uncorrupted by the network. Naïvely, one might simply compute the MAC over the packet and pktID to compute the tag. However, this would only meet a portion of the clauses from Property 1 – that there exists *some* valid NF which transmitted this packet. If a malicious or misconfigured switch delivers a valid packet, intended for some NF A, to NF B, the packet would appear to have a valid tag.

Adding the srcNF and dstNF fields explicitly to the header bind the packet to the *policy-compliant route*, meeting the final clause of Property 1. The sender explicitly encodes its own NF ID and the intended destination NF ID into the packet and computes the MAC over the packet, pktID, srcNF, and dstNF fields. The receiver, by validating the MAC stored in the tag, can be sure that the sending NF intended to send the packet to the receiver; the receiver can also (redundantly, as a sanity-check) re-compute the policy compliant next-hop NF for the received packet and the source NF to further verify that it is indeed the correct recipient.

We specify this protocol in Algorithm 5 and prove that it meets the requirements of Property 1 in Appendix D.

Theorem 1 (Packet Correctness) *Consider the game described in §D.3 with adversary $\mathcal{A}$ and instantiated with the AuditBox packet-correctness protocol. Specifically, looking at Algorithm 5, we instantiate f_{in} with the function* GENERATE *and f_i with* PROCESS$_i$*. The probability that the game outputs an event log E that violates Property 1 is negligible.*

5.3 Flow Correctness Protocol

We now discuss the implementation of the Flow Correctness Protocol, which in addition to Property 1 also meets Properties 2, 3, 4. We envision that networks that use only stateless or order-insensitive NFs will prefer the lighter packet-correct protocol, but those with stateful NFs whose operations are sensitive to packet ordering may use the stronger flow-correctness protocol.

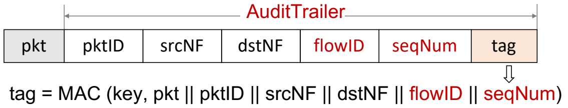

AuditTrailer
pkt
pktID
srcNF
dstNF
flowID
seqNum
tag
tag = MAC (key, pkt || pktID || srcNF || dstNF || flowID || seqNum)

Figure 8: The AuditTrailer for flow-level integrity.

As shown in Figure 8, to extend the AuditTrailer for flow-level semantics, we add two additional fields over the packet-level version: a flowID field (4 bytes), and a seqNum field (4 bytes). The tag is now computed over the packet and all fields including flowID and seqNum.

When packets enter the cluster through the gateway, the gateway hashes the classic '5-tuple' and looks this up in a flow table mapping 5-tuples to flowIDs. If a flowID already exists for this flow, the gateway appends the existing flowID to the AuditTrailer. If a flowID does not exist for this flow, the gateway assigns an unallocated ID number to the flow and inserts this into the flow table and the packet. For non-TCP and non-UDP packets, the flowID is simply set to 0. Like the pktID, the flowID is unmodified as packets flow through the network – even if header values and port numbers are rewritten, the flowID remains the same across NFs; as we will show, this is important for the creation of audit trails (§5.4).

seqNum values are maintained per-hop and represent the ordering of packets between a sending NF$_A$ and a receiving NF$_B$ for a given flow. The sending NF maintains an incrementing per-flow counter, and, for each packet from the same flow, appends the sequence number to the seqNum field in the AuditTrailer. At the receiver, there is a corresponding table of per-flow counters with the next sequence number expected. In our implementation, there is no reason for in-network reordering within a flow and hence the receiver raises an alert for any out-of-order packets; we could configure the receiver to maintain a small buffer to wait for reordered packets and put them back in order for processing in a cluster (and to only raise an alert after multiple out of order arrivals) where reordering were for some reason possible. Thus, packets are discarded at the receiver if they *either* fail the tag verification *or* fail the expected-next-sequence number test.

Because the protocol rejects out-of-order packets, it easily meets Property 3. Since duplicate packets will have the same sequence number, they are detected. Thus, the protocol also meets Property 4. Finally, if a packet is dropped, the arrival of subsequent packets will induce alerts due to out-of-order sequence numbers, meeting Property 2.[4] We specify this protocol in Algorithm 6 and formally prove its security in Appendix D.

Theorem 2 (Flow Correctness) *Consider the game described in §D.2 with adversary $\mathcal{A}$ and instantiated with the AuditBox flow-correctness protocol. Specifically, looking at Algorithm 6, we instantiate f_{in} with the function* GENERATE *and f_i with* PROCESS$_i$*. The probability that the game outputs an event log E that violates Properties 1-4 is negligible.*

Why not use existing TCP sequence numbers? Rather than embedding an additional sequence number, one might attempt to naïvely re-use the sequence number already embedded in TCP flows to save additional bytes in the header. However, packets may *already be missing* when they enter the cluster (leading to sequence number gaps which are not the result of misbehavior within the cluster) and NFs may choose to drop packets (also creating legitimate sequence number gaps). Hence we need a new sequence number whose role is only to detect gaps that are the result of drops *between NFs* in our cluster.

Why not one sequence number per pair of NFs, rather than per-flow? At first glance, it may seem simpler to keep a sequence number across all flows rather than a sequence number per-flow. However, many NF implementations use receive side scaling (RSS) at the receive NIC to fan out packets across multiple cores; in such an architecture a unified sequence number becomes a performance bottleneck; per-flow counters are more parallelizable and hence more scalable.

5.4 Logging & Auditing

Traditional logging (as mandated by AU-2 in §2) focuses on recording packets which resulted in alerts, policy violations,

[4] However, an attacker who blocks *all* packets from a flow may go undetected – to avoid this noncompliance, a receiver detecting a flow with over a minute without transmissions will query the sender for its sequence number to detect any dropping behavior.

and anomalies. At each shim, any packet which results in a violation is logged in its entirety, including all fields of the AuditTrailer; the log is written to local storage and encrypted with an NF-local symmetric logging key provided by the controller.

AuditBox also records additional logged events to produce an 'audit trail', which restores 'what you see is what you get' confidence in the correctness of the underlying system. As defined in §B.2, an audit trail consists of a hop-by-hop trace of a packet (or all packets in a flow) through a sequence of NFs as well as their intermediary rewritten states.

The basic idea is as follows. The gateway probabilistically samples incoming packets and tags them with a 'log bit' which follows the packet through the cluster along with the AuditTrailer. Any packet whose log bit is set to true is logged at each shim, including all metadata or AuditTrailer fields. The administrator or inspector specifies three parameters to define which packets are sampled: (1) a Berkeley Packet Filter (BPF) to match for selected packets, (2) a sampling rate (*e.g.,* 0.001%), and (3) whether to log at the packet- or flow- granularity. Logging at the flow granularity is only permitted if the policy is in flow-correctness mode.

Upon querying, the administrator can then inspect manually (or via an automated script) the path of packets or flows throughout the entire system. Even if packet headers or fields are changed, the packetID or flowID are preserved between NFs; even in traditional service chain deployments such tracing is not possible today when NFs modify packets in a way that they cannot be connected to their corresponding inputs.

There is only one 'trick' to this very simple design: in a truly malicious environment, the network could selectively treat logged packets according to correct policy and only manipulate unlogged packets. Although we would still expect our runtime checks at each shim to detect the error, auditors would no longer be able to retrieve an audit trail associated with the violation. Hence, it is important that we *hide* the logging bit at the gateway so that that the attacker cannot mount such an attack. In the following section, as we discuss our implementation, we describe how the logging bit is stored virtually in a way that is cryptographically secure while consuming 0 bits of overhead in the AuditTrailer.

6 AuditBox Implementation

We now discuss two performance optimizations to our protocol (§6.1) and our end-to-end prototype (§6.2).

6.1 Optimizing Verification

Updatable GMAC: To support mutable packets and dynamic paths, AuditBox computes the MAC twice for each packet at each hop: first to verify the packet when receiving it, and second to authenticate the packet before sending it out (§5). In our implementation, we use the GMAC algorithm [53]. While GMAC is one of the fastest authentication algorithms (thanks in part to acceleration from Intel's AES-NI instructions [40]), it still adds non-trivial overhead to packet processing.

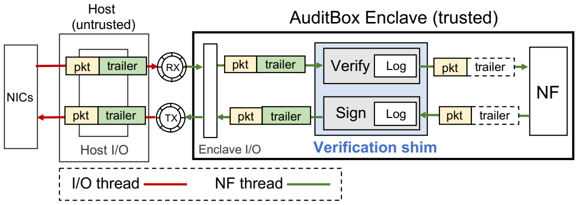


Figure 9: AuditBox software architecture (white boxes denote existing Safebricks components).

To reduce AuditBox's overhead, we implement a proven secure [52] *updatable* version of GMAC on top of EverCrypt [62], a formally verified cryptographic library. We use EverCrypt because it is easy to port into SGX and it is fast; its verified properties are a pleasant bonus. We defer to future work the extension of EverCrypt's formal verification to our updatable API.

The key optimization of updatable GMAC is to take advantage of GMAC's algebraic structure to securely *reuse* the first MAC when computing the second MAC [52]. With this optimization, the second MAC's cost is proportional to the number of modified data blocks, rather than the total packet length. In practice, this improves performance for NFs that only read the packet or that modify a small portion of it (§7).

In addition to a key and a message, GMAC requires as input an initialization vector (IV) that must be unique for each MAC invocation with a given key. For this we use the concatenation of the srcNF and pktID fields. This leads to a bound on how long we can use the same key K_{σ}. At 100Gbps, we would expect to overflow the pktID once every 22 days and hence we use we use a 14 day key rotation which ensures that the same IV is never used twice. When the pktID wraps around, one can use timestamps to differentiate entries in audit trails.

Secret Logging: In §5.4 we discussed that we must encrypt the bit used to mark which packets should be logged for auditing; we now describe how we introduce 0 bits of overhead and 0 computational overhead for unsampled packets. Our idea is to embed a *virtual logging bit* in the AuditTrailer. When the ingress gateway generates the *tag*, it appends this virtual logging bit as the last field when computing the MAC. For example, $tag = \mathrm{MAC}_{K_{\sigma}}(pkt||pktID||srcNF||dstNF||\ 1)$ for a packet that should be logged. When the packet arrives at the verification shim, it verifies the MAC by appending a 0 (assuming most packets will not be logged). If the verification fails, the NF then appends 1 and performs a second MAC verification. The success of the second verification means the NF should log the packet, while failure indicates a malicious/mangled packet. We formally prove this approach is secure in §D.

Theorem 3 (Secret Logging Security) *Consider the game described in §D.5 with adversary $\mathcal{A}$. When the MAC algorithm is GMAC, the adversary's advantage is negligible.*

6.2 Prototype Details

We implement an end-to-end prototype using Safebricks [61]. Safebricks is an NFV framework that builds on top of DPDK [5] and runs NFs in Intel Enclaves [25]. While the idea of AuditBox could be applied to other NFV frameworks [48, 58, 59, 79], we choose Safebricks mainly to leverage its I/O optimization to avoid expensive enclave transitions. We modified approximately 4k lines of Rust to implement our protocols, and added 2.5k lines of C and 100 lines of x86 assembly to implement and test the updatable GMAC implementation. Here, we focus on the implementation of the verification shim and the AuditTrailer, which are the key enablers to avoid any NF changes.

Verification Shim: As shown in Figure 9, AuditBox inserts a verification shim that sits between the enclave I/O interface and the NF in each enclave. The shim implements our custom verification protocol using two modules: one verifies the incoming packets by checking the AuditTrailer, and the other updates the AuditTrailer on outgoing packets. Both modules use our *updatable* GMAC algorithm, and both modules have access to the logging function to save logs for offline auditing.

Packet Trailer: Typically, to carry a wrapper or metadata header through unmodified NFs, one incurs two overheads. First, one must *strip* the header from the packet and copy the packet (starting from the IP or Ethernet header) to the first byte of the packet buffer; the packet can then enter the NF as if it had come in off the wire. Second, one must *restore* the header to the packet; when packets have been modified by the NF this step may require complex algorithms to infer the correct mapping from original input packet to final output packet [32].

By using a trailer, AuditBox sidesteps these challenges in many cases. Before passing the pointer to the packet buffer into the NF, the shim adjusts the packet length to the end of the encapsulated packet, leaving the trailer at the bottom of the buffer and invisible to the NF code. Even if the packet has been modified or shortened, when the packet egresses the NF, the shim can simply restore the trailer sitting at the bottom of the allocated memory. When NFs extend the length of the packet, this overwrites the trailer and so the trailer must be restored similarly to the header operations above, however, we find for most NFs leaving the trailer at the base of the buffer is an effective way to improve performance (§7.3).

7 AuditBox Evaluation

AuditBox aims to enable real-time auditing for NFV deployments with low overhead. In this section, we evaluate its overhead using a testbed and traces and show that:

- AuditBox correctly detects a broad class of practical policy violations (§7.1).
- AuditBox enables auditing for unmodified NFs while adding less overhead than existing verification protocols (§7.2). We discuss our optimizations in §7.3.

Setup: Our testbed has four severs: three SGX servers (4-core 3.80 GHz Intel Xeon E3-1270 v6 CPUs, 64 GB RAM, Intel XL710 40Gb NICs) run AuditBox, and one server (dual-socketed Intel Xeon E5-2680 v2 GHz Xeon CPUs, with 10 cores, 128 GB RAM, Intel XL710 40Gb NICs) is used as a traffic generator. Each server runs Ubuntu 18.04 with Linux kernel 4.4.186. We use Moongen (DPDK-based) [30] to generate synthetic test traffic, as well as replay empirical traces [10] of varying packet sizes. We enable jumbo frames to allow the trailer to be added to large packets (which makes them larger than the default 1500 byte MTU). For each experiment, we report the median value of 20 tests, error bars represent one standard deviation (which in some cases are too small to see).

	Blue Team Policies	Attacks	AuditBox		OPT
			Packet	Flow	
1	Mutable packets (Fig. 2): Load balancer modifies packets	-	no	no	yes
2	Mutable packets (Fig. 2): Load balancer modifies packets	modify	yes	yes	yes
3	Dynamic paths (Fig. 3): Light IPS reroutes packets	-	no	no	yes
4	Dynamic paths (Fig. 3): Light IPS reroutes packets	reroute	yes	yes	yes
5	Stateful NFs (Fig. 4): NAT tracks flow states	reorder	-	yes	no
6	Stateful NFs (Fig. 4): NAT tracks flow states	drop	-	yes	no

Table 1: Example scenarios that use AuditBox and OPT [46] to verify whether policy violations happen; "yes"/"no" indicate whether the system reports a violation. Shaded cells are correct auditing results.

Sample NFs: AuditBox supports all existing NFs in Safebricks without any NF changes. To evaluate the performance of AuditBox, we choose three sample NFs with varying complexity: (1) *NAT* rewrites IP and TCP headers, representing NFs that modify packets (Figure 2); (2) *Stateful Firewall* which tracks connection states (Figure 4), configured with a campus ruleset (643 rules); and (3) *DPI*, which represents the most computationally expensive NF in Safebricks, configured with the Snort Community ruleset [14].

7.1 Functionality Evaluation

To validate the end-to-end effectiveness of AuditBox, we run different red-blue team exercises. In each scenario, the blue team (operator) chooses a service chain policy and the protocol (AuditBox packet, AuditBox flow, or OPT). The red team (attacker) randomly picks an attack vector. Then, we emulate this scenario with generated traffic. Finally, we reveal the ground truth and the audit trails to check if the protocol helped the blue team correctly identify/diagnose the attack.

We run these scenarios in a combination of a real testbed and a custom simulator. For the testbed we use one server to generate traffic, and three to run NFs. We introduce attacks (*e.g.,* modifying packets) using the I/O thread (Figure 9), and apply them when packets arrive at the NF. We built a custom simulator, where all NFs are connected via "one big switch" and this switch executes one or more adversarial actions (*e.g.,* rerouting) when forwarding packets between NFs.

Table 1 shows a subset of the scenarios, including the motivating examples from §2. In all scenarios, with AuditBox, the blue team successfully detects the policy violations and has correct auditing trails for doing so. OPT [46] detects some

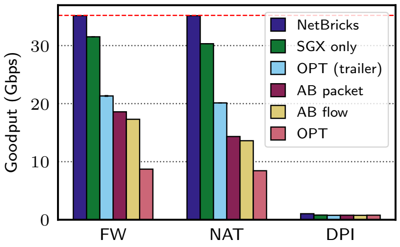


Figure 10: Cost of auditing for a single NF using empirical packet traces (AB represents AuditBox).

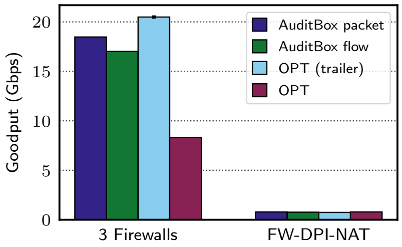


Figure 11: Cost of auditing for different chains of NFs using empirical packet traces.

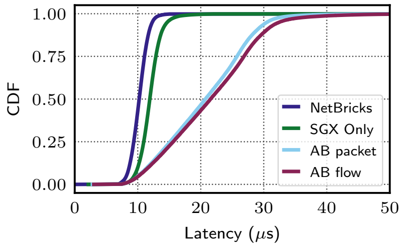


Figure 12: RTT at 80% load using empirical packet traces.

scenarios (scenarios 2 and 4), but has both false positives (1 and 3) and false negatives (5 and 6). Note that this is not unique to OPT; other VRPs [55] have the same issues as well.

7.2 Performance Evaluation

We first measure the performance of AuditBox for a single NF and a chain of NFs using empirical traffic [10]. For all tests, we run the NF using one core, and we report the goodput by excluding extra bytes used for verification. In these experiments, we compare against three alternative baselines, none of which are an apples-to-apples comparison with AuditBox (in that each was designed for a different purpose) but nonetheless we believe the comparison helps to put our results in context.

The first baseline is 'NetBricks' [59], which is a Rust-based NFV framework and does not use SGX. The second baseline is 'SGX-only', which runs NFs in enclaves but does not use any special protocol for verified routing. Our third and final baseline is OPT, which, as we have discussed (§2.4) is most closely related to AuditBox's use case but nonetheless cannot provide correct routing in the presence of dynamic routing and packet modifications.

Single NF: Figure 10 shows the goodput for running a single NF. The red dotted line shows the maximum rate of our traffic generator. NetBricks is able to process packets at the packet generator rate for both firewall and NAT. Compared to running NFs outside the enclave, the SGX baseline ('SGX only') incurs up to 15% overhead. Relative to running NFs in the enclave alone, AuditBox incurs 19% overhead for the firewall, 38% for the NAT, and 3% overhead for the DPI. The flow-level incurs slightly more overhead than the packet-level as it involves flow-table updates to track flow states.

AuditBox achieves up to two times higher goodput than the strawman OPT due to our reduced packet overhead (24B for AuditBox-pkt, 32B for AuditBox-flow vs. 84B for OPT) and our use of a trailer, instead of a header. We hypothesize that OPT's high overhead stems from its need to strip and restore large headers at each hop. To test this hypothesis, we implement OPT using our trailer optimization ('OPT-trailer'). The optimized OPT achieves a higher throughput than AuditBox, which is expected as it requires fewer MAC computations.

NF Chains: We also ran experiments to evaluate the performance of AuditBox under different NF chains across multiple nodes, similar to prior work [61]. As shown in Figure 11, for a chain of 3 firewalls AuditBox has 67% better goodput than OPT. Unlike AuditBox, which has a constant packet-size overhead regardless of chain length, OPT's header grows with chain length (116B for 3 NFs and 148B for 5 NFs), which contributes to the drop in goodput. Our optimized version, 'OPT-trailer,' gets better goodput after eliminating the header stripping and restoring overhead. We also evaluated a chain with a Firewall, followed by a DPI and a NAT. For this chain, both AuditBox and OPT achieve similar performance since the entire chain is bottlenecked by the heavy DPI.

Cost Analysis: It is worth putting the performance numbers in context to see the effective "cost" of auditability. Consider an organization deploying specialized hardware appliances for regulatory compliance. Today, this costs roughly $600-$3500 per-Gbps for firewalls and $6000-$12000 per-Gbps for IPS (*e.g.,* Cisco FirePOWER 8140 [4], Juniper Networks SRX345 [9]). If this organization shifts to NFV on commodity servers because of the auditability offered by AuditBox, we estimate the cost will be 12X-60X lower for the firewall, and 1.9X-9X lower for the IPS.While we acknowledge that any such cost analysis is fraught with uncertainties (e.g., cost at scale, reliability, service contracts, power), this rough estimate suggests that RegTech customers can still achieve financial gains through NFV.

Latency Overhead: Figure 12 shows the latency overhead of AuditBox using empirical packet traces. For each test, we measure the RTT at 80% of the maximum throughput of the system under test as a metric for latency. Compared to the SGX baseline, AuditBox-pkt adds around 18μs for 99th percentile latency, and AuditBox-flow adds another 6μs.

Sweep packet size and NF type: Figure 16 (Appendix E) shows the overhead of AuditBox for varying packet sizes and NF types. Across all NFs, the overhead of AuditBox decreases as packet sizes increase. Since NAT modifies the packet, we do not use our updatable GMAC, making the overhead of AuditBox more noticeable.

7.3 Impact of Our Optimizations

Header vs Trailer: Figure 13 shows the benefit of implementing AuditTrailer as a trailer instead of a header. Unlike our trailer which can be made invisible inside the NF (shown in Figure 9), the shim needs to strip off the added header

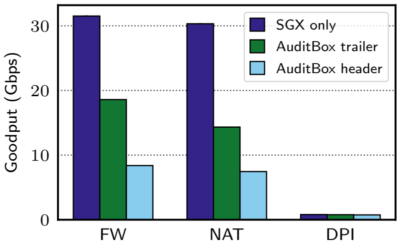


Figure 13: Effect of using trailer vs. header to carry verification data.

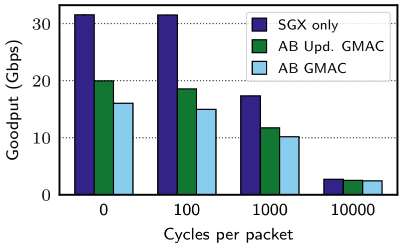


Figure 14: Effect of using updatable GMAC as NF complexity scales.

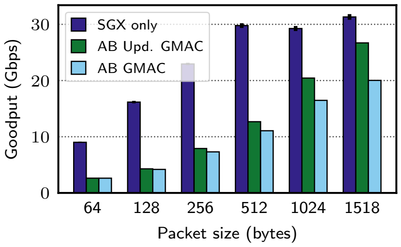


Figure 15: Effect of using updatable GMAC for varying packet sizes.

before delivering the packet to the NF to avoid modifying the NF. After the packet is processed, the shim needs to prepend the header before sending it to the next NF. These header manipulations slow down packet processing and verification, resulting in decreases of goodput by up 2×.

Regular GMAC vs Updatable GMAC: In Figures 14 and 15, we compare the performance of AuditBox when using our implementation of the updatable GMAC with the regular GMAC that is used by popular cryptographic libraries (*e.g.,* OpenSSL [12], NSS [11]). When varying the NF complexity (cycles/pkt), using updatable GMAC improves goodput by up to 23%. When varying the packet sizes, our updatable GMAC outperforms the regular GMAC by up to 25% for large packets. These improvements can largely be attributed to reusing the first MAC to compute the second MAC, which avoids the recomputing overhead for non-modified data blocks.

Dedicated log bit vs Virtual log bit: We compared our virtual logging bit with an approach with a real encrypted bit. On average, using a virtual log bit improves goodput by 4-5%.

8 Related Work

We have already discussed existing VRPs (*e.g.,* [46, 55]), service chaining policies and mechanisms (*e.g.,* [32, 58]), and how auditing is different from network verification (*e.g.,* [44, 45, 50, 60]). Here, we focus on other classes of related work.

Traditional NFV Frameworks: Prior NFV [6] frameworks focus on management [35, 42, 58, 65, 67], performance [28, 43, 51, 79], and programability [23, 48, 59]. We argue auditability should be added as a first-order feature of the NFV framework, which would relieve enterprises' concerns about deploying the NFV for security-critical services or outsourcing them to a third-party provider [33, 68].

Securing NFs: Prior work has proposed the use of SGX to protect an NF's source code [61, 72], an NF's state [69], traffic metadata [29], and to support end-to-end encryption [39, 56]. While these enhance security for NFs, they only focus on individual NFs on a single server, and they do not provide mechanisms to audit the entire service chain.

NF Verification: NF verification [27, 75, 76, 78] guarantees that a certain NF implementation is correct (memory-safe, crash-free, *etc.*). AuditBox assumes that vendors have 'certified' NF implementations as secure, and we expect this class of work would strengthen trust in such vendor certifications.

Verifiable fault localization and measurements: In addition to VRPs, auditing is also related to prior work on secure fault localization (*e.g.,* [81, 82]), robust sampling algorithms (*e.g.,* [57]), verifiable performance measurements (*e.g.,* [16]), and secure network provenance (SNP) (*e.g.,* [37, 84]). Some of our building blocks share conceptual similarity with these efforts. However, they focus on different goals. For example, SNP leverages tamper-evident logging [38] to identify misbehavior offline, unlike the runtime guarantees we provide.

9 Conclusions

In this paper, we have presented AuditBox, a framework that brings NFV to security-critical environments that require auditing. By leveraging enclaves to run NFs and continuously verifying the traffic between NFs, AuditBox provides a strong run-time guarantee that the NFV system remains in compliance with policy goals. AuditBox also supports traditional offline auditing by generating audit trails for manual inspection.

We see AuditBox as a first step in a conversation with regulators. Would our audit trails be more trustworthy if they included additional data? Should we combine our SGX protection with formal verification of the NF code [27]? In practice, should AuditBox be combined with formal verification of operator policies [60]? We expect, and hope, to see considerable evolution in supporting the RegTech space beyond AuditBox's current capabilities.

In the long run, adoption only comes when human auditors *feel comfortable* trusting the guarantees provided by any particular system. Our hope is that by not only replicating the capabilities that auditors have today, but also strengthening SFCs with runtime correctness guarantees, AuditBox will merit the trust of auditors and hence hasten the adoption of NFV in security-sensitive networks.

Acknowledgements: We thank our shepherd Alex Snoeren and the anonymous reviewers for their insightful comments. We also thank Rishabh Poddar, Limin Jia and Sze Yiu Chaus. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, ERDF Project AIDA (POCI-01-0247-FEDER-045907), in part by the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521, and by NSF awards 1440056 and 1440065.

References

[1] ARM TrustZone. https://developer.arm.com/technologies/trustzone.

[2] AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&T%20Domai%202.0%20Vision%20White%20Paper.pdf.

[3] Blue Planet NFV Service Orchestration. https://www.blueplanet.com/products/nfv-orchestration.html.

[4] Cisco-FirePOWER-8140. https://www.cdw.com/product/Cisco-FirePOWER-8140-security-appliance.

[5] Data Plane Development Kit (DPDK). http://www.dpdk.org/.

[6] European Telecommunications Standards Institute. NFV whitepaper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[7] Family Educational Rights and Privacy Act (FERPA). https://www2.ed.gov/policy/gen/guid/fpco/ferpa/index.html.

[8] HIPAA Security Rule. https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html.

[9] Juniper Networks SRX345 Services Gateway . https://www.cdw.com/product/juniper-networks-srx345\-services-gateway-security-appliance/4739102?pfm=srh.

[10] Malware Capture Facility Project. https://www.stratosphereips.org/datasets-normal.

[11] Network Security Services. https://hg.mozilla.org/projects/nss.

[12] OpenSSL-Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[13] Payment Card Industry Security Standards Council. https://www.pcisecuritystandards.org/.

[14] Snort Community Rulesets. https://www.snort.org/downloads.

[15] ISO/IEC 27001 Information Security Management. https://www.iso.org/isoiec-27001-information-security.html, 2013.

[16] Katerina J. Argyraki, Petros Maniatis, and Ankit Singla. Verifiable network-performance measurements. In *Proceedings of the 2010 ACM Conference on Emerging Networking Experiments and Technology, CoNEXT 2010, Philadelphia, PA, USA, November 30 - December 03, 2010*, page 1. ACM, 2010.

[17] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[18] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[19] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology*. CRYPTO, 1993.

[20] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[21] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. Jan 2020.

[22] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[23] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016.

[24] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXSpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[25] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086 (2016). https://datatracker.ietf.org/doc/draft-brockners-proof-of-transit/.

[26] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[27] Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. *Communications of the ACM*, 2015.

[28] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[29] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. Lightbox: Full-stack protected stateful middlebox at lightning speed. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, 2019.

[30] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, pages 275–287. ACM, 2015.

[31] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. Buzz: Testing context-dependent policies in stateful networks. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, 2016.

[32] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[33] Seyed Kaveh Fayazbakhsh, Michael K Reiter, and Vyas Sekar. Verifiable network function outsourcing: requirements, challenges, and roadmap. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*, pages 25–30, 2013.

[34] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[35] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: enabling innovation in network function control. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, 2014.

[36] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[37] Andreas Haeberlen, Ioannis Avramopoulos, Jennifer Rexford, and Peter Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI'09)*, Apr 2009.

[38] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct 2007.

[39] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module. In *Proceedings of the First Asia-Pacific Workshop on Networking - APNet'17*, 2017.

[40] Gael Hofemeier and Robert Chesebrough. Introduction to Intel AES-NI and Intel Secure Key instructions. `https://software.intel.com/en-us/articles/introduction-to-intel\-aes-ni-and-intel-secure-key-instructions`, 2014.

[41] Joint Task Force Transformation Initiative. Security and privacy controls for federal information systems and organizations. Technical Report Special Publication 800-53 (Revision 4), NIST, April 2013.

[42] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, HotMiddlebox@SIGCOMM 2015, London, United Kingdom, August 21, 2015*, 2015.

[43] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[44] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[45] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[46] Tiffany Hyun-Jin Kim, Cristina Basescu, Limin Jia, Soo Bum Lee, Yih-Chun Hu, and Adrian Perrig. Lightweight source authentication and path validation. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[47] Markus Legner, Tobias Klenze, Marc Wyss, Christoph Sprenger, and Adrian Perrig. Epic: Every packet is checked in the data plane of a path-aware internet. In *29th USENIX Security Symposium*, USENIX Security '20, 2020.

[48] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K. K. Ramakrishnan, and Timothy Wood. Microboxes: High performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[49] Xin Liu, Ang Li, Xiaowei Yang, and David Wetherall. Passport: Secure and adoptable source authentication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, 2008.

[50] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, 2011.

[51] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, et al. ClickOS and the art of network function virtualization. In *USENIX NSDI*, 2014.

[52] D. McGrew. Efficient authentication of large , dynamic data sets using Galois/counter mode (GCM). In *In Security in Storage Workshop*. IEEE, 2005.

[53] D. McGrew and J. Viega. The Galois/counter mode of operation (GCM). In *Submission to NIST Modes of Operation Process*, 2004.

[54] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[55] Jad Naous, Michael Walfish, Antonio Nicolosi, David Mazières, Michael Miller, and Arun Seehra. Verifying and enforcing network paths with icing. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, 2011.

[56] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And then there were more: Secure communication for more than two parties. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, 2017.

[57] Pavlos Nikolopoulos, Christos Pappas, Katerina J. Argyraki, and Adrian Perrig. Retroactive packet sampling for traffic receipts. *POMACS*, 3(1):19:1–19:39, 2019.

[58] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for NFV applications. In *SOSP '15*. ACM, 2015.

[59] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 203–216, Savannah, GA, November 2016. USENIX Association.

[60] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying Reachability in Networks with Mutable Datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[61] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[62] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cedric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[63] Zafar Ayyub Qazi, Rui Miao, Cheng-Chun Tu, Vyas Sekar, Luis Chiang, and Minlan Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, 2013.

[64] P. Quinn, U. Elzur, and C. Pignataro. Network Service Header (NSH). RFC 8300, January 2018.

[65] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, 2013.

[66] Grand View Research. RegTech Market Size Worth $55.28 Billion by 2025. `https://www.bloomberg.com/press-releases/2019-08-14/regtech-market-size-worth-55-28-billion-by-2025\-cagr-52-8-grand-view-research-inc`, Aug 2019.

[67] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[68] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the 2012 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.

[69] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing nfv states by using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFV Security, 2016.

[70] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[71] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. In *IACR Cryptology*. ePrint Archive, 2004.

[72] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the Symposium on SDN Research - SOSR '18*, 2018.

[73] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[74] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. NetSMC: A custom symbolic model checker for stateful network verification. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, 2020.

[75] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th Symposium on Operating Systems Principles - SOSP '19)*, 2019.

[76] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17*, 2017.

[77] Fuyuan Zhang, Limin Jia, Cristina Basescu, Tiffany Hyun-Jin Kim, Yih-Chun Hu, and Adrian Perrig. Mechanized Network Origin and Path Authenticity Proofs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, 2014.

[78] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI '20, 2020.

[79] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A platform for high performance network service chains. In *HotMiddlebox*. ACM, 2016.

[80] Xin Zhang, Abhishek Jain, and Adrian Perrig. Packet-dropping adversary identification for data plane security. In *Proceedings of the 2008 ACM CoNEXT Conference on - CONEXT '08*, 2008.

[81] Xin Zhang, Chang Lan, and Adrian Perrig. Secure and Scalable Fault Localization under Dynamic Traffic Patterns. In *2012 IEEE Symposium on Security and Privacy*. IEEE, May 2012.

[82] Xin Zhang, Zongwei Zhou, Geoff Hasker, Adrian Perrig, and Virgil Gligor. Network fault localization with small TCB. In *2011 19th IEEE International Conference on Network Protocols*. IEEE, 2011.

[83] Xin Zhang, Zongwei Zhou, Hsu-Chun Hsiao, Tiffany Hyun-Jin Kim, Adrian Perrig, and Patrick Tague. ShortMAC: Efficient Data-Plane Fault Localization. In *19th Annual Network and Distributed System Security Symposium (NDSS)'12*, 2012.

[84] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*, 2011.

A Modeling Duplicate Packets

The formalization of Property 4 assumes that the same packet is never sent twice. This seems to be a reasonable assumption. As we are forwarding under L2, an NF_A and an NF_B will always have different Ethernet headers even if the IP, TCP/UDP, and Payload are identical. And it seems reasonable to assume that an NF_A will never transmit the same packet twice either – even a re-transmitted TCP packet will come with a different IPID value.

Nonetheless, we could remove these requirements and allow NFs to transmit the same packet repeatedly by defining correctness as follows:

Let $\textsc{send-to}_{F,P}(E) \rightarrow \{0,1\}$ be defined:

```
function $\textsc{send-to}_{\mathrm{NF}_i,p_i}(e)$
    if $e.\mathrm{op} = \textsc{send} \wedge e.\mathrm{pkt} = p_i \wedge \mathrm{policy}(p_i, e.\mathrm{NF}) = \mathrm{NF}_i$ then
        return 1
    else
        return 0
```

Let $\textsc{recv-pkt}_{F,P}(E) \rightarrow \{0,1\}$ be defined:

```
function $\textsc{recv-pkt}_{\mathrm{NF}_i,p_i}(e)$
    if $e.\mathrm{op} = \textsc{recv} \wedge e.\mathrm{pkt} = p_i \wedge e.\mathrm{NF} = \mathrm{NF}_i$ then
        return 1
    else
        return 0
```

To allow identical packets, we could then say the system was correct under packet correctness iff:

$$\forall e \in E \text{ s.t. } e.\mathrm{op} = \textsc{recv}:$$

$$\sum_{i=1}^{e.\mathrm{t}} \textsc{send-to}_{e.\mathrm{NF},e.\mathrm{pkt}}(E[i]) = \sum_{i=1}^{e.\mathrm{t}} \textsc{recv-pkt}_{e.\mathrm{NF},e.\mathrm{pkt}}(E[i])$$

B Additional Definitions

B.1 Modeling Packet Exit

We define GW_{out} for when a packet exits the cluster as follows:

Algorithm 3 Model of Packets Exiting the Cluster

```
function $GW_{\mathrm{OUT}}$(input)
    input $\leftarrow f_{in}(input)$
    if input $\neq \perp$ then
        $E$.append(input, $\perp$, $\mathrm{GW_{out}}$, $E$.length + 1)
    return input
```

B.2 Audit Trail Definition

While our routing protocol provides hop-by-hop guarantees, auditors are familiar with end to end 'what you see is what you get' *evidence* that packets are indeed following the correct route. To provide auditors the confidence of proven correctness with empirical evidence, AuditBox provides empirical evidence in the form of audit trails. Following the model of our event log E, one can take a packet in any of its states (prior to entry, between two NFs, post exit) and compute the forms the packet took on and all NFs it traversed across its entire traversal of the system. While 5.4 describes how this works in practice, we describe audit trails in the context of our model with Algorithm 4.

Algorithm 4 Audit Trail Definition

```
function CAUSED-BY(event)
    if event.pkt_in = ⊥ then
        return [(event.NF, event.pkt_out)]
    trail ← (event.pkt_in, event.NF, event.pkt_out)
    prev ← get-event_E(event.pkt_in, pkt_out)
    return caused-by(prev) + trail
function LEADS-TO(event)
    if event.pkt_out = ⊥ then
        return [(event.pkt_in, event.NF)]
    prev ← get-event_E(event.pkt_out, pkt_in)
    prev_trail ← (prev.pkt_in, prev.NF, prev.pkt_out)
    return prev_trail + leads-to(prev)
function AUDIT-TRAIL(event)
    return CAUSED-BY(event) + LEADS-TO(event)
```

Note that the definition assumes that all transmitted packets are unique.

C Pseudocode

In the algorithms below, we expand the notion of a packet to also include the AuditBox trailer.

In the flow-verification algorithm (Algorithm 6), we assume each NF, including the gateway, maintains a flow-counter table FC which maps a flow ID and destination NF to a counter value:

$$ctr \leftarrow FC[flowID, NF]$$

Performing a lookup with a new $flowID, NF$ pair implicitly initializes the counter to zero. Comments highlight differences relative to the packet-verification algorithm (Algorithm 5).

Algorithm 5 Packet Verification Protocol

```
Input: The shared symmetric key $K_\sigma$ for the current epoch $\sigma$.
▷ Generate an AuditTrailer for each packet at the gateway $GW_{in}$
function GENERATE(pkt)
    $out.pkt = pkt$
    $out.pktID = genPktID(pkt)$
    $out.srcNF = GW_{in}$
    $out.dstNF = Policy(pkt, GW_{in})$
    $out.tag = MAC_{K_\sigma}(out.pkt|out.pktID|out.srcNF|out.dstNF)$
    return out

▷ Process a packet in at $NF_i$
function PROCESS_i(in)
    ok ← $in.dstNF = NF_i$ ∧
            $Verify_{K_\sigma}(in.pkt|in.pktID|in.srcNF|in.dstNF, in.tag)$
    if ok then:
        $out.pkt = f_i(in.pkt)$
        $out.pktID = in.pktID$
        $out.srcNF = NF_i$
        $out.dstNF = Policy(out.pkt, NF_i)$
        $out.tag = MAC_{K_\sigma}(out.pkt|out.pktID|out.srcNF|out.dstNF)$
    else
        out ← ⊥        ▷ Drop packet and raise alert
    return out
```

Algorithm 6 Flow Verification Protocol

```
Input: The shared symmetric key $K_\sigma$ for the current epoch $\sigma$.
▷ Generate a flow AuditTrailer for each packet at the gateway $GW_{in}$
function GENERATE(pkt)
    ▷ Same as Packet Verification
    $out.pkt = pkt$
    $out.pktID = genPktID(pkt)$
    $out.srcNF = GW_{in}$
    $out.dstNF = Policy(pkt, GW_{in})$
    ▷ New for Flow Verification
    $out.flowID = computeFlowID(pkt)$
    $out.seqNum = FC[out.flowID, out.dstNF]{+}{+}$
    $out.tag = MAC_{K_\sigma}(out.pkt|out.pktID|out.srcNF|out.dstNF$
            $|out.flowID|out.seqNum)$
    return out

▷ Process a packet in at $NF_i$
function PROCESS_i(in)
    ok ← $in.dstNF = NF_i$
        ∧  $Verify_{K_\sigma}(in.pkt|in.pktID|in.srcNF|in.dstNF|in.flowID$
                $|in.seqNum, in.tag)$
        ∧  $in.seqNum = FC[in.flowID, in.srcNF]$     ▷ New
    if ok then:
        $FC[in.flowID, in.srcNF]{+}{+}$    ▷ New
        $out.pkt = f_i(in.pkt)$
        $out.pktID = in.pktID$
        $out.srcNF = NF_i$
        $out.dstNF = Policy(out.pkt, NF_i)$
        ▷ New for Flow Verification
        $out.flowID = in.flowID$
        $out.seqNum = FC[out.flowID, out.dstNF]{+}{+}$
        $out.tag = MAC_{K_\sigma}(out.pkt|out.pktID|out.srcNF|out.dstNF$
                $|out.flowID|out.seqNum)$
    else
        out ← ⊥        ▷ Drop packet and raise alert
    return out
```

D Security Proofs

D.1 Cryptographic Assumptions

We introduce the standard notation we use and the standard cryptographic assumptions we make.

We write $x|y$ for the uniquely delimited (either via tags or fixed widths) concatenation of x and y. Hence, $x_0|y_0 == x_1|y_1$ implies $x_0 == x_1$ and $y_0 == y_1$.

Our scheme relies on a Message Authentication Code (MAC) scheme, which consists of three algorithms. A symmetric key K is generated by the KeyGen() algorithm. We write $\tau \leftarrow \text{MAC}_K(m)$ to indicate using key K to compute a MAC *tag* τ on message m, and $\text{Verify}_K(m,\tau)$ for the algorithm that uses key K to check the validity of tag τ for message m.

We assume that the MAC scheme is existentially unforgeable under chosen-message attacks (EUF-CMA) [21]. Intuitively, the definition says that an adversary who can request validly computed tags for n adaptively chosen messages cannot produce a new pair $(m,\tau) \notin \{(m_1,\tau_1),\ldots,(m_n,\tau_n)\}$ which passes $\text{Verify}_K(m,\tau)$. Standard algorithms, such as HMAC [20] and GMAC [53], are EUF-CMA secure.

D.2 Security Definition

To formalize AuditBox's security, we take the standard approach of defining our desired security property via a cryptographic *game* involving a challenger $\mathcal{C}$, and a probabilistic, polynomial-time adversary $\mathcal{A}$, which intuitively corresponds to the untrusted network, Φ. The game can be instantiated with an *audit protocol* that supplies NF functions f_i.

The challenger begins the game by creating an empty event log E and calling KeyGen() to produce key K. The adversary is then allowed to run and can call the following oracles which represent the various NFs in the system.

1. $p_{out} \leftarrow \text{Oracle-GW}_{IN}(p_{in})$ allows the adversary to introduce a new packet p_{in} to the gateway and obtain the packet p_{out} produced by the gateway.
2. $p_{out} \leftarrow \text{Oracle-NF}_i(p_{in})$ invokes NF_i on the adversarially supplied input packet p_{in} and gives the adversary the resulting packet p_{out}.

The challenger instantiates these oracles using the models described in §4.1. Specifically, $\text{Oracle-GW}_{IN}(p_{in})$ runs Algorithm 2 using the protocol-supplied function f_{in}, and $\text{Oracle-NF}_i(p_{in})$ runs Algorithm 1 using the protocol-supplied function f_i. Note that both oracles append to the event log E

When the adversary terminates, the game ends and outputs E.

D.3 Security Proof of Packet Correctness

Theorem 1 (Packet Correctness) *Consider the game described above with adversary $\mathcal{A}$ and instantiated with the AuditBox packet correctness protocol (§5.2). Specifically, looking at Algorithm 5, we instantiate f_{in} with the function* GENERATE *and f_i with* PROCESS$_i$*. The probability that the game outputs an event log E that violates Property 1 is negligible.*

Proof of Theorem 1: We prove security by reducing the security of our protocol to the EUF-CMA security of our MAC algorithm though a series of cryptographic games [19, 71]. The initial game matches the game defined in §D.2, and each subsequent game idealizes a portion of the protocol. At each step, we calculate the adversary's success in distinguishing between the two games.

Game 0 is defined as in §D.2 with an adversary $\mathcal{A}$ which queries its oracles a total of α times.

Game 1 is the same as above, except that the NF oracle, when given an input packet p_{in}, computes

$$m \leftarrow p_{in}.pkt|p_{in}.pktID|p_{in}.srcNF|p_{in}.dstNF$$

and immediately rejects the packet if m was not previously passed as an argument to $\text{MAC}_K(\cdot)$ (*i.e.,* $\mathcal{C}$ keeps a list $\mathcal{L}$ of all values passed to $\text{MAC}_K(\cdot)$, and upon receiving a packet p_{in} on which it would normally call $\text{Verify}_K(m,\tau)$, it looks up m in $\mathcal{L}$ and accepts/rejects based on the lookup, without looking at τ).

The adversary can distinguish Game 1 from Game 0 only if it can forge a valid tag τ for m. This happens with probability at most $\texttt{EUF-CMA}(\alpha)$, the probability of breaking our unforgability assumption given at most α chosen-message tags.

From Game 1, we show that Property 1 perfectly holds, which means that the probability that an adversary can break Property 1 is at most $\texttt{EUF-CMA}(\alpha)$, which is, by definition, negligible when we employ a secure MAC scheme.

In Game 1, consider any $e_b \in E$ such that $e_b.pkt_{in} \neq \perp$. By the construction of the NF model (Algorithm 1) an event e_b is only added to E after the NF runs f_i, which we instantiated with PROCESS$_i$. PROCESS$_i$ ensures $e_b.pkt_{in}.dstNF == e_b.NF$ (Line 12 of Algorithm 5). In Game 1, instead of running the MAC's verification algorithm on Line 13, it checks that the computed m is on the list $\mathcal{L}$ of previously MAC'ed messages. For this check to succeed, there must have been a logically earlier MAC call, which must have occurred during a previous invocation of Oracle-GW_{IN} or Oracle-NF_i, which each compute a tag on their outbound packet. For the verification lookup in $\mathcal{L}$ to succeed, that MAC call must have computed an m' for its output packet, where $m' == m$. This earlier oracle invocation would have produced event $e_a = (p, e_b.pkt_{in}, e_b.pkt_{in}.srcNF, i)$ for some other input packet p and index i, with $i < e_b.t$, since this was an earlier invocation, and the log E was necessarily shorter. Hence, the equality of m and m' implies that we have $e_a.pkt_{out} = e_b.pkt_{in}$, and $policy(e_a.pkt_{out}, e_a.NF) = e_b.NF$ (note Lines 6 and 18 of Algorithm 5), satisfying Property 1.

■

D.4 Security Proof of Flow Correctness

Theorem 2 (Flow Correctness) *Consider the game described in §D.2 with adversary $\mathcal{A}$ and instantiated with the AuditBox flow-correctness protocol (§5.3). Specifically, looking at Algorithm 6, we instantiate f_{in} with the function* GENERATE *and f_i with* PROCESS$_i$*. The probability that the game outputs an event log E that violates Properties 1-4 is negligible.*

Proof of Theorem 2:

We prove Theorem 2 by considering each property in turn.

Proof of Property 1 We begin by observing that compared with the packet-verification protocol, the flow-verification protocol

1. Includes in its AuditTrailer a superset of the packet-verification fields.
2. Computes the values the packet-verification fields in an identical manner.
3. Computes MACs over a superset of the packet-verification fields. algor
4. Performs a superset of the validation checks (compare Lines 16-18 of Algorithm 6 with Lines 12-13 of Algorithm 5).

Hence, we can apply an identical set of arguments as we did in our proof of Theorem 1 to show that Property 1 still holds when we employ a secure MAC. In the discussion, Property 1 allows us to assume that all packets in E originated from an NF, and hence we no longer need worry about adversarially mangled or injected packets.

Proof of Property 2 To show that Property 2 (*i.e.,* no packet injection or modification) holds, choose an arbitrary $\text{NF}_a, \text{NF}_b \in F$, and $e_{a1}, e_{a2}, e_{b2} \in E$ such that Lines 5-9 of Property 2 hold. We will show that there must exist $e_{b1} \in E$ such that

$$e_{b1}.\text{t} < e_{b2}.\text{t} \wedge e_{b1}.\text{NF} = \text{NF}_b \wedge e_{b1}.\text{pkt}_{\text{in}} = e_{a1}.\text{pkt}_{\text{out}}$$

When $e_{a1}.\text{pkt}_{\text{out}}$ was produced by $\textsc{Process}_a$, it was assigned a flow sequence number $FC_a[e_{a1}.\text{pkt}_{\text{out}}.\mathit{flowID},\ e_{a1}.\text{pkt}_{\text{out}}.\mathit{srcNF}]$ (Line 27 of Algorithm 6), and NF_a immediately increments the counter.

When we subsequently call $\textsc{Process}_a$ to produce $e_{a2}.\text{pkt}_{\text{out}}$ (we know this is a subsequent invocation of $\textsc{Process}_a$ because $e_{a1}.\text{t} < e_{a2}.\text{t}$), we can show that it will read the same counter from FC_a, which by observation always increases monotonically. Hence, it must be that $e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum} > e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum}$.

We can show that $\textsc{Process}_a$ accesses the same counter by showing that $e_{a1}.\text{pkt}_{\text{out}}.\mathit{flowID} = e_{a2}.\text{pkt}_{\text{out}}.\mathit{flowID} \wedge e_{a1}.\text{pkt}_{\text{out}}.\mathit{srcNF} = e_{a2}.\text{pkt}_{\text{out}}.\mathit{srcNF}$. The first is straightforward, since $e_{a1}.\text{pkt}_{\text{out}} = e_{a2}.\text{pkt}_{\text{out}} \implies \mathit{flow}(e_{a1}) = \mathit{flow}(e_{a2})$, and the second follows from Line 7 of Property 2.

For $e_{a2}.\text{pkt}_{\text{out}}$ to have passed the "ok" check in $\textsc{Process}_b$, and hence to have generated e_{b2}, it must have passed the check on Line 18, which means that at the time, $e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum} = FC_b[e_{a2}.\text{pkt}_{\text{out}}.\mathit{flowID},\ e_{a2}.\text{pkt}_{\text{out}}.\mathit{srcNF}]$. Now consider the set $E_{b1} \subseteq E$ of all e_{b1} such that $e_{b1}.\text{t} < e_{b2}.\text{t} \wedge e_{b1}.\text{NF} = \text{NF}_b \wedge e_{b1}.\text{pkt}_{\text{in}}.\mathit{srcNF} = \mathit{NF}_a \wedge \mathit{flow}(e_{b1}) = \mathit{flow}(e_{a1})$; *i.e.,* all previous events generated by NF_b that came from NF_a and are part of the same flow as e_{a1} (and hence e_{a2} and e_{b2}). The crucial observation is that each such e_{b1} increments $FC_b[e_{b1}.\text{pkt}_{\text{in}}.\mathit{flowID},\ e_{b1}.\text{pkt}_{\text{in}}.\mathit{srcNF}]$, and these are the *only* events that do so prior to $e_{b2}.\text{t}$. Hence, it must be the case that $|E_{b1}| = e_{b2}.\text{pkt}_{\text{in}}.\mathit{seq}$, and each $e_{b1} \in E_{b1}$ has a unique sequence number (as guaranteed by the monotonically increasing counter value). Since we know that $0 \leq e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum} < e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum} = e_{b2}.\text{pkt}_{\text{in}}.\mathit{seqNum}$, there must be an $e_{b1} \in E_{b1}$ with $e_{b1}.\text{pkt}_{\text{in}}.\mathit{seqNum} = e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum}$. Furthermore, the monotonic counter on the sending side (NF_a) guarantees that NF_a must have only assigned the sequence number $e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum}$ to a single packet, namely $e_{a1}.\text{pkt}_{\text{in}}$. Since we have proven Property 1 holds (*i.e.,* the attacker cannot inject or modify packets), if NF_b received a packet $e_{b1}.\text{pkt}_{\text{in}}$ with sequence number $e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum}$ from NF_a, it must be the case that $e_{a1}.\text{pkt}_{\text{out}} = e_{b1}.\text{pkt}_{\text{in}}$. Hence, we have identified an e_{b1} such that $e_{b1}.\text{t} < e_{b2}.\text{t} \wedge e_{b1}.\text{NF} = \text{NF}_b \wedge e_{b1}.\text{pkt}_{\text{in}} = e_{a1}.\text{pkt}_{\text{out}}$, proving that Property 2 holds.

Proof of Property 3 To show that Property 3 (no packet reordering) holds, choose an arbitrary $\text{NF}_a, \text{NF}_b \in F$ and $e_{a1}, e_{b1}, e_{a2}, e_{b2} \in E$ such that Lines 14-17 of Property 3 hold. Suppose for the sake of contradiction that $e_{a1}.\text{t} < e_{a2}.\text{t}$ but $e_{b1}.\text{t} \geq e_{b2}.\text{t}$ (*i.e.,* NF_b received the packets in reverse order). Since each NF increments its flow counter after sending a packet (Lines 10 and 27 in Algorithm 6), and we know that $\text{flow}(e_{a1}) = \text{flow}(e_{a2})$, it must be the case that $e_{a1}.\text{pkt}_{\text{in}}.\mathit{seqNum} < e_{a2}.\text{pkt}_{\text{in}}.\mathit{seqNum}$. Since we supposed $e_{b1}.\text{t} \geq e_{b2}.\text{t}$ (and $e_{b1}.\text{t} \neq e_{b2}.\text{t}$ because each entry in E has a unique position) it must be the case that $\textsc{Process}_b$ was called on $e_{a2}.\text{pkt}_{\text{out}}$ before it was called on $e_{a1}.\text{pkt}_{\text{out}}$. For $e_{a2}.\text{pkt}_{\text{out}}$ to have passed the "ok" check in $\textsc{Process}_b$, it must have passed the check on Line 18, which means that at the time, $e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum} = FC_b[e_{a2}.\text{pkt}_{\text{out}}.\mathit{flowID},\ e_{a2}.\text{pkt}_{\text{out}}.\mathit{srcNF}]$. The counter is then incremented on Line 20, and continues to increase monotonically on subsequent invocations. Hence, when $\textsc{Process}_b$ was later called on $e_{a1}.\text{pkt}_{\text{out}}$, to have passed the "ok" check, it must be the case that $e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum} > e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum}$. This contradicts our starting point that $e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum} < e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum}$. Hence we can conclude that if $e_{a1}.\text{t} < e_{a2}.\text{t}$ then $e_{b1}.\text{t} < e_{b2}.\text{t}$.

To prove the other direction of the implication, namely that if $e_{a1}.\text{t} \geq e_{a2}.\text{t}$ then $e_{b1}.\text{t} \geq e_{b2}.\text{t}$. we can apply the same argument as above, swapping $a2$ for $a1$ and $b2$ for $b1$.

Proof of Property 4 To show that Property 4 (no packet replay) holds, choose an arbitrary $e_a \in E$. Suppose for the sake of contradiction that $\exists e_b \in E$ such that $e_a.\text{pkt}_{\text{in}} = e_b.\text{pkt}_{\text{in}}$. Since each entry in E has a unique position, we know $e_a.\text{t} \neq e_b.\text{t}$, so without loss of generality, assume $e_a.\text{t} < e_b.\text{t}$. Since we know that $e_a.\text{pkt}_{\text{in}}.\mathit{dstNF} = e_b.\text{pkt}_{in}.\mathit{dstNF}$, both entries must have been produced by invocations of $\textsc{Process}_{e_a.\text{pkt}_{\text{in}}.\mathit{dstNF}}$, and it was invoked on $e_a.\text{pkt}_{\text{in}}$ before

$e_b.\text{pkt}_{\text{in}}$ (because $e_a.\text{t} < e_b.\text{t}$). Since PROCESS did not raise an alert on $e_a.\text{pkt}_{\text{in}}$, it must be the case that it passed the check on Line 18, which means that at the time, $e_a.\text{pkt}_{\text{in}}.seqNum = FC_b[e_a.\text{pkt}_{\text{in}}.flowID, e_a.\text{pkt}_{\text{in}}.srcNF]$. The counter is then incremented on Line 20, and continues to increase monotonically on subsequent invocations. Hence, when PROCESS was later called on $e_b.pkt$, to have passed the "ok" check, it must be the case that $e_b.\text{pkt}_{\text{in}}.seqNum > e_a.\text{pkt}_{\text{in}}.seqNum$. But we supposed that $e_a.\text{pkt}_{\text{in}} = e_b.\text{pkt}_{\text{in}}$, which means $e_b.\text{pkt}_{\text{in}}.seqNum = e_a.\text{pkt}_{\text{in}}.seqNum$. This contradiction show that $e_b.\text{pkt}_{\text{in}}.seqNum \neq e_a.\text{pkt}_{\text{in}}.seqNum$. ■

D.5 Security of Secret Logging

As described in §6.1, AuditBox implements secret logging via a *virtual* bit that is appended to the real data carried in the AuditTrailer when computing a MAC. For this approach to effectively sample packets even in the presence of an adversary, it should be computationally difficult to distinguish packets with the virtual bit set to one (indicating a packet that should be logged) from those with the virtual bit set to zero.

Security Definition We capture this security notion with the following game involving a challenger $\mathcal{C}$, and a probabilistic, polynomial-time adversary $\mathcal{A}$. When the game begins, $\mathcal{C}$ chooses a random bit b and then runs KeyGen() to produce key K. The adversary is then allowed to run and given access to Oracle-MAC($\cdot$), which when given a message m, returns $\text{MAC}_K(m|b)$. $\mathcal{A}$ eventually terminates and outputs its guess b'. The game returns 1 if $b == b'$ and 0 otherwise.

We define $\mathcal{A}$'s advantage after making α queries to its oracle as $2\left|P - \frac{1}{2}\right|$, where P is the probability that the game returns 1.

Security Proof The game above is not secure for arbitrary MACs. In other words, given a EUF-CMA secure MAC scheme $\mathcal{M}$, we can construct a new scheme $\mathcal{N}$ that is also EUF-CMA secure, but where an adversary can achieve advantage 1 in the game above. For example, we could define $\mathcal{N}.\text{MAC}_K(m) = \mathcal{M}.\text{MAC}_K(m)|m$. This is EUF-CMA secure (since the adversary still cannot produce a forgery against $\mathcal{M}$), but message secrecy is entirely broken.

Fortunately, we *can* prove security for specific MAC algorithms, including HMAC and, crucially for our implementation, GMAC. In particular, we leverage the fact that GMAC is defined (simplifying slightly) as:

$$\text{GMAC}_K(IV, m) = \text{PRF}_K(IV) \oplus \text{GHASH}(m)$$

Theorem 3 (Secret Logging Security) *Consider the game described above with adversary $\mathcal{A}$. When the MAC algorithm is GMAC, the adversary's advantage is negligible.*

Proof of Theorem 3: We prove security via the following two games.

Game 0 is defined as described above with an adversary $\mathcal{A}$ which queries its oracle a total of α times.

Game 1 is the same as above, except that Oracle-MAC(m) returns

$$\text{GMAC}_K(IV, m) = R \oplus \text{GHASH}(m)$$

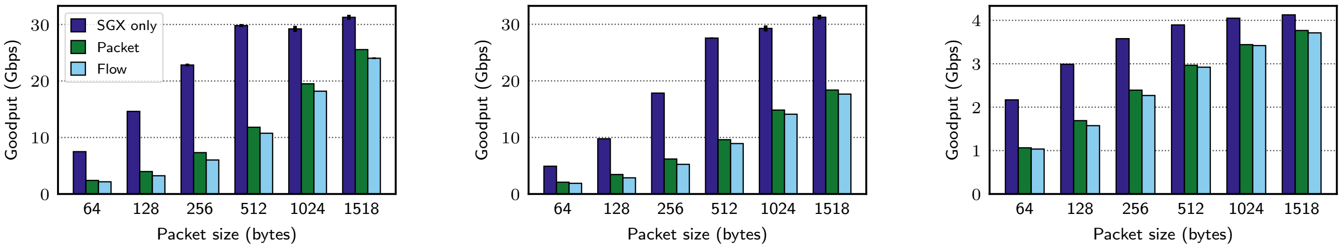
where R is a randomly sampled value.

The adversary can distinguish Game 1 from Game 0 only if it can distinguish the output of the PRF from the output of a truly randomly selected function. This happens with probability at most $\texttt{PRF}(\alpha)$, the probability of breaking the security of GMAC's PRF given at most α queries.

From Game 1, we show that the adversary has no information about the underlying message (and hence about the value of b), which means that the adversary's advantage in the security game is at most $\texttt{PRF}(\alpha)$, which is, by definition, negligible when we employ a secure PRF, which is also necessary for the standard EUF-CMA security of GMAC to hold.

In Game 1, the output of GHASH (which is the only information derived from m) is XOR'ed with a randomly chosen value of the same length. In other words, the output is the result of applying a one-time pad to GHASH(m), which is an informationally secure encryption scheme. Hence the adversary learns nothing about m from the output of its oracle. ■

E Performance Sensitivity Analysis



(a) Firewall. **(b) NAT.** **(c) DPI.**

Figure 16: Sensitivity analysis across NFs for different packet sizes.