Secure and Efficient Network Fault Localization

Xin Zhang

CMU-CS-12-104
Feburary 27, 2012

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Adrian Perrig, chair
Hui Zhang, chair
Virgil D. Gligor
David Maltz

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

© 2012 Xin Zhang

This research was supported by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389, W911NF-09-1-0273, and MURI W 911 NF 0710287 from the Army Research Office, and by support from NSF under the TRUST STC award CCF-0424422, CNS-0831440, and CNS-1040801. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, CMU, NSF or the U.S. Government or any of its agencies.

Keywords: fault localization, secure forwarding, data plane security, accountability, network reliability

PhD. Dissertation: Xin Zhang

Abstract

High-quality online services demand reliable packet delivery at the network layer. However, clear evidence documents the existence of compromised routers in ISP and enterprise networks, threatening network availability and reliability. A compromised router can stealthily drop, modify, inject, or delay packets in the forwarding path to launch Denial-of-Service, surveillance, man-in-the-middle attacks, etc. Unfortunately, current networks fail to provide any assurance of data delivery in *adversarial* environments, nor a reliable way to identify misbehaving routers that jeopardize packet delivery. Data-plane fault localization serves as an imperative building block to enhance network availability and reliability, since it localizes faulty links of misbehaving routers, enables a sender to find a fault-free path, and enforces contractual obligations among network nodes. Until recently, however, the design of secure fault localization protocols has proven to be surprisingly elusive. Existing fault localization protocols fail to achieve high security and efficiency, incur unacceptably long detection delays, and require forwarding paths to be impractically long-lived. In this dissertation, we show a suite of secure *and* efficient fault localization protocols exploring distinct dimensions in the design space of fault localization. Our key idea is to achieve a lower bound on packet forwarding correctness via fault localization by *limiting* the amount of malicious packet drops/forgeries at the data plane, instead of perfectly detecting *every single* malicious activity which tends to result in high overhead. In this way, we trap an attacker into a dilemma: if the attacker inflicts damage worse than a threshold, it will be detected, which may lead to removal from the network; otherwise, the damage is limited and thus a lower bound on data-plane packet delivery is achieved. This design principle enables the construction of efficient probabilistic algorithms and the derivation of provable performance bounds. Both the analytical and experimental results show that the proposed protocols outperform prior work by 100 to 1000 times regarding efficiency with provable security against sophisticated attackers.

Acknowledgments

I am grateful to my advisors, Adrian Perrig and Hui Zhang, for guiding me in my research, helping me to develop strong research skills, and encouraging me to to become a better person overall. I enjoyed many stimulating research discussions with Adrian and Hui, as well as casual chats about life in the US. I especially cherish my first summer at CMU when Adrian met with me almost every day to help me improve my writing skills. Adrian and Hui have been my greatest guides, friends and supporters.

I am also very grateful to the other members of my thesis committee, Professor Virgil Gligor and Dr. David Maltz, for their advice and guidance throughout the dissertation process. Their invaluable comments, guidance and encouragement dramatically helped me clarify my thesis, refine my approach, improve the algorithms, and broaden my vision of networking and security research. Seeing their work and accomplishments inspired me to become a more rigorous researcher.

I would also like to thank all my co-authors for their indispensable collaboration help: Dave Andersen, Fan Bai, Bhargav Bellur, David Bermingham, Laxmi Bhuyan, Zhiping Cai, Haowen Chan, Hao Che, Kai Chen, Yan Chen, Rituik Dubey, Virgil Gligor, Xiaohong Guan, Geoff Hasker, Hsu-Chun Hsiao, Chengchen Hu, Abhishek Jain, Tiffany Kim, Chang Lan, Soo Bum Lee, Wei Li, Yanlin Li, Dong Lin, Bin Liu, Yunhao Liu, Zhenhua Liu, Hongbin Lu, Aravind Lyer, Maggie McGinnis, Hossen Mustafa, Onur Mutlu, Derek Pao, Adrian Perrig, Ahren Studer, Patrick Tague, Thanos Vasilakos, Xiaojun Wang, Zhijun Wang, Ying Xi, Wenyuan Xu, Baohua Yang, Mu Yang, Hui Zhang, Yue Zhang, Kai Zheng, Jian Zhou, Peng Zhou, and Zongwei Zhou.

I greatly appreciate the guidance and assistance of my mentors Dr. Yinglian Xie and Dr. Fang Yu at Microsoft Research, and Dr. Sylvia Ratnasamy and Dr. Gianluca Iannaccone at Intel

Research during my internships at their institutions.

Finally, I dedicate this dissertation to my wife, Jiayao Han, for her continuous love and support, and my daughter, Jinjin Zhang, for the ultimate joy and happiness she brings to me.

Contents

1 Introduction **1**

1.1 What is Fault Localization? 1

1.2 Challenges and Insights 3

1.3 Dissertation Overview 5

2 Thesis, Problem Statements, Metrics, and Assumptions **9**

2.1 General Thesis 9

2.2 Scope and Assumptions 10

2.3 Attacker Model 12

2.4 Problem Formulation 13

2.5 Metrics 14

3 Security Challenges **17**

3.1 Challenge 1: Acknowledgment-based Approach 17

3.2 Challenge 2: Sophisticated Packet Modification Attacks 18

3.3 Challenge 3: Colluding attacks 18

4 PAAI **21**

4.1 Introduction 22

4.2 Setting 23

4.3 A Strawman Approach: Full-ack 24

4.4 Overview of PAAI 25

4.5 The PAAI Protocols 28
4.5.1 PAAI-1 28
4.6 PAAI-2 30
4.7 Security Properties 32
4.8 Theoretical Analysis 34
4.8.1 Bounding Malicious End-to-End Corruption Rate 34
4.8.2 Detection Delay 36
4.8.3 Communication Overhead 37
4.8.4 Storage Overhead 38
4.9 Simulation Results and Analysis 39
4.9.1 Methodology 39
4.9.2 Results and Analysis 40
4.10 Summary of Results 44
4.11 Combination 45
4.12 Summary 47

5 ShortMAC **49**

5.1 Introduction 49
5.2 ShortMAC Overview 51
5.2.1 ShortMAC Packet Authentication 52
5.2.2 ShortMAC Example 53
5.2.3 Fault Localization and Guaranteed θ 54
5.3 ShortMAC Details 55
5.3.1 ShortMAC Packet Format 55
5.3.2 Protocol Details 57
5.4 Security Analysis 59
5.5 Theoretical Results and Comparison 62
5.6 SSFNet-based Evaluation 65
5.7 Linux Prototype and Evaluation 71

5.8 Discussion and Limitations 75
5.9 Summary 78

6 TrueNet **81**
6.1 Introduction 81
6.2 Setting 84
6.3 Fundamental Challenges 85
6.4 Design Building Blocks 86
6.5 TrueNet Overview 88
6.6 TrueNet Setup 91
6.7 TrueNet 1-Hop Monitoring 93
6.7.1 Per-packet Monitoring 93
6.7.2 Aggregate Monitoring 95
6.7.3 Per-Packet vs. Aggregate Monitoring 97
6.8 TrueNet Trustworthy Broadcasting 97
6.9 TrueNet Fault Localization Analysis 99
6.9.1 Fault Localization Delay 99
6.9.2 Security analysis 100
6.9.3 Overhead Analysis 101
6.10 TrueNet Router Architecture 102
6.11 Implementation and Evaluation 104
6.11.1 Prototype and Computational Overhead 104
6.11.2 Storage Overhead Measurement 107
6.12 Discussion 108
6.12.1 Incremental Deployment 108
6.12.2 Interdomain Deployment 109
6.13 Summary 110

7 DynaFL **111**
7.1 Introduction 111

7.2 Setting 115

7.2.1 Problem Formulation 116

7.3 Challenges and Overview 117

7.3.1 High-Level Steps 117

7.3.2 The Fingerprinting Function $\mathcal{F}$ 118

7.3.3 Challenges in a Neighborhood-based fault localization 120

7.3.4 DynaFL Key Ideas 123

7.4 Recording Traffic Summaries 124

7.4.1 Storing Packets 124

7.4.2 Secure Function Disclosure 126

7.4.3 Sampling and Fingerprinting 128

7.5 Reporting Traffic Summaries 132

7.6 Detection 132

7.7 Security Analysis 134

7.8 Performance Evaluation 136

7.8.1 Storage Overhead 136

7.8.2 Key Management Overhead 137

7.8.3 Bandwidth Overhead 139

7.8.4 Detection Delay 139

7.9 Summary 140

8 Related Work **145**

8.1 Detecting the Presence of Data-Plane Attacks 145

8.2 Vulnerabilities of Existing Fault Localization Schemes 146

8.3 Applicability and Practicality 147

8.4 Trusted Computing for Network Security 148

9 Conclusion **149**

A Proofs for PAAI **161**

A.1 Proof of Theorem 7 161

A.2 Proof of Corollary 8 162

A.3 Proof of Corollary 9 163

A.4 Proof of Theorem 10 164

A.5 Proof of Corollary 11 166

B Proofs for ShortMAC **167**

B.1 Proof of Lemma 13 167

B.2 Proof of Lemma 14 169

B.3 Proof of Theorem 15 171

C Proof for DynaFL **173**

C.1 Proof of Property 2 for Sketch 173

Chapter 1

Introduction

1.1 What is Fault Localization?

Performance-sensitive services, such as cloud computing, and mission-critical networks, such as the military and ISP networks, require high assurance of network data delivery. However, *real-world* incidents [2, 7, 13, 41, 55, 87] and studies [14, 21, 71, 97] reveal the existence of compromised routers in ISP and enterprise networks, and demonstrate that current networks are surprisingly vulnerable to data-plane attacks. Also, in a 2010 worldwide security survey [1], 61% network operators ranked infrastructure outages due to misconfigured network equipment such as routers as the No. 2 security threat. A compromised router or a dishonest transit ISP can easily drop, delay, inject or modify packets on the forwarding path to mount Denial-of-Service, surveillance, man-in-the-middle attacks, etc.

Unfortunately, current networks do not provide any assurance of data delivery in the presence of misbehaving routers, and lack a reliable way to identify misbehaving routers that jeopardize packet delivery. For example, a malicious or misconfigured router can "correctly" respond to `ping` or `traceroute` probes while corrupting other packets, thus cloaking the attacks from `ping` or `traceroute`. Yet most recent network diagnosis protocols are not designed for adversarial environments and can be evaded by adversaries [48, 49, 99, 47]. Though secure end-to-end path monitoring [18, 36] and multi-path routing [33, 35, 54, 72, 90, 91, 95] can mitigate data-plane

attacks to some extent, they are proven to render poor performance guarantees [76, 97]: without knowing exactly which link is faulty, a source node may need to explore an *exponential* number of paths in the number of faulty links in the worst case. As illustrated in Figure 1.1, where the default route from S to D is path $(1', 2, 3', 4)$, end-to-end monitoring only indicates if the current *path* is faulty without localizing a specific faulty link (if any) of a compromised or misconfigured router on the path. In the worst case, S needs to explore 2^4 paths to find the path with no faulty links, i.e., path $(1, 2, 3, 4)$.

Data-plane *fault localization* serves as a promising remedy for securing data delivery. In a nutshell, a fault localization protocol monitors data forwarding at each hop and localizes abnormally high packet loss, injection, and/or forgery on a certain *link*. Fault localization provides the following vital benefits.

Intelligent path selection. The current Internet Protocol (IP) instantiates a best-effort service model without indicating *if*, *when*, or *where* a packet is lost or corrupted during the packet transmission. Though aiming to provide reliable packet transmission, TCP is an *end-to-end* protocol which only detects whether or not a packet is lost on an end-to-end path but not *exactly where* the packet is lost. In contrast, fault localization provides accurate feedback about link quality and forwarding behavior of transit routers in the path. In recently advocated edge-controlled or multi-path routing protocols [94, 98, 96, 79], edge routers or source nodes can utilize such information on the network status to make the optimal path selection and adapt to adverse network conditions for improved reliability and quality of service.

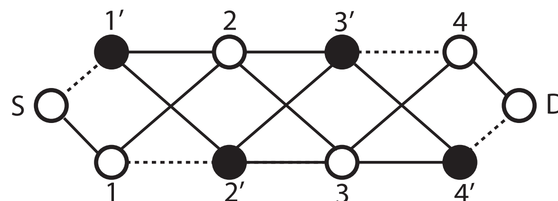


Figure 1.1: Exponential path exploration problem for end-to-end monitoring. Dotted links are faulty links of malicious routers (black nodes).

Accountability. Computer networks (such as the Internet and mesh networks) tend to represent a contractual business in which a node pays its peers or providers for forwarding its packets. Failing to provide information on the fate of transmitted packets by current Internet protocols prevents nodes from detecting failures of their peers or providers. Fault localization provides **forwarding accountability**, which refers to the ability to associate a certain forwarding behavior to a specific node, or to hold a specific node responsible for its activities. Forwarding accountability proves to be a *necessary* component for enforcing contractual obligations between participating nodes in a contractual networking service, as demonstrated by Laskowski and Chuang [56]. Intuitively, forwarding accountability can assure each node that its partners are indeed fulfilling the service agreement for packet forwarding.

Fast failure recovery. Fault localization enables a source node S to identify a faulty link of a malicious router M on which M drops, modifies, or injects packets during packet forwarding. By integrating the fault localization mechanism into edge-controlled routing, a source node can avoid using the identified faulty links when selecting routing paths, thus eliminating the exponential path exploration problem as shown in Figure 1.1. Assuming Ω faulty links in the network, a benign source node can identify and remove all faulty links and thus find a fault-free path after exploring at most Ω paths (linear in Ω) in the worst case. Figure 1.2 depicts the interaction between fault localization and routing for fast failure recovery.

Network diagnosis and performance measurement. Network diagnosis and performance measurement play an important role in ensuring normal network operations and performing informed traffic engineering. However, current practice and research studies in network diagnosis and performance measurement largely rely on ad hoc monitoring and probing, and assume *no* presence of malicious routers in the network [48, 49, 99, 47]. Secure fault localization provides information about link quality which cannot be biased by malicious routers and is thus *verifiable* to others even in the presence of adversaries.

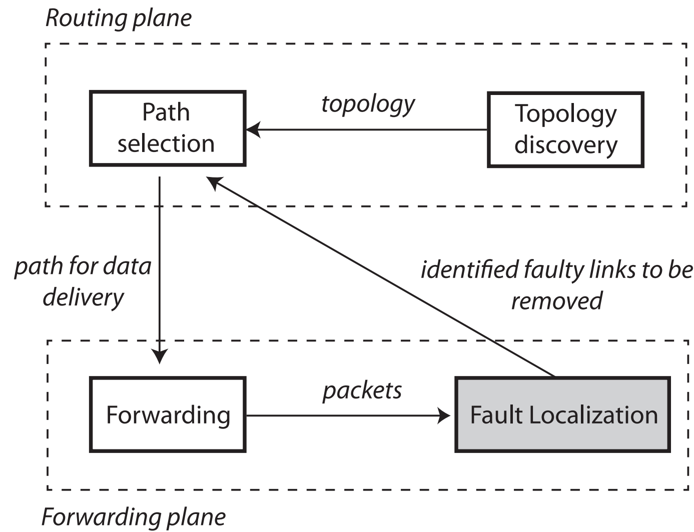


Figure 1.2: The network layer integrating data-plane fault localization for fast failure recovery (linear path exploration).

1.2 Challenges and Insights

In addition to providing security against strong adversaries, a fault localization scheme must also be *practical*; in particular, it must possess *all* of the following properties: (i) low detection delay (i.e., the time required to accurately localize a faulty link), (ii) low computational overhead, (iii) low communication overhead, and (iv) low storage overhead. Failing to achieve any of the above four properties may render the protocol impractical. For example, a fault localization protocol with high communication and/or storage overhead will perform poorly even when the network data plane is not under attack; this will be unacceptable in most settings, especially in resource-constrained networks such as sensor networks. Similarly, a fault localization protocol with a long detection delay will enforce only a poor bound on an adversary's ability to degrade end-to-end throughput before being identified. This may result in a significant monetary loss to a service provider and, worse, in cases where routing paths change periodically, the attacker may escape unscathed.

Until now, the design of fault localization protocols has proven to be surprisingly difficult when confronting *security*, *efficiency*, and *agility* challenges in the presence of strong adversaries.

- **Security and efficiency:** Sophisticated attacks such as framing and collusion attacks and natural packet loss tend to break fault localization protocols (e.g., Fatih [71], ODSBR [19], Watchers [25], AudIt [14], Network Confessional [15], etc) or lead to heavy-weight protocols (to prevent sophisticated attacks).

- **Agility:** In addition, current *secure* and *relatively* light-weight protocols leverage coarse-grained flow fingerprinting along end-to-end paths to prevent packet modification attacks while reducing communication overhead. However, in addition to having high storage overhead, these techniques result in long detection delays and require monitored paths to be long-lived (e.g., after monitoring 10^8 packets over the same path in *Statistical FL* by Barak et al. [21]), which is impractical for networks with short-lived flows and agile routing paths.

Our key insight is that we can achieve a high packet forwarding guarantee via fault localization by *limiting* the number of malicious packet drops/forgeries at the data plane, instead of perfectly detecting every single malicious activity which tends to result in high overhead. Therefore, strong per-packet monitoring or authentication to achieve perfect detection of every single dropped or forged packet is unnecessary for limiting the adversary's influence. Instead, the fault localization protocol can employ *probabilistic* approaches to yield statistical guarantees, e.g., via probabilistic packet monitoring using packet sampling or probabilistic packet authentication using a set of short packet-dependent random integrity bits. In this way, each dropped or forged packet has a *non-trivial* probability of detection. Hence, if a malicious node drops or forges more than a threshold number of packets, the malicious activity will cause a detectable deviation in the state maintained at different routers. Essentially, this methodology traps an attacker in a dilemma: if the attacker inflicts damage worse than a threshold, it will be detected, which may lead to removal from the network; if the attacker inflicts damage under a threshold, the damage is limited and thus a guarantee on data-plane packet delivery is achieved. To measure the effectiveness of confining data-plane attackers with fault localization, we propose a new metric called **guaranteed forwarding correctness**, which is the *lower bound* of the successful ratio of packet forwarding achievable along an end-to-end path, even in the presence of adversaries.

1.3 Dissertation Overview

Based on the philosophy of limiting the adversarial activities, we propose four protocols in this dissertation: PAAI, ShortMAC, TrueNet, and DynaFL, for secure and practical data-plane fault localization. PAAI, ShortMAC, and DynaFL are probabilistic protocols. More specifically, PAAI utilizes a secure packet probabilistic sampling technique, ShortMAC features a probabilistic packet authentication mechanism, and DynaFL employs a probabilistic packet fingerprinting data structure. In contrast, TrueNet is a deterministic protocol leveraging trusted computing technologies with special hardware support (such as TPM chips). From another perspective, both PAAI and ShortMAC are *path-based*, where the fault localization procedure is executed on the granularity of end-to-end paths and the source node of a path needs to directly interact with each router in that path. In contrast, TrueNet and DynaFL are 1-hop-based, as the operations required by fault localization are only performed between 1-hop neighbors. Figure 1.3 summarizes the characteristics of the four protocols.

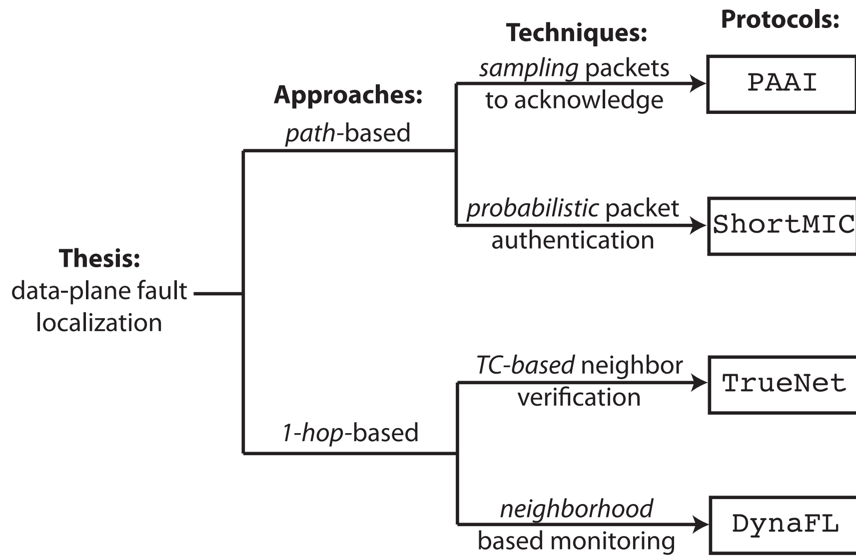


Figure 1.3: Summary of the proposed protocols in this dissertation.

These four protocols explore different approaches and directions in the design space of fault localization, and achieve various tradeoffs between storage overhead, communication overhead, computational overhead, and deployability. We summarize the tradeoffs of the proposed protocols

using these performance metrics in Table 1.1 to provide some intuition (we will provide a more detailed comparison including the effectiveness of fault localization in Chapter 9). Our results in this dissertation show that by limiting the adversarial activities with probabilistic algorithms or emerging hardware technologies, secure fault localization can be achieved with a lower bound on the forwarding performance and fundamentally higher efficiency than previously known protocols for fault localization. Our proposed fault localization protocols also address the security threats that defy most prior work.

Protocol	Storage	Communication	Computation	Deployability
PAAI	per-path state	$\approx 3\%$	per-packet PRF	loose time sync
ShortMAC	per-path state	$< 0.1\%$	per-packet MAC	change packet header
TrueNet	per-neighbor state	$< 0.1\%$	per-packet MAC	change packet header require TPMs
DynaFL	per-neighbor state	$< 0.1\%$	per-packet hash	loose time sync

Table 1.1: Metrics and tradeoffs.

The remainder of this dissertation includes the following chapters. Chapter 2 formally defines the problem and states the assumptions. Chapter 3 sketches the challenges in achieving secure fault localization by presenting several strawman approaches and their security vulnerabilities.

Chapters 4 and 5 present the two path-based protocols, PAAI and ShortMAC, respectively. Both protocols (and path-based fault localization protocols in general) require a source node S to solicit *acknowledgments* from intermediate routers in the forwarding path for the packets S has sent. PAAI explores schemes where a single acknowledgment returned by a router only acknowledges a *single* packet that S has sent, and focuses on studying how to employ secure sampling to reduce protocol overhead: whether and how to sample a subset of packets to acknowledge, or whether and how to sample a subset of routers to send the acknowledgments. In contrast, ShortMAC studies a different approach, using a single acknowledgment to acknowledge a set of aggregated packets for higher efficiency.

To overcome certain limitations of path-based fault localization protocols (e.g., poor support for dynamic routing paths), Chapters 6 and 7 present two 1-hop-based protocols, TrueNet and DynaFL, respectively. TrueNet assumes the deployment of trusted computing components in the network, and thus achieves secure fault localization with high efficiency unachievable in traditional

networks. Due to the inherent limitation of current trusted computing technologies, TrueNet is most effective against software-based (as opposed to hardware-based) data-plane attacks, which we argue is the major form of large-scale data-plane attacks. DynaFL implements secure 1-hop-based fault localization without relying on trusted computing, and thus is resilient against hardware-based attacks as well. DynaFL aims to localize forwarding faults to a specific 1-hop neighborhood instead of a specific link, trading precision for practicality of fault localization.

Finally, Chapter 8 summarizes the related work and Chapter 9 concludes the dissertation.

Chapter 2

Thesis, Problem Statements, Metrics, and Assumptions

2.1 General Thesis

This dissertation aims to achieve secure and efficient data-plane fault localization and explore the tradeoffs in this design space. More specifically, given a set of adversarial nodes in the network, we are interested in the design of protocols that monitor the forwarding behavior of intermediate nodes for packet dropping, modification, injection, and delaying activities over a period of time and then securely localize the presence of the adversary on a particular link (or a set of links). Note that the literature has showed that such protocols can only identify *links* adjacent to malicious nodes, rather than identifying the nodes [21]. Our thesis statement is as follows:

Thesis statement. *Instead of aiming to detect any single forwarding failure, we explore if fault localization can be utilized to limit the damage an adversary can inflict at the data plane and in turn produce a provable lower bound on the forwarding correctness. We also attempt to see if the philosophy of limiting the adversarial activities can enable the use of probabilistic algorithms and emerging hardware virtualization technologies, which may give rise to negligible protocol overhead without sacrificing security.*

To support this thesis statement, we take the following steps in this dissertation. We classify the fault localization protocols into *path-based* and *1-hop-based*. In a path-based approach, the fault localization process is operating on individual end-to-end paths, where the source node of a path requires provable "receipts"(or acknowledgments) from the destination and the intermediate nodes on the forwarding path for the packets that the source node has sent. We study probabilistic algorithms exploring different design dimensions to reduce the protocol overhead:

- intermediate nodes only send packet receipts for a probabilistically selected subset of packets (PAAI);
- only a probabilistically selected subset of packets send packet receipts to the source (PAAI);
- instead of acknowledging a single packet, a packet receipt can acknowledge a set of packets aggregated in a probabilistic and efficient way (ShortMAC).

In a 1-hop-based approach, the fault localization process is running between 1-hop neighbors, i.e., each node only monitors its 1-hop neighbors to detect any forwarding fault. As we show later, compared to path-based approaches, 1-hop-based fault localization can better cope with dynamic routing paths and traffic patterns, but tends to localize data-plane faults to a 1-hop neighborhood instead of a specific link (DynaFL). We also show that, with the aid of trusted computing, 1-hop-based fault localization can localize data-plane faults to a specific link with much lower overhead (TrueNet).

2.2 Scope and Assumptions

Scope. Since we focus on data-plane security at the network layer, we assume the following network control-plane and link-layer mechanisms, each of which represents a separate line of research orthogonal to ours.

- We can borrow existing secure routing protocols [50, 42, 73, 96] by which nodes can learn the genuine network topology and the source can know the outgoing path.

- We assume secure neighbor identification so that a node upon receiving a packet knows which neighbor sent that packet, which can be achieved via link-layer authentication.

- In addition, when needed, a source node S can set up a shared secret key K_{si} with router f_i using a well-studied key exchange protocol, e.g., Diffie-Hellman as in Passport [60]. This symmetric key exchange happens very infrequently thus representing only a one-time cost. Barak et al. [21] prove that such a shared secret is *necessary* for any *secure* fault localization protocol via path monitoring.

We focus on achieving secure fault localization against malicious routers. We do not consider control-plane or routing attacks and endhost- or source-based attacks such as DoS, while TrueNet complements existing secure routing [26, 27, 37] or DoS prevention schemes [92].

Cryptography assumptions. For the sake of efficiency, we avoid using per-packet asymmetric cryptography due to its high per-packet computation and communication overhead. We assume that the nodes can perform symmetric key operations as well as compute a collision-resistant hash function and a keyed pseudo-random function PRF.

Network model. We consider a general multi-hop network model where *routers* relay packets between *sources* and *destinations*, such as the ISP, enterprise, and datacenter networks. We assume that the links in the network independently exhibit some natural packet loss due to congestion and/or transmission errors. Throughout the paper, we follow the notation as illustrated in Figure 2.1. We denote the routers in a path by $f_1, f_2, \ldots, f_{d-1}$, the destination by f_d, and the link between f_{i-1} and f_i by l_i. We call nodes closer to the destination *downstream* nodes, and nodes that are further away from the destination as *upstream* nodes.

Basic notation. We denote the round-trip time from a node f_i to f_d as r_i. Let $\mathrm{E}_K(\cdot)$ denote encryption using symmetric key K. Further, let $\mathrm{MAC}_K(m)$ denote a message m authenticated by key K using a message authentication code (MAC). For simplicity, in our description, we do not differentiate between the keys for encryption and MAC computation; although in practice, one would derive separate keys for encryption and MAC computation.

Figure 2.1: An example path and notation.

2.3 Attacker Model

We assume an adversary in complete control of an arbitrary number of intermediate nodes on a path, including knowledge of their secret keys. The adversary can eavesdrop and perform traffic analysis anywhere on the path. The adversary may drop, inject or alter packets on the links that are under its control. We allow the protocol parameters to be public; consequently, the adversary may try to bias the measurement results in order to evade detection or incriminate honest links. However, the adversary cannot control the natural packet loss rate on the links in the path, because this would constitute a physical-layer attack which can be dealt with through physical-layer protections.

The goal of an adversary who controls malicious routers is to sabotage data delivery at the forwarding path. Instead of considering an individual forwarding attack, we seek a general way of defining malicious forwarding behavior. We identify packet dropping and packet injection as the two fundamental data-plane threats, while other data-plane attacks can be reduced to these two threats as follows: (i) packet modification is equivalent to dropping the original packet and injecting a fabricated packet, (ii) packet replay can be regarded as packet injection, (iii) packet delay can be treated as dropping the original packet and later injecting it, and (iv) packet misrouting can be regarded as dropping packets along the original path and injecting them on the new path. A formal definition follows:

Definition 1. *An* $(x, y)-$**Malicious Router** *is a router that intentionally drops* up to *a fraction* x *of the legitimate data packets from a source* S *to a destination* f_d*, and injects* up to y *spurious packets to* f_d*, pretending that the packets originate from* S*. The* misbehavior space *of such a malicious router comprises (i) dropping packets, (ii) injecting packets on any of its adjacent links which we call* ***malicious links*** *(non-malicious links are called* ***benign links****), (iii) strategically claiming*

arbitrary local state (e.g., number of packets received) to its own advantage, or (iv) colluding with other malicious routers to perform the above attacks.

Such a strong attacker model is not merely born out of academic curiosity, but has been widely witnessed in practice. For example, outsider attackers have leveraged social engineering, phishing [7], exploration of router software vulnerabilities [2, 13], and compromising weak passwords [41] to compromise ISP and enterprise routers [87]. Also, in a 2010 worldwide security survey [1], 61% of network operators ranked infrastructure outages due to misconfigured routers, which also fall under our attacker model, as the No. 2 security threat.

Finally, we note that the protocols proposed in this dissertation heavily depend on several cryptographic primitives, such as the Message Authentication Code (MAC) and Pseudo-Random Function (PRF). Though different protocols may utilize different implementations or instantiations of these cryptographic primitives, we assume the MAC resists existential forgery under chosen-plaintext attacks, and the PRF with a randomly chosen key provides outputs that look unpredictable and cannot be distinguished from a truly random function (except with a negligible probability). In other words, we assume the adversary cannot break these cryptographic primitives (though different implementations of them may be resilient against different numbers of adversarial queries, assuming their high-level security properties suffices for this dissertation).

2.4 Problem Formulation

This dissertation focuses on providing data-plane fault localization for a lower-bound guarantee on data-plane packet delivery. In this section, we define detection thresholds, faulty links, and finally we formalize fault localization.

We introduce the detection thresholds to limit the adversarial activities at network data plane:

Definition 2. *Given a* ***drop detection threshold*** T_{dr} *(i.e., fraction of dropped packets) and an* ***injection detection threshold*** T_{in} *(i.e., number of injected packets), a link* l_i *is defined as* ***faulty*** *iff: (i) more than* T_{dr} *fraction of packets are dropped on* l_i *by* f_i*,* or *(ii) more than* T_{in} *packets are injected by* f_i*,* or *(iii) the* adjacent *router* f_i *or* f_{i+1} *makes* l_i *appear faulty over a period of time.*

When T_{dr} and T_{in} are carefully set based on the prior knowledge such that the natural packet loss and corruption are below T_{dr} and T_{in}, respectively, a faulty link must be a malicious link.

Definition 3. $(N, \delta)-$***Data-Plane Fault Localization*** *is achieved iff: given an end-to-end communication path p, after a* ***detection delay*** *of sending N packets, the source node S of path p can identify a specific faulty link along that path (if any) with false positive or negative rate less than* δ*.*

Definition 4. $(\Omega, \theta)-$***Guaranteed Forwarding Correctness (Guaranteed Data-Plane Packet Delivery)*** *is achieved iff: after exploring at most* Ω *paths, a source can find a* ***non-faulty path*** *(*if any*) along which all routers have correctly forwarded at least* θ *fraction of the source's data packets sent along the path to* f_d*.*

To achieve a guaranteed θ, we need to *bound* (not necessarily *eliminate*) the adversary's ability to drop packets and to inject packets so that if the adversary drops more than α percent of packets or injects β bogus packets, it will be detected with a high probability. A formal definition follows.

Definition 5. *For an epoch with a sufficiently large number of data packets by a source, a fault localization protocol achieves* $(\alpha, \beta)_\delta-$***Forwarding Security*** *iff two conditions are simultaneously satisfied:*

1. *(*Low False Negative Rate*) When the adversary drops more than* α *percent of the data packets on a single link, or injects more than* β *fake packets on a single link, the source will detect at least one of the malicious links under the adversary's control with probability at least* $1 - \delta$*;*

2. *(*Low False Positive Rate*) The probability of falsely incriminating at least one benign link is at most* δ*.*

2.5 Metrics

The forwarding correctness (Definition 4) and forwarding security (Definition 5) provide a systematic way to quantify the effectiveness and security of fault localization. We also identify three key metrics to evaluate the efficiency or practicality of such protocols:

- *detection rate*, i.e., the number of data packet transmissions required to detect a malicious link (with the false positive and negative rates below a certain threshold),
- *communication overhead*, i.e., the additional packets (and their size) that are sent per data packet from the source, and
- *storage overhead*, i.e., the amount of temporary storage that must be maintained at each intermediate node per unit time.

Chapter 3

Security Challenges

A common approach to achieve data-plane fault localization is for the source node to require acknowledgment packets (ACK) from the destination and the intermediate nodes in the forwarding paths. In a realistic setting, a forwarding link may incur some benign packet loss due to congestion or channel errors. At the same time, an adversary who potentially controls multiple intermediate nodes may try to bias the identification procedure by selectively dropping, modifying, or injecting packets in order to evade detection or incriminate honest nodes. Consequently, a secure fault localization protocol must be simultaneously robust to both benign packet loss and malicious behavior. In other words, it must exhibit low false positive (falsely identifying a legitimate link as malicious) and false negative (falsely leaving a malicious link undetected) rates. This chapter presents several common security pitfalls or vulnerabilities in prior fault localization schemes to illustrate the challenges in achieving security against strong adversaries.

3.1 Challenge 1: Acknowledgment-based Approach

Let us consider that the source S in Figure 2.1 sends out a data packet m towards the destination f_d. Upon receiving m at each hop in the path, router f_i must return an acknowledgment (ACK) to S authenticated with the secret key shared with S (assuming S and f_i have pre-established a secret key using Diffie-Hellman [32] as in Passport [60], or some other key exchange protocol). If S receives correct ACKs from routers $f_1, \ldots, f_{i-1}$ but not from router f_i, S concludes link l_{i-1} is

faulty. In this approach however, a malicious router f_i can drop the ACK from another *remote* router, say f_{i+5}, without dropping other packets to frame l_{i+4} as malicious to the source (*framing attack*).

To reduce the overhead of ACK packets, the source node may "sample" a subset of packets and only the sampled packets will require ACKs from the routers. In this approach however, if a malicious router f_m can distinguish between sampled and non-sampled packets, f_m can safely drop *all and only* non-sampled packets without being detected.

Chapter 4 presents more sophisticated attacks against such acknowledgment-based fault localization protocols.

3.2 Challenge 2: Sophisticated Packet Modification Attacks

In Fatih [71], WATCHERS [25, 43], and AudIt [14], each router records a *traffic summary* based on counters or Bloom Filters [24], which are updated with *no* secret keys for the packets the router forwards. The routers periodically exchange local summaries with others for fault detection based on *flow reservation*. Without any authentication of the data packets, these schemes suffer from packet modification attacks. For example in AudIt [14], each router simply counts the number of packets it received for a certain path, and periodically sends the counter to the source node of the path for packet loss detection. However, malicious packet modification cannot be detected based solely on packet counts. Even when Bloom Filters are used [71] to reflect the packet contents, a malicious router can still tactically modify packets without affecting the Bloom Filter image (since Bloom Filters may not be collision-resistant).

Chapter 7 describes more challenges in dealing with packet modification attacks in fault localization protocols relying on flow conservation.

3.3 Challenge 3: Colluding attacks

Routers in a path may employ "hop-by-hop" monitoring to detect packet delivery fault to reduce the communication overhead of sending the traffic summaries back to the source. For example in Figure 2.1, each router f_i asks for the traffic summaries (e.g., acknowledgments) *only from* the

2-hop neighbor f_{i+2} in the path, and accuses l_i if f_i does not receive the correct traffic summaries. In this approach however, if f_i is colluding with f_{i+1} and does not accuse f_{i+1} even if f_i does not receive the correct traffic summaries from f_{i+2}, then f_{i+1} can safely drop packets without being detected. Watchdog [66], Catch [65], and the proposal due to Liu et al. [58] are vulnerable to similar colluding attacks.

Chapter 4

PAAI

In this chapter, we present our first path-based fault localization protocol, PAAI (including PAAI-1 and PAAI-2), that is robust against strong adversaries with a practical tradeoff between detection delay, communication overhead, and storage overhead. As a path-based protocol, in PAAI, the source node requires packet receipts, or acknowledgments (ACKs), for the packets the source has sent.

Instead of sending an ACK for each received packet by each router, PAAI employs secure sampling to reduce the communication overhead incurred by the ACKs. We *systematically* explore the design space of utilizing secure sampling for path-based fault localization protocols. We investigate a set of basic protocols, each exemplifying a design dimension and examine the underlying tradeoffs. In particular, PAAI-1 and PAAI-2 sample along different dimenions: PAAI-1 investigates how to sample a *subset of packets* to be acknowledged by all the routers in the forwarding path, and PAAI-2 investigates how to sample a *subset of routers* to send the ACKs for each packet. We also show the possibility of constructing hybrid protocols based on PAAI-1 and PAAI-2.

To clearly demonstrate the tradeoff between the two sampling approaches, in PAAI, a single ACK sent by a router only acknowledges one single packet sent by the source, while we may extend the protocols to enable one ACK to acknowledge a set of packets, just like ShortMAC shown in Chapter 5. For the ease of understanding, the PAAI protocol described in this chapter only focuses on detecting packet dropping and modification attacks. For PAAI-1 and PAAI-2,

we present both upper and lower performance bounds via theoretical analysis, and average-case results via simulations. We conclude that the proposed PAAI-1 protocol outperforms other related schemes.

4.1 Introduction

We observe that designing *any* path-based fault localization protocols using ACKs involves making two fundamental decisions:

1. which data packets to acknowledge; and
2. which intermediate nodes should send the ACKs.

With this in mind, we explore different design choices along the two aforementioned aspects and investigate the tradeoff using the performance metrics. More specifically, we study the following approaches:

1. A strawman approach: *Every* intermediate node sends an ACK for *every* lost or modified data packet.
2. The Probabilistic ACK-based Adversary Identification (PAAI) approaches: either (i) only a subset of data packets must be acknowledged (PAAI-1), or (ii) only a subset of intermediate nodes must respond to an ACK request (PAAI-2).

The full-ACK scheme achieves the lowest detection delay by determining the link for *every single* packet transmission failure. However, gathering such *fine-grained* information introduces high communication overhead. In contrast, PAAI-1 and PAAI-2 employ probabilistic sampling to gather only *coarse-grained* information, differing from each other in that they perform probabilistic sampling in different dimensions. In both PAAI schemes, we aim to achieve a low detection delay while retaining practicality for most networks.

The PAAI-1 protocol is fairly intuitive, simple and flexible, yet achieves more desirable properties than the full-ACK scheme, PAAI-2, and other related work. Finally, we also discuss the viability of constructing protocols that exemplify hybrids of the basic design primitives (Section 4.11).

Contribution. To the best of our knowledge, this is the first attempt to design a secure fault localization protocol that obtains a practical trade-off between the detection rate and the communication and storage overhead for realistic network settings. It is also the first systematic study of the design space for fault localization protocols (Section 4.3, Section 4.4 and Section 4.5). We propose a set of basic fault localization protocols, one for each design dimension, where the PAAI-1 protocol (Section 4.5.1) is distinctly more practical than the others. We obtain theoretical bounds for the performance of our protocols (Section 4.8), and also launch simulations to derive average-case results and validate our theoretical results (Section 4.9).

4.2 Setting

Besides the problem formulation described in Chapter 2, we introduce additional assumptions and conventions for this chapter below.

ACK structure. For any data packet m sent out by S, let the hash of m, denoted by $H[m]$, be a packet identifier for m. For any m, we define the corresponding ACK from f_i to have the structure $a_i = \langle H[m]||\mathcal{A}_i^m\rangle$, where $\mathcal{A}_i^m$ is a report computed by f_i. The report $\mathcal{A}_i^m$ will be a function of f_i's *local* report $\mathcal{R}_i$ and its downstream neighbor f_{i+1}'s ACK (if present). Specific details may vary in each protocol description.

Onion reports. We recall the well-known notion of an *onion report*. When each intermediate node f_i must return a local report $\mathcal{R}_i$ to S in an authenticated manner, then we have inductively, for $i \in [1, d-1]$, $\mathcal{A}_i = \mathrm{MAC}_{K_i}(i||\mathcal{R}_i||\mathcal{A}_{i+1})$, while $\mathcal{A}_d = \mathrm{MAC}_{K_d}(d||\mathcal{R}_d)$.

Assumptions. We assume the presence of *symmetric* paths, where the forward path (for data) and reverse paths (for acknowledgments) are identical; and we assume that a source node knows its forwarding path to the destination. We assume that the nodes on any given path are loosely time-synchronized.

Finally, given a path from a source to a destination, recall from Section 2.2 that the source can establish a shared pairwise symmetric key K_{si} with each intermediate node f_i on the path to the

destination.

Throughout this chapter, we focus on the localization of packet dropping and modification attacks, and use the general term *packet corruption* to denote packet dropping and modification activities.

4.3 A Strawman Approach: Full-ack

We observe that designing *any* path-based fault localization protocol involves making two fundamental decisions: (i) which data packets to acknowledge, and (ii) which intermediate nodes should send the acknowledgments. As a first step towards a systematic exploration of the protocol design space for fault localization protocols along the two aforementioned aspects, we discuss the simple and fairly intuitive 'full-ack' protocol (similar to the Optimistic Per-Packet FL Protocol from Barak et al. [21]), where *every* intermediate node on the forwarding path must return an ACK for *every* corrupted data packet sent by the source. A corrupted packet refers to one that fails to reach the destination intact (either dropped or modified). Below, we give a brief description of the protocol and discuss its security and performance. A theoretical analysis and simulation results for the full-ACK protocol are given later in Section 4.8 and Section 4.9 respectively.

Protocol. Let us consider that S sends out a data packet m towards the destination f_d. On receiving m, f_d must return an ack, a_d, authenticated with the secret key shared with S, i.e., $a_d = \text{MAC}_{K_{sd}}(H[m])$. If no valid ACK is received from f_d within a pre-specified wait-time, S will send out an onion report request. The onion report is computed by the intermediate nodes in the manner explained earlier, wherein a local report $\mathcal{R}_i$ is set to be $\langle i||H[m]||a_d\rangle$. Upon receiving the ACK containing the onion report from f_1, S can sequentially verify each report embedded in it. For some $i < d$, if the MAC from each intermediate node $f_j, j \in [1, i]$ is valid but the MAC from f_{i+1} is invalid or not present in the final ack, then S identifies link l_i as faulty and adds one to its *corruption score*. Over a period of time, if the corruption score of a particular link exceeds a fixed threshold determined from the natural packet loss rate then that link is identified as malicious.

Security. In the above protocol, if a malicious node corrupts a packet (data or ack), one of its adjacent links has its corruption score increased. This follows largely from the security of onion reports. Since PAAI-1 employs similar techniques, we defer more details to Section 4.4. The adversarial nodes on the path may collude to share packet corruption activities among themselves; however, in this case, the corruption rate will still be bounded (proportional to the number of malicious nodes in the path).

Performance. For *each* corrupted packet, the full-ACK scheme can determine precisely the location of the packet corruption, thus it is able to directly compute the corruption rate of each link on a given path and identify malicious links within a small number of packet corruption. However, this high detection rate is achieved at the price of a *large* amount of communication overhead at each node. Specifically, the full-ACK scheme imposes an overhead of at least one packet of O(1)-size per data packet sent out by S; and an additional overhead of one packet of O(d)-size (the onion report) in case packet corruption occurs. The storage overhead is high in the worst case but lower on average due to the low detection delay. More details are given later in Section 4.8 and Section 4.9.

The high overhead makes the full-ACK protocol unaffordable for most networks; therefore, fault localization protocols with a better trade off amongst the three performance metrics are desirable.

4.4 Overview of PAAI

In contrast to the full-ACK protocol, where the ACK mechanism was completely deterministic, we now investigate *probabilistic ack-based adversary identification* (PAAI) approaches with the underlying motive of reducing the protocol overhead at the expense of slightly worsening the detection delay. Loosely speaking, we investigate two contrasting approaches: one where only a subset of data packets must be acknowledged, and another where only a subset of intermediate nodes must send the acknowledgments[1]. In particular, we construct, (i) the PAAI-1 protocol: every intermediate node sends an ACK for only a selected fraction of data packets; and (ii) the PAAI-2 protocol: only one selected intermediate node sends an ACK for each data packet. At first glance, the two approaches may seem to be only minor variations of the full-ack mechanism; however, we stress

[1] It is natural to imagine the possibility of composing these approaches. We discuss this in Section 4.11.

that there are several challenges involved in ensuring security of these approaches. We now briefly outline these approaches along with the challenges involved.

In the first approach (PAAI-1), S monitors the path for only a fraction of the total traffic. More specifically, for a given data packet, S solicits an ACK from every intermediate node *only with some probability* p. Now, since a fraction of traffic is *unmonitored*, the protocol must ensure that a malicious node f_z is not able to determine from the content of a data packet m whether S solicits an ACK for m. Otherwise, on receiving m, if f_z determines that m need not be acknowledged, then it could safely corrupt m without increasing its probability of being identified.

In PAAI-2, S monitors the path for every data packet, with the provision that S solicits an ACK for a *corrupted* data packet from only one *selected* node on the path. However, the protocol must ensure that a malicious node f_z cannot decipher the identity of the selected node f_e from the content of a data packet m. Otherwise, on receiving m, if f_z determines that $f_e \leq f_z$ (i.e., whether f_e is upstream to or equal to f_z), then it could safely corrupt m without increasing its probability of being identified.

In order to circumvent the above attacks and still perform probabilistic monitoring, we make use of a *delayed sampling* mechanism. Specifically, in both PAAI protocols, S sends out an ack request (henceforth referred to as a *probe*) at a later time for a data packet sent earlier. In PAAI-1, the probe conveys the information that the corresponding data packet must be acknowledged (otherwise no probe is sent). In PAAI-2, the probe content determines which intermediate node is selected. However, in either protocol, a malicious node may withhold a data packet until the arrival of the corresponding probe in an attempt to decide whether to corrupt m. To circumvent this, we require loose time-synchronization among the nodes in the network such that the clock error between two adjacent nodes f_i and f_{i+1} is less than $\min(r_0)$, i.e., the minimum value of the round trip time from S to the destination. In this scenario, an intermediate node would discard a data packet if it carries an expired timestamp.

Both PAAI protocols employ a scoring mechanism in order to identify malicious links over a period of time. We set a threshold for the end-to-end corruption rate of data packets for a given path. The threshold value is chosen based on the natural packet loss rate, such that the natural end-to-end loss rate will not exceed the threshold value. At the end of each probe, S computes

the end-to-end corruption rate so far, based on the number of sent data packets and successfully received ACKs from the destination; if the corruption rate exceeds the threshold value, then it indicates that an adversary is present on the path. Using the history of scores (i.e., the scores accumulated so far) of the links, S will identify the adversarial presence on a link (or a set of links) whose score exceeds a per-link score threshold within a bounded number of probes. On the other hand, the score of an honest link will not exceed the per-link score threshold. Note that this mechanism is in sharp contrast to the on-demand secure routing approach [19] where the probing is launched only when the end-to-end drop exceeds a certain threshold; consequently there is no history of scores which can be used, thus allowing an adversary to freely corrupt packets until the end-to-end corruption rate reaches the threshold and then cause arbitrary links to be incriminated due to natural packet loss when probing is initiated.

We now give some details on the specific scoring mechanism employed by each PAAI protocol. Loosely speaking, in PAAI-1, if an intermediate node fails to return an ACK for a probed data packet, then S will increase the corruption score of its upstream link. However, note that if each intermediate node were to send a separate ack, then a malicious node could selectively drop the ACKs from legitimate nodes in order to incriminate honest links. To circumvent this, PAAI-1 employs the use of onion reports similar to the full-ACK protocol.

PAAI-2, on the other hand, utilizes a slightly different scoring mechanism. For a given data packet, if the selected node f_e fails to return an ack, then S infers that there exists at least one malicious link upstream of f_e; consequently S will increase the corruption score of *each* link between f_e and itself. Now, suppose that a malicious packet corruption occurred at a link l_{i-1}. Then, let X be the event that the intermediate node f_i is selected. We ensure that event X occurs with a *fixed probability*. Due to the above scoring mechanism, each occurrence of X will create a difference in the scores of the links on either side of f_i. Over a period of time, a difference in the score of two adjacent links would indicate a potential malicious link. In order to ensure that event X occurs with a fixed probability, PAAI-2 selects an intermediate node *uniformly at random* for any data packet. The protocol must also ensure that the identity of the selected node for any data packet is not revealed at *any point in time*; otherwise, a malicious node could selectively drop ACKs from legitimate nodes in order to incriminate honest links. Specifically, in order to incriminate an honest

link l_h, a malicious node could drop the ACK every time f_{h+1} is selected, while behaving honestly every time f_h is selected. This would create a difference between the scores of l_{h-1} and l_h. In order to circumvent this, we design an *oblivious* selection and acknowledgment procedure, such that the identity of the selected node is hidden to each node (except S) *even through traffic analysis*.

Finally, we remark that an adversary may choose to modify or drop any of the following: (i) data packet, (ii) probe, or (iii) ack. However, our protocol design ensures that the source node S interprets each such activity simply as a data packet drop. In what follows, we will simply use the term *drop* to refer to any kind of packet modification or drop. Looking ahead, in Section 4.8, we show that an adversary achieves the same total end-to-end corruption rate by employing different individual corruption rates for different packet types.

4.5 The PAAI Protocols

Formally, the two PAAI protocols PAAI-1 and PAAI-2 consist of five stages: (i) *send data and decide whether to probe*, (ii) *probe*, (iii) *acknowledge*, (iv) *score*, and (v) *identify*. We give the details of both PAAI-1 and PAAI-2 below.

4.5.1 PAAI-1

PAAI-1 employs probabilistic sampling in order to determine which data packets must be acknowledged. For every sampled data packet, PAAI-1 requires each intermediate node and the destination to return an onion report. The protocol details follow.

Stage 1: send data and decide whether to probe

Consider that S sends out a data packet $m = \langle \text{data} || \text{timestamp} \rangle$ towards the destination. On receiving m, an intermediate node f_i first checks whether the embedded timestamp is recent. If verification fails, then m is dropped. Otherwise, f_i stores the identifier $H[m]$ for m and starts a wait timer $t_i = r_0/2$. Finally, m is forwarded toward the destination.

S then uses a secure sampling (SS) algorithm to determine whether it must send out a probe for m. When given any input m, the SS algorithm must output “Yes” with a fixed probability p,

where p is the probe frequency fixed at setup time. Such an algorithm can easily be constructed by making use of a PRF keyed with a secret key known only to S. Note that such a mechanism is necessary to prevent an adversary from correctly predicting whether or not a specific data packet is sampled.

If the SS algorithm outputs "No", then the protocol is terminated for the current round. Otherwise, S executes the next stage of the protocol. In the following, it is implicit that a node f_i accepts a packet (probe or ack) iff it contains a data packet identifier already stored at f_i.

Stage 2: probe

S sends out a probe $c = H[m]$ towards the destination. The probe contains the identifier $H[m]$ for the data packet m sent earlier. On receiving a probe, an intermediate node f_i starts a wait-timer $t_i = r_i$, forwards the probe towards the destination, and moves to the next stage. Note that, in practice, the probe frequency p will be set to a very low value. Therefore, if we use unauthenticated probes, an adversary could potentially waste a lot of communication power of the intermediate nodes by sending bogus probes. As a countermeasure, one could use authenticated probe packets, where a chain of MACs (one for each intermediate node) is attached to each probe.

Stage 3: acknowledge

In this stage, the destination f_d and intermediate nodes must return an onion report to S. Ideally, the onion report must either originate at the destination, or at the upstream node of the link where m_j was dropped. To this end, we employ the following rules: (i) If no downstream ACK is received within the wait time t_i, f_i originates an onion report $\mathcal{A}_i = \text{MAC}_{K_{si}}(i||H[m])$. (ii) Otherwise, on receiving a downstream ACK within the wait-time, f_i sets the local report $\mathcal{R}_i$ to be $\langle i||H[m]\rangle$ to create an onion report $\mathcal{A}_i$ as explained earlier in Section 4.2. Finally, f_i sends out an ACK $a_i = \langle H[m]||\mathcal{A}_i\rangle$ towards S.

Stage 4: score

Upon receiving the ACK containing the onion report from f_1, S can sequentially verify each report embedded in it. For some $i < d$, if the MAC from each intermediate node $f_j, j \in [1, i]$ is valid

but the MAC from f_{i+1} is invalid or not present in the final ack, then S identifies link l_i as faulty and adds one to its *corruption score*. In the case where S does not receive any report within a wait-time, S can simply conclude that a packet corruption occurred at its downstream link l_0.

Stage 5: identify

At any point in time, let s_i be the corruption score of link l_i, and n be the total number of probes evoked by S so far. The average packet corruption rate ρ_i^* for link l_i so far can be computed as $\frac{s_i}{n}$. We set a *per-link corruption rate threshold* (denoted by T_{dr}) according to the natural loss rate ρ_i ($T_{dr} > \rho_i$). Then if $\rho_i^* > T_{dr}$, S convicts l_i as a malicious link. More details are given in Section 4.8.

4.6 PAAI-2

Now we turn to the other design alternative: probabilistically sampling a subset of intermediate nodes which must return an ack. We propose PAAI-2 where only *one* intermediate node is selected to return a report for every data packet. We remark that the strategy of selecting a subset of intermediate nodes which must return an ACK tends to be vulnerable to selective dropping attacks (see Section 4.4). Consequently, we find that PAAI-2 requires more algorithmic complexity but achieves a higher detection delay than PAAI-1.

Stage 1: send data and decide whether to probe

Consider that S sends out a data packet $m = \langle \text{data}||\text{timestamp} \rangle$ towards f_d. On receiving m, an intermediate node (including f_d) first checks whether the embedded timestamp is recent. If verification fails, then m is dropped. Otherwise, f_i stores the identifier $H[m]$ for m and starts a wait timer $t_i = r_i$. Finally, m is forwarded toward the destination.

On receiving m, f_d creates a report $\mathcal{A}_i = \text{MAC}_{K_{sd}}(H[m])$ and returns an ACK $a_d = \langle H[m]||\mathcal{A}_i \rangle$ to S. On receiving an ACK from f_d within the wait-time, an intermediate node f_i stores a copy of it, forwards it towards S, and starts a waiting time $t_i = r_0 - r_i$.

If S receives a valid ACK from f_d within a waiting time, it concludes that m arrived unaltered at f_d and the protocol is terminated for the current round. Otherwise, S executes the next stage

of the protocol. In the following, it is implicit that a node f_i accepts a packet (probe or ack) iff it contains a data packet identifier already stored at f_i.

Stage 2: probe

S sends out a probe $c = \langle H[m] || \mathcal{Z} \rangle$ towards f_d. The probe contains an identifier $H[m]$ for m, and a random challenge $\mathcal{Z}$.

On receiving a probe within the wait-time, an intermediate node f_i computes a $\mathrm{PRF}_{K_i}(\cdot)$-based predicate T_i over input $\mathcal{Z}$, where T_i returns "true" with probability $\frac{1}{d-i+1}$. If the wait-timer expires, then the state maintained for m is deleted. In what follows, we say that a node f_i is *sampled* for a data packet m if T_i returns true on input $\mathcal{R}$.

Finally, f_i starts a wait-timer $t_i = r_i$ and forwards the probe towards f_d.

Stage 3: acknowledge

In this stage, the intermediate nodes must return an ACK to S. Ideally, the ACK must originate at the upstream node of the link where m was corrupted. To this end, we employ the following rules: (i) If an intermediate node f_i does not receive any ACK from its downstream neighbor within the wait-time t_i, it generates an encrypted report $\mathcal{A}_i = \mathrm{E}_{K_{si}}(\mathrm{MAC}_{K_{si}}(i||c||a_d))$. If no ACK was received from f_d in stage 1, then a_d is set to $\perp$. (ii) Otherwise, on receiving a downstream ACK within the wait-time, f_i performs one of the following actions. If f_i was *sampled* for m during stage 1, it generates an encrypted report $\mathcal{A}_i$ (as described in previous case) to overwrite the report in the received ack. Otherwise it re-encrypts the report in the received ack, i.e., $\mathcal{A}_i = \mathrm{E}_{K_{si}}(\mathcal{A}_{i+1})$. The security reason for the re-encryption is given in Section 4.7. Finally, f_i sends out an ack $a_i = \langle H[m] || \mathcal{A}_i \rangle$ towards S.

Definition 6. *We say that a node f_e is* ***selected*** *for a data packet m, if (i) f_e is sampled for m, and (ii) $f_1, \ldots, f_{e-1}$ are not sampled .*

From the above definition, it follows that, for a given data packet, only one intermediate node is *selected* uniformly at random with probability $\frac{1}{d}$. Observe that due to the ACK forwarding mechanism described above, S expects an ACK that was generated at the *selected* node f_e and re-encrypted by *each* upstream node between f_e and S.

Stage 4: score

In this stage, S assigns numerical *scores* to the links. On receiving an ACK from f_1, S first decodes the embedded report $\mathcal{A}_1^m$ by performing successive decryption using the keys $K_{s1}, \ldots, K_{se}$ *in that order*, where K_{se} is the secret key shared between S and f_e. If the final decoded value matches the expected value $\langle \mathrm{MAC}_{K_{se}}(e||c) \rangle$, then S decides that there was no malicious activity in the interval $[l_0, l_{e-1}]$; consequently, no scores are updated. Otherwise, S is convinced that *there exists at least one malicious link* in the interval $[l_0, l_{e-1}]$. Since each link in this interval has equal probability of being malicious, S adds 1 to the individual score of each link in the interval. No scores are updated for the links in the interval $[l_e, l_{d-1}]$.

Stage 5: identify

S pre-determines a per-link corruption rate threshold T_{dr}, based on which it further sets a threshold ψ_{th} for the end-to-end corruption rate of data packets. S constantly monitors the actual end-to-end data packet corruption rate ψ based on the number of sent data packets and successfully received ACKs from the destination. It is guaranteed that $\psi_{th} < \psi$ ***iff*** there is at least one link with a corruption rate exceeding T_{dr}. Then the source can compute per-link corruption rate based on the accumulated data and identify the link with the excessive corruption rate. More details are given in Section 4.8.

4.7 Security Properties

In order to prove our theoretical results in section, we require the PAAI protocols to exhibit some key security properties. Below, we discuss four important security properties of the PAAI protocols.

Delayed Sampling. Recall that in PAAI-1, for a given data packet, S solicits an ACK only with some probability p. We note that a malicious node f_m should not be able to decipher from the content of a data packet m whether S solicits an ACK for m. Otherwise, on receiving m, if f_m determines that m need not be acknowledged, it could safely corrupt m without increasing its probability of being identified. Now recall from Stage 3 of PAAI-2 that for a given data packet,

a *sampled* node must overwrite the ACK received from its downstream neighbor with a fresh ack. Hence, we note that on receiving a data packet m, if a malicious node could decipher from its content whether it is sampled for m, then it could safely corrupt m without increasing its probability of being identified.

To prevent the above attacks, in both PAAI protocols, a probe c is sent at a later time to request ACKs for a data packet sent at an earlier time. In PAAI-1, the probe conveys the information that the corresponding data packet must be acknowledged. In PAAI-2, the probe content determines whether an intermediate node is sampled. However, in both PAAI protocols, a malicious node may now try to wait for the arrival of the probe c before forwarding m, in an attempt to decide whether to drop m. Therefore, we require loose time-synchronization amongst the nodes in the network such that the clock error between two adjacent nodes f_i and f_{i+1} is less than $min(r_0)$, i.e. the minimum value of the round trip time from S to the destination. In this scenario, an intermediate node would discard a data packet that carries an expired timestamp.

Security against selective packet corruption. It is easy to observe that the use of onion report mechanism prevents any selective dropping attacks by an adversary in PAAI-1. Now recall that in PAAI-2, if an intermediate node does not receive any ACK within a wait-time, it generates a new ACK even if it is not sampled; otherwise an adversary could observe the ACK origin to infer whether an intermediate node is sampled. Further, the *re-encrypt or overwrite* technique in PAAI-2 ensures that a constant size ACK is forwarded at each hop. If this were not the case, then an adversary who eavesdrops at all the links on the path to observe any difference in the size of the ACK at various links can infer additional information about the origin of the ack. Furthermore in PAAI-2, for a given data packet, the probed node is *selected uniformly* at random; otherwise an adversarial node can simply preferentially perform data-plane attacks at nodes that are not as likely to be sampled as others.

Adversary localization. In PAAI-1, for each sampled data packet (i.e., the data packet for which an onion report is requested) that was corrupted, S can localize the location of the packet corruption to a specific link from the verification of the onion report. Now recall that in PAAI-2,

for a given data packet, the ACK expected by S is the one that is generated by the *selected* node. Therefore, if the *selected* node is located between S and the adversary, then the adversary cannot influence the final ack received at S. This implies that if no ACK or an invalid ACK is received at S, then there must exist at least one malicious link in the interval $[l_0, l_{e-1}]$.

4.8 Theoretical Analysis

In this section, we theoretically analyze the guaranteed end-to-end forwarding correctness, detection delay, communication and storage overhead of the proposed protocols. Proofs of the theorems and corollaries are given in the appendix. The results are summarized in Table 4.1, which also gives a clear comparison between the full-ack, PAAI, and statistical FL protocol [21]. We compare our PAAI protocols mainly with the statistical FL protocol because it is the state-of-the-art and the only protocol with a rigorous theoretical analysis to the best of our knowledge. In Section 4.9, we validate our theoretical results and present average-case results from simulations.

Definitions and notation. Let ρ_i be the natural packet loss rate of link l_i, and suppose that ρ_i's are i.i.d. random variables with maximum value ρ. Let T_{dr} denote the per-link corruption rate threshold; and ρ_i^* be the actual average corruption rate of link l_i, including both natural and malicious corruption. Let ζ be the malicious end-to-end corruption rate, i.e., the corruption rate due to malicious links. When the observed corruption rate value approaches its true value within a small uncertainty interval, the fault localization false positive/negative rate is limited below a certain threshold ϵ. We call this the ***converged*** condition.

Let p be the probe frequency employed in PAAI-1. Further, in PAAI-2, let ψ_{th} be the threshold of the end-to-end data packet corruption rate. Let η_i be the number of times that node f_i is *selected* so far.

4.8.1 Bounding Malicious End-to-End Corruption Rate

For ease of understanding, all the theoretical bounds in this subsection are computed under the converged condition. In Section 4.8.2 we derive the detection delay (number of data packets sent

by the source required to reach converged condition) for the full-ACK and PAAI schemes. We can see the detection rates are high in the full ACK and PAAI-1 schemes, so the "unconverged" time period is negligible.

For simplicity, we first assume that an adversary employs an identical corruption rate for all types of packets (data, probe or ACK packets) at a controlled link l_i, and thus the probability that a packet of any kind is corrupted at l_i is ρ_i^*. The following theorem proves the (Ω, θ)-guranteed forwarding correctness (Definition 4) and $(\alpha, \beta)_\delta$-forwarding security (Definition 5) of full-ACK, PAAI-1, and PAAI-2. Since PAAI does not consider packet injection attacks, β is inapplicable here. In addition, Ω equals to the number of malicious links in the network, which is explained in Section 1.1. The following theorem also provides a general bound on the damage that an adversary with an *arbitrary* number of links under its control can inflict to the network's *end-to-end* throughput.

Theorem 7. *Forwarding Security and Correctness:* *Given a path of length d, the fractions (α) of packets an adversary can drop on any link without being detected in full-ACK, PAAI-1, and PAAI-2 are: (i) $\alpha = T_{dr}$ in full-ACK and PAAI-1, and (ii) $\alpha = 1 - \frac{(1-T_{dr})^{2d}}{(1-\rho)^{2(d-1)}}$ in PAAI-2 by setting the end-to-end corruption rate threshold ψ_{th} as $\psi_{th} = 1-(1-T_{dr})^{2d}$, respectively. And the guaranteed forwarding correctness is $\theta = (1 - T_{dr})^d$, given the drop detection threshold T_{dr}.*

In general, an adversary in control of z intermediate links can cause (at most) the following malicious end-to-end corruption rates without being detected: (i) $\zeta = zT_{dr}$ in full-ACK and PAAI-1, and (ii) $\zeta = 1 - \frac{(1-T_{dr})^{2d}}{(1-\rho)^{2(d-z)}}$ in PAAI-2 by setting the end-to-end corruption rate threshold ψ_{th} as $\psi_{th} = 1 - (1 - T_{dr})^{2d}$.

It is possible that an adversary may choose to corrupt different types of packets at different rates. However we can intuitively see that the adversary cannot gain any advantage by doing this, because corrupting any type of packet will always result in an increase in the corruption score of the link where the packet was corrupted.

Corollary 8. *An adversary who employs different corruption rates for different types of packets achieves the same maximum end-to-end corruption rate.*

Corollary 9 presents the *optimal* strategy that an adversary can employ in order to cause

maximum degradation to the network throughput. The corresponding bounds on the degradation in network throughput under the optimal strategy are also presented.

Corollary 9. *Given a fixed number of malicious links, the malicious end-to-end corruption rate* ζ *increases approximately linearly with the increase of natural loss rate* ρ*. Given a fixed number* z *of malicious links, the optimal strategy for the adversary in order to cause the maximum end-to-end corruption rate across all the paths containing malicious links in the network is to deploy one malicious link for one path. In this case, the total malicious corruption rate across all paths containing compromised links increases linearly with* z*.*

4.8.2 Detection Delay

We compute the detection delays and prove the (N, δ)-data-plane fault localization (Definition 3) for the full-ACK scheme and PAAIs in the following theorem.

Theorem 10. (N, δ)- ***Data-Plane Fault Localization:*** *Given the threshold* $T_{dr} = \rho + \epsilon$ *and the allowed false positive rate* δ*, the full-ACK and the PAAI protocols require the following number of packets transmitted by the source to converge. (i)* $N_1 = \frac{ln(\frac{2}{\delta})}{8\epsilon^2 \cdot (1-\rho)^{2+d}}$ *for full-ACK scheme, (ii)* $N_2 = \frac{N_1}{p}$ *for PAAI-1, where* p *is the probe frequency, and (iii)* $N_3 = 2^d \frac{ln(\frac{2}{\delta})}{18\epsilon^2} \cdot d \cdot log(d)$ *for PAAI-2.*

Corollary 11 shows the sensitivity of the detection delay (achieved by the full-ACK and PAAI protocols) to the various protocol parameters. As it turns out, PAAI-1 can achieve shorter detection delays under various parameter settings (and thus, a wide range of empirical scenarios).

Corollary 11. *For both the full-ACK scheme and PAAI-1, the allowed false positive rate* δ *is the dominating factor on their detection delays, while the network-related parameters (natural packet loss rate* ρ *and path length* d*) have negligible influence on the detection delays. However, the detection delay of PAAI-2 heavily depends on the path length* d*.*

For example, if we set $\delta = 0.03$ and $p = \frac{1}{d^2}$, and choose an arbitrary network setting where $T_{dr} = 0.03$, $\rho = 0.01$ and $d = 6$, then we have $N_1 \doteq 1500$, $N_2 \doteq 5 \times 10^4$ and $N_3 \doteq 6 \times 10^5$; whereas the detection delay in statistical FL protocol [21] is 2×10^7. Per Corollary 11, the detection delay for PAAI-1 does not vary much given other network-related parameter settings. Table 4.1 compares the detection delays achieved by the different protocols.

Protocol	Detection Delay	Communication	Storage	
			worst	ideal
Full-ACK	$\frac{ln(\frac{2}{\delta})}{8\epsilon^2 \cdot (1-\rho)^{2+d}}$	$O(1+\psi d)$	$O(2r_0\nu)$	$O(r_0\nu)$
PAAI-1	$p\frac{ln(\frac{2}{\delta})}{8\epsilon^2 \cdot (1-\rho)^{2+d}}$	$O(pd)$	$O(r_0(0.5+p)\nu)$	$O(r_0(0.5+p)\nu)$
PAAI-2	$2^d\frac{ln(\frac{2}{\delta})}{18\epsilon^2} \cdot d \cdot log(d)$	$O(1)$	$O(2r_0\nu)$	$O(r_0\nu)$
Statistical FL [21]	$d^2\frac{ln\frac{d}{\delta}}{p\epsilon^2}$	$O(\frac{p\epsilon^2}{dln\frac{d}{\delta}})$	$O(pr_0\nu)$	$O(pr_0\nu)$
Combination 1	$p\frac{ln(\frac{2}{\delta})}{8\epsilon^2 \cdot (1-\rho)^{2+d}}$	$O(p(1+\psi d))$	$O(r_0(0.5+2p)\nu)$	$O(r_0(0.5+2p)\nu)$
Combination 2	$2^d\frac{ln(\frac{2}{\delta})}{18\epsilon^2 \times p} \cdot d \cdot log(d)$	$O(p)$	$O(r_0(1+p)\nu)$	$O(r_0\nu)$

Table 4.1: Detection rate and overhead comparison. The notation is given at the beginning of Section 4.8. We translate the related results [21] using our notation. Combination 1 and Combination 2 are described in Section 4.11.

4.8.3 Communication Overhead

In this section we compute and compare the communication overhead incurred by the full-ACK and the PAAI protocols for a given path of length d. The analysis results are presented in Table 4.1.

Full-ack. Recall from Section 4.3 that in the benign case where *no* packet corruption occurs, each data packet requires one O(1)-sized ACK from the destination. When a packet corruption happens, the source solicits a $O(d)$-sized onion report via a O(1)-sized probe packet. Therefore, given the end-to-end corruption rate ψ, the overall communication overhead per packet is $O(1 + d\psi)$.

PAAI-1. Recall from Section 4.5.1 that for each sampled data packet, the source solicits one $O(d)$-sized onion report (in case of authenticated probes, the size of a probe packet is also $O(d)$). Since a given data packet is sampled only with probability p, the amortized communication overhead per data packet is $O(pd)$. By setting $p = \frac{1}{d^2}$ we can get $O(\frac{1}{d})$ overall communication overhead per packet. Note that the above results apply *regardless of whether there are packet corruption activities or not.*

PAAI-2. Recall from Section 4.6 that each intermediate node f_i on the forwarding path either generates a new ACK or re-encrypts the ACK received from downstream. Therefore, an ACK packet

traversing the path has a constant size (O(1)) at any point in time. Further, PAAI-2 requires one O(1)-sized probe packet per data packet sent by the source. Note that the above results apply *regardless of whether there are packet corruption activities or not.*

4.8.4 Storage Overhead

Storage is a major concern in certain resource-constrained networks. An adversary may even exploit the storage limitation and manipulate packet corruption activities to intentionally create the worst case condition for the storage overhead of an fault localization protocol. On the other hand, in practical settings, including when the adversary has been identified (and bypassed), excessive packet corruption is infrequent (thus the worst cases do not arise frequently). A high storage overhead in such an *ideal* case is undesirable. Therefore, in this section we analyze and compare the storage overhead in both worst and ideal cases for the full-ACK scheme and PAAIs. In Section 4.9 we present the average-case storage overhead via simulations.

In the following, let ν be the number of data packets that S sends out per unit time. Recall that r_i denotes the round trip time between node f_i and f_d. The results given below are summarized in Table 4.1.

Full-ack. In the worst case, on receiving a data packet m, an intermediate node f_i needs to first wait r_0 time for a probe from the source, and r_i time for an ACK from f_{i+1}. Therefore f_i can at most store $O(2r_0\nu)$ packets at a time. In the ideal case without packet drop, f_i only needs to store a packet for r_i time before receiving an ACK from f_{i+1}.

PAAI-1. If a data packet m is not selected for a probe, f_i needs to wait $\frac{r_0}{2}$ time for a probe packet from the source. If m is selected for a probe, in the worst case f_i needs to further wait r_i time for an ACK from f_{i+1}; whereas in the ideal case, f_i needs to further wait r_i time for the ACK from f_{i+1}. Therefore given the probe frequency p, f_i can at most store $(0.5 + p)r_0 \times \nu$ packets at a time in both the worst and ideal cases.

PAAI-2. In the worst case, on receiving m, f_i waits r_i time for an ACK from f_{i+1}, $r_0 - r_i$ time for a probe from the source, and r_i time for an ACK from f_{i+1} again, which gives the worst case

storage overhead O($2r_0\nu$). In the ideal case, f_i only needs to wait r_i time for the ACK from f_{i+1}. Therefore in ideal case the storage bound is O($r_0 \times \nu$) packets at a time.

4.9 Simulation Results and Analysis

We implement a simulator to study the *average-case* performance of the proposed protocols, and also contrast the average-case results with the theoretical results (as listed in Table 4.2). Through simulations, we not only validate our theoretical results and make comparisons, but also derive new observations missing from theoretical analysis by itself.

4.9.1 Methodology

Adversary. Note that, in practice, an adversary usually directly compromises a *node*, corrupting the traffic flowing through that node at the adversary's will. We emulate such a realistic scenario by setting malicious *nodes* in the path to perform malicious packet corruption activity. We simulate the adversary's *optimal strategy* by deploying exactly one malicious node on the path (Corollary 8). Recall that, in our protocols, if a malicious node corrupts packets, it can manifest high corruption rates only on its adjacent links. We also set the adversary to employ the following tactics: (i) Since the full-ACK scheme and PAAI protocols ensure that the adversary cannot gain benefit by corrupting different packets at different rates (Corollary 8), the adversary corrupts all types of packets at the same rate. (ii) Without loss of generality, we assume that, when the malicious node receives but corrupts a data packet, on receiving an ACK request it will still send back the ACK as if it were functioning correctly. In this way, a malicious node f_i's corruption activity always increases the corruption score of its downstream adjacent link l_i. Therefore l_i is the target to identify.

Topology and Parameters. Recall the example topology given in Figure 2.1. We simulate the proposed protocols on one path with various lengths and varying locations of the malicious link. Due to lack of space, here we only present the results for an arbitrary setting where $d = 6$ and f_4 is set to be the node controlled by the adversary (results from other settings present similar trends and conclusions). According to our aforementioned adversarial setting, the malicious packet corruption

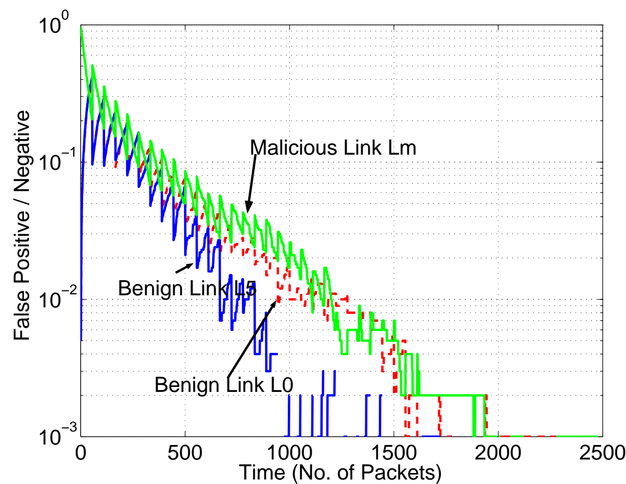
will directly increase l_4's corruption score; thus l_4 is the target link for our fault localization protocols to identify. In the following we also call l_4 as the malicious link l_M. We follow the example parameter settings used in our previous theoretical analysis, i.e., we set benign per-link loss rate threshold $\rho = 0.01$ and malicious per-link corruption rate $T_{dr} = 0.03$ (we implement this by setting a corruption rate of 0.02 for the malicious *node* f_4). However, recall from Corollary 11 and Table 4.1 that the performance of PAAI-1 does not degrade in the case of longer paths and higher natural loss rates. Each packet traversing a link (or the malicious node) has an independent probability of being corrupted bi-directionally below the corresponding corruption rate threshold of that link (or the malicious node). We also set per-link bi-directional latency distributed within 0 to 5 ms uniformly at random.

Evaluation Metrics. We evaluate (i) fault localization false positive and negative rates (which directly relate to detection delays) and (ii) storage overhead of each node for the full-ACK and PAAI protocols. We did not simulate the communication overhead because the theoretical analysis already gives straightforward and tightly bounded results. We run the simulation 10000 times for each protocol to calculate the false positive and false negative rates and plot their dynamics over time. Recall from Table 4.1 that storage overhead directly depends on packet origination rate; as such we evaluate it for different orders of origination rate: 1000 and 100 data packets per second (the storage overhead under a source's sending rate of 10 packets per second is too low to exhibit any insightful traits).

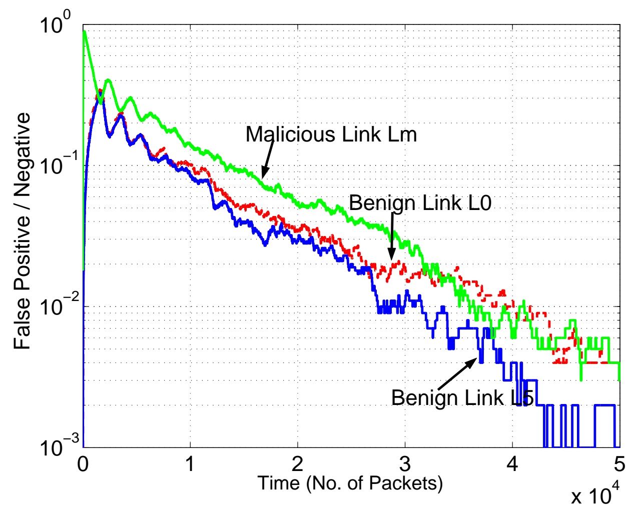
4.9.2 Results and Analysis

As presented below, we are able to both validate our theoretical results and to derive new and interesting observations from the simulation results.

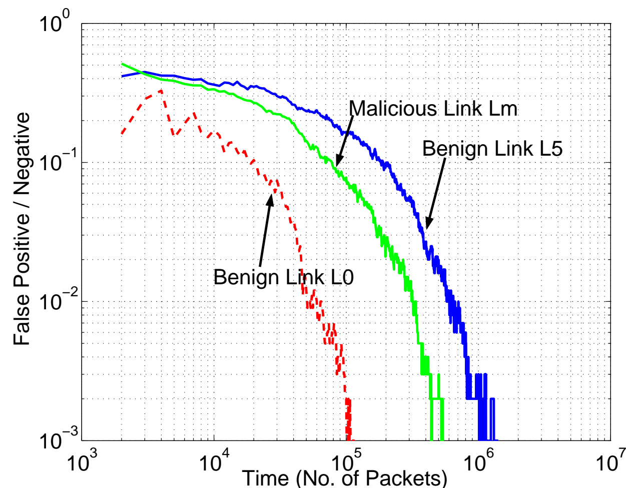
False positive and negative rates. Figure 4.1 plots the false positive and false negative rates observed from 10000 simulation runs for each protocol. From the figure we can observe that, given the same false positive threshold $\delta = 0.03$, the detection delays are nearly half of the corresponding theoretical bounds. We summarize the comparisons between theoretical and experimental results in



(a) Full-ACK scheme. We use logarithmic scale for the y-axis.

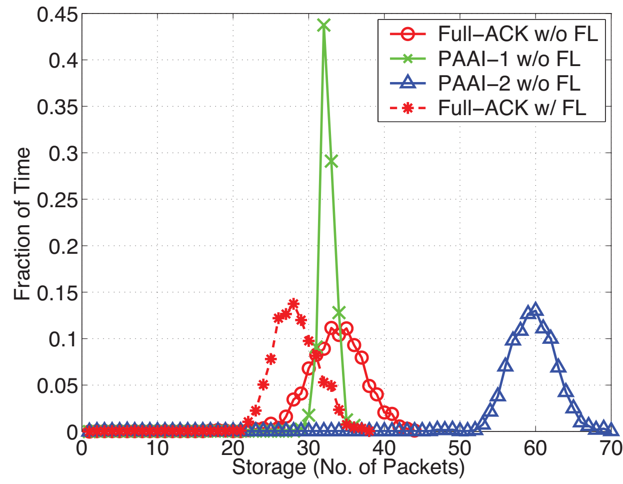


(b) PAAI-1. We use logarithmic scale for the y-axis.

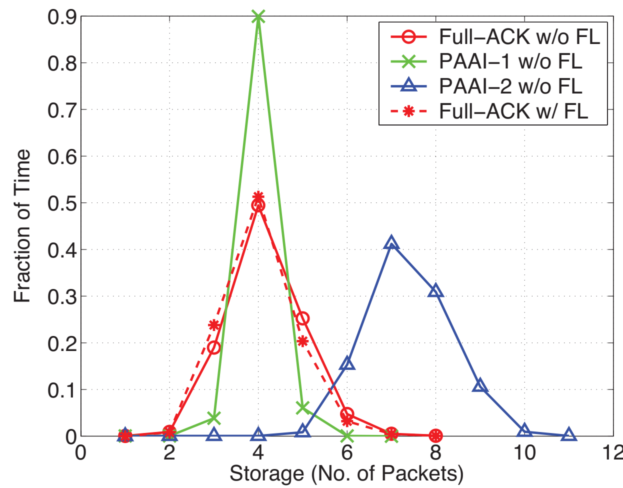


(c) PAAI-2. We use logarithmic scale for both axes.

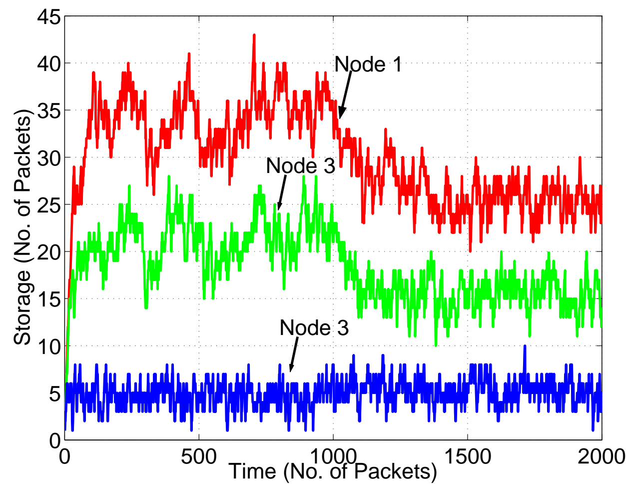
Figure 4.1: False positive and negative rates. The time is measured by the number of packets sent by the source.



(a) Sending rate = 1000 pkt/sec.



(b) Sending rate = 100 pkt/sec.



(c) Storage traits with sending rate = 1000 pkt/sec in full-ACK scheme.

Figure 4.2: Storage overhead. The storage is measured by the number of packets stored at any given time.

Protocol	Detection Delay (minutes)		Storage (# pkt)	
	bound	average	bound	average
Full-ACK	0.25	0.17	12	3.2
PAAI-1	9	4.2	3.2	3.0
PAAI-2	100	50	12	6.4
Statistical FL [21]	3333	N/A	< 1	N/A

Table 4.2: Comparison of detection rates between theoretical results and simulation results. The source's sending rate is set to 100 data packets per second. The storage overhead is the average number of packets stored in f_1 with the presence of a malicious link l_4.

Table 4.2. In addition, we can see that in PAAI-2, the source takes more time to accurately observe the per-link corruption rate for a link farther away from the source. This fact can be theoretically proved via the mathematical formula (we defer the proof to the full version).

Storage overhead. We launch two different sets of simulations to study the characteristics of storage overhead in fault localization protocols. In each scenario if a fault localization protocol reaches the converged condition (after 10^3, 2.5×10^4 and 3×10^5 data packets sent by the source in full-ack, PAAI-1 and PAAI-2 schemes, respectively), we assume the source bypasses the identified l_4 by replacing f_4 with a honest node $f_4^{\prime}$ to connect nodes f_3 and f_5 (we implement this in the simulation by resetting f_4's corruption rate to zero). We label cases where adversary identification comes into play as "w/ FL". We also simulate the case where the existing adversary is not identified and bypassed, which is labeled as "w/o FL".

We first investigate the storage overhead of a single node f_1 (which has the highest storage overhead, as we show later) under different source's sending rates (1000 and 100 data packets per second). We first let the source send 2000 data packets in total, within which only the full-ACK scheme can reach the converged condition. However, we present the results for the full-ACK scheme in both "w/ FL" and "w/o FL" cases to compare with the PAAI protocols. Figures 4.2(a) and 4.2(b) present f_1's storage overhead when the source's sending rate is 1000 or 100 data packets per second, respectively. It is apparent that the storage overhead decreases with the lower sending rate. We further observe that, in the "w/o FL case, PAAI-1 possesses the lowest storage overhead; and the storage overhead of each protocol increases roughly linearly with the source's sending rate.

This fact complies with our theoretical bounds (Table 4.1). In addition, it is clear that the full-ACK scheme achieves a lower storage overhead after bypassing the adversary ("w/ FL"). Therefore, though the full-ACK scheme presents the highest theoretical bound of worst-case storage overhead, it achieves the lowest storage overhead in practice when fault localization comes into play. This observation implies that, in essence, a protocol with a lower detection delay benefits more in the ideal cases where packet corruption activities are rare after the adversary is quickly bypassed.

In another simulation, we investigate the storage overhead of nodes at different locations in the path and the influence of fault localization on storage overhead. Since the full-ACK scheme has the lowest detection delay, we only present the simulation results of the full-ACK scheme due to space limitations (the results derived from other protocols present common trends). To make the influence of fault localization more graphically obvious, we enlarge the corruption rate of f_4 to 0.1. In this simulation we let the source send 2000 data packets at the rate of 1000 data packets per second, and bypass the adversary after sending 1000 data packets. Figure 4.2(c) plots the resulting dynamics of the storage overhead of nodes f_1, f_3 and f_5, from which we can observe that, nodes closer to the destination have lower storage overhead and are less affected after adversarial packet corruption. This observation can be explained according to the theoretical analysis in Section 4.8.4.

4.10 Summary of Results

From the theoretical and experimental results, we can make the following major observations:

Theory vs. Simulation. The average-case results derived from our simulations are within the corresponding theoretical bounds. For the detection delay, the average-case results are nearly two times better than the corresponding theoretical results. For the storage overhead, the average-case result of the full-ACK scheme is far smaller than its worst-case bound, thanks to its fast convergence. The PAAI-1 protocol also presents low storage overhead, even with the presence of an adversary.

Practicality. We make the following conclusions about the trade-off between the three performance metrics achieved by the various protocols: (i) The full-ACK scheme offers the shortest

detection delay and incurs a low storage overhead, but at the cost of impractical communication overhead. (ii) The PAAI-1 protocol offers a practical (though not the best) detection delay and communication and storage overhead simultaneously. More specifically, given that each data packet is 1.5KB (which is the currently popular MTU standard), per Figures 4.2(a) and 4.2(b), PAAI-1 introduces less than 45KB additional storage overhead even at its peak value under an intense packet sending rate of 1.5MB per second, and around 6KB at its peak value under a packet sending rate of 150KB per second. Furthermore, by setting the sampling rate $p = \frac{1}{5d^2}$, PAAI-1 poses only around 3% additional communication overhead in a path with length $d = 6$, per Table 4.1; while the detection delay is 45 minutes given by the theoretical bound, and around 20 minutes on average per Table 4.2 (in previous analysis and simulation we set $p = \frac{1}{d^2}$). (iii) The PAAI-2 protocol presents worse performance compared to the full-ACK scheme and PAAI-1 protocol, but still presents a more practical detection delay compared to the statistical FL scheme [21] (see below). (iv) The statistical FL protocol [21] incurs almost optimal communication and storage overhead, but achieves a rather impractical detection delay – nearly 50 hours in the worst case (Table 4.2). We conclude that PAAI-1 offers the most desirable trade-off between the performance metrics. In contrast, all the other protocols only optimize at most two performance metrics at the cost of deteriorating the other metric(s) undesirably.

4.11 Combination

So far we have explored three different basic approaches, namely: (i) every node acknowledges every corrupted data packet (exemplified by the full-ACK scheme), (ii) every node acknowledges a selected fraction of data packets (instantiated by the PAAI-1 protocol), and (iii) a selected subset of nodes acknowledge every data packet (represented by the PAAI-2 protocol). Intuitively, it might be tempting to consider combinations of the above basic approaches in order to improve upon a certain performance metric. However, as we demonstrate below, the combinations may not necessarily achieve a better trade-off between the performance metrics as compared to the basic approaches, and may therefore be unfavorable in practice. Specifically, although a combination may further optimize a certain performance metric, other metrics can degrade undesirably at the

same time. Due to lack of space, we will briefly discuss two sample combinations and analyze the corresponding tradeoff.

Combination 1. By combining the basic approaches (a) and (b) above, we can design a protocol where every node must acknowledge a selected fraction of *corrupted* data packets. The PAAI-1 protocol can be easily modified to follow the above approach. Specifically, instead of using a secret key known only to S to implement the probe function, we will use the secret key K_d shared between S and f_d. Now, on receiving a data packet, f_d can independently decide whether it must be acknowledged. For a sampled data packet m, S will send out a probe only if it fails to receive an ack from f_d. The remaining details follow from PAAI-1. While retaining the same detection delay as PAAI-1, the new protocol further reduces the communication overhead, since S now solicits an onion report for only a *corrupted* sampled packet (instead of every sampled packet in PAAI-1). However, the storage overhead increases: in the worst case, on receiving m, each node must first wait an additional r_0 time for an ACK from f_d, such waiting time which was not required in PAAI-1. Its performance is summarized in Table 4.1.

Combination 2. By combining the basic approaches (b) and (c) above, we can design a protocol where one *selected* node acknowledges a selected fraction of data packets. Similar to Combination 1, we will use a probe function that is implemented using the secret key K_d. The data packet structure will be similar to that in PAAI-2. Now, on receiving a data packet, f_d can independently decide whether it must be acknowledged. If an intermediate node receives a *valid* ACK from f_d, it immediately knows that the packet was sampled and that there will be no further probe. For a sampled data packet, S will send out a probe only if it fails to receive an ACK from f_d. The remaining details follow from PAAI-2. It is intuitive to see the new protocol incurs lower communication overhead than both PAAI-1 and PAAI-2, but at the price of a longer detection delay. Its performance is summarized in Table 4.1.

4.12 Summary

In this chapter, we address the problem of designing a secure fault localization protocol that offers a *practical* trade-off between detection delay, communication overhead, and storage overhead. To this end, we systematically explore the design space of path-based fault localization protocols where an ACK packet acknowledges a single data packet, and propose a set of basic protocols where each protocol exemplifies a design dimension. Based on our theoretical analysis and simulation results, we conclude that the proposed PAAI-1 protocol achieves the best trade-off, and as a result is more practical than the other protocols. We note, however, that PAAI bears some limitations in its extensibility and generality; e.g., both PAAI-1 and PAAI-2 require loose time-synchronization, which, although a viable assumption for many network settings, might limit their applicability. We address these limitations in the next chapter.

Chapter 5

ShortMAC

Existing fault localization protocols cannot achieve a practical tradeoff between security and efficiency. For example, they require unacceptably long detection delays and require monitored flows to be impractically long-lived. Though PAAI improves the practicality of fault localization compared to prior work, in both PAAI-1 and PAAI-2, an ACK packet sent by a router only acknowledges a *single* corresponding packet. Intuitively, acknowledging *a set of* packets with one ACK packet might further reduce the communication overhead, eliminate the need of packet sampling, and eventually reduce the detection delay. In this chapter, we propose an efficient path-based fault localization protocol called ShortMAC, in which routers locally cache fingerprints for a set of packets it receives, and periodically send the fingerprints with a single ACK packet to the source. By leveraging *probabilistic* packet authentication and efficient fingerprinting data structure, ShortMAC achieves $100 - 10000$ times lower detection delay and overhead than related work.

5.1 Introduction

In this chapter, we propose ShortMAC, an efficient fault localization protocol to provide a *theoretically proven* guarantee on end-to-end *data-plane* packet delivery even in the presence of sophisticated adversaries. More specifically, we aim to guarantee that, given a correct routing infrastructure, a benign source node can quickly find a non-faulty path along which a very high fraction of packets can be correctly delivered. Our key insights are two-fold:

Insight 1. We first observe that localizing data-plane faults along a communication path can be reduced to monitoring packet *count* (number of received packets) and packet *content* (payload of received packets) at each router on that path. Furthermore, if packets can be *efficiently* authenticated, packet count also becomes a verifiable measure of packet content, because forged packets (with invalid contents) will be dropped by the routers and manifest an observable deviation in the packet count. Thus, routers can dramatically reduce storage overhead by storing counters instead of packet contents.

Insight 2. We also observe that we can achieve a high packet delivery guarantee via fault localization by *limiting* the amount of malicious packet drops/modifications, instead of perfectly detecting *each single* malicious activity. Furthermore, strong *per-packet* authentication to achieve perfect detection of *every single* bogus packet is unnecessary for *limiting* the adversary's ability to modify/inject bogus packets. Instead, the source can use much shorter packet-dependent random integrity bits as a weak authenticator for each packet such that each forged packet has a *non-trivial probability* to be detected. In this way, if a malicious node modifies or injects more than a threshold number of (e.g., tens of) packets, the malicious activity will cause a detectable deviation on the counter values maintained at different routers. Essentially, ShortMAC traps an attacker into a dilemma: if the attacker inflicts damage worse than a threshold, it will be detected, which may lead to removal from the network; otherwise, the damage is limited and thus a guarantee on data-plane packet delivery is achieved.

Contributions. 1) We propose a data-plane fault localization protocol ShortMAC that achieves high security assurance with 100 - 10000 times lower detection delay and storage overhead than related work.

2) We derive a provable lower bound on successful end-to-end packet forwarding rate, by limiting adversarial activities instead of perfectly detecting every single malicious action which would incur high protocol overhead.

3) We theoretically derive the performance bounds of ShortMAC and evaluate ShortMAC via SSFNet-based [6] simulation and Linux/Click router implementation. Our implementation and

evaluation results show that ShortMAC causes *negligible* throughput and latency costs while retaining a high level of security.

5.2 ShortMAC Overview

ShortMAC monitors both the packet *count* and *content* at each hop. Specifically, a router maintains per-path counters to record the number of received data packets originated from the source in the current epoch. To ensure that the packet count is a verifiable measure of the desired monitoring task, we require that both packet modification and injection by malicious (colluding) routers affect counter values at benign nodes.

We first introduce the concept of an *epoch* to facilitate our protocol design and formal analysis:

Definition 12. *An end-to-end communication is composed of a set of consecutive* ***epochs****. An epoch* for an end-to-end path *is defined as the duration of transmitting a sequence of N data packets by a source S toward a destination f_d along that path. The epochs are* asynchronous *among different paths.*

At the beginning of each epoch denoted by e_k, a source node S selects a path p and starts sending packets along p, with each packet carrying several ShortMAC authentication bits. The routers verify the authentication bits in each received packet based on the symmetric key shared with the source node, increment locally stored counters for p accordingly, and forward only the authentic packets. Due to the ShortMAC authentication bits, modified/injected packets can result in an observable deviation in the counter values which enable fault localization by the source at the end of each epoch.

At the end of each epoch e_k, the source S retrieves the counter reports from all routers and the destination in p for e_k, via a secure channel as Section 5.3 will describe. S then performs fault detection based on the retrieved counters, and bypasses the detected faulty link (if any) by finding another path excluding the identified faulty link (e.g., via source routing, path splicing [72], pathlet routing [35], or SCION routing [96]). The detection result is only used by S itself for selecting its own routing paths, instead of being shared with other nodes which is susceptible to framing attacks.

Although the high-level *epoch-based* protocol flow (nodes periodically send certain locally logged traffic summaries to the source) bears great similarity with Fatih [71], AudIt [14], and Statistical FL with sketch [21], both Fatih and AudIt use simple counters or Bloom Filters without *keyed* hash functions as the traffic summaries, thus remaining vulnerable to packet modification/injection attacks. In addition, the sketch-based packet fingerprints used in Statistical FL consume several hundreds of bytes *for each path*. In contrast, ShortMAC efficiently tackles packet modification attacks with only several-byte counters as shown below.

5.2.1 ShortMAC Packet Authentication

Our approach is to turn packet count into a reliable measure of packet content so that routers only need to store space-efficient counters. To this end, the integrity of the source's data packets must be ensured in order to detect malicious packet modification during the forwarding path; otherwise, a malicious router can always perform packet modification attacks without affecting the counter values, or inject bogus packets on behalf of the source to manipulate the counter values of the reporting routers. Hence, we reduce the problem to how the source node can authenticate its packets to all the routers in the path. However, traditional broadcast authentication schemes provide high authenticity for every *single* message, which is neither necessary nor practical in our setting where the messages are line-rate packets:

1) Not practical: On one hand, perfectly ensuring the authenticity of *every single* data packet introduces high overhead in a high-speed network. For example, digital signatures or one-time signatures for per-packet authentication is either computationally expensive or bandwidth-exhaustive, and using amortized signatures would either fail in the presence of packet loss or incur high communication overhead [63]. Attaching a Message Authentication Code (MAC) for each node along the path (as is used by Avramopoulos et al. [17]) is too bandwidth-expensive (e.g., reserving a 160-bit MAC space for each hop). In addition, TESLA authentication [77] would require time synchronization and routers to cache the received packets until the authentication key is later disclosed (longer than the end-to-end path latency). Finally, some recently proposed multicast/broadcast authentication schemes still require considerable communication overhead (e.g., up to hundreds of bytes per packet [64]) or multiple rounds for authenticating a message [29].

2) Not necessary: On the other hand, as we aim to *limit* the damage the adversary can inflict for a lower-bound guarantee on data-plane packet delivery, perfect per-packet authenticity is not necessary. Instead, our goal only requires the authenticity of a large fraction of data packets.

ShortMAC approach. Based on these observations, we propose ShortMAC, a light-weight scheme trading per-hop overhead with the adversary's ability to forge only a few (e.g., tens of) packets. More specifically, in ShortMAC, the source attaches to each packet a *k-bit random nonce, called k-bit MAC*, for each node on the path, where the parameter k is significantly less than the length of a typical MAC (e.g., $k = 2$). To construct the k-bit MAC for f_i, the source S uses a Pseudo-Random Function (PRF) which constructs a k-bit string as a function of the packet m and key K_{si} shared between S and f_i. We rely on the result that the output k-bit MAC is indistinguishable from a random k-bit string to any observer without the secret key K_{si} [67]. Each router f_i maintains two path-specific counters C_i^{good} and C_i^{bad} to record the numbers of received packets along that path with correct and incorrect k-bit MACs, respectively, in the current epoch. Such a scheme considerably reduces communication overhead compared to attaching entire MACs while retaining high security assurance and communication throughput, as shown later.

5.2.2 ShortMAC Example

We present a toy example in Figure 5.1 to provide intuition on how ShortMAC enables data-plane fault localization. Suppose the source node sends out 1000 packets in a certain epoch. The source uses a PRF taking a secret key as input which can map a packet into two bits (called 2-bit MAC) uniformly at random to anyone without knowledge of the secret key. The source computes the PRF four times for each packet, taking as input the epoch symmetric key shared with f_1, f_2, f_3, and the

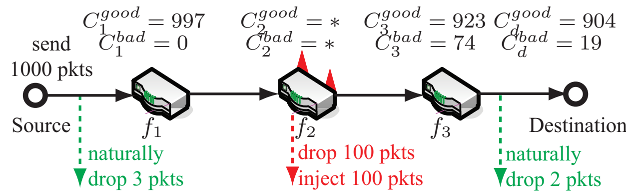


Figure 5.1: Fault localization example with ShortMAC using 2-bit MAC. f_2 is malicious.

destination, respectively. Then the source attaches the resulting four 2-bit MACs to each packet.

Among the 1000 packets, suppose three packets are spontaneously dropped on the first link, and router f_1 receives the remaining 997 packets. f_1 computes the PRF on each of the received packets taking as input the epoch symmetric key shared with the source, and compares the resulting 2-bit MACs with the one embedded in each packet. All verifications are successful, so f_1 has $C_1^{good} = 997$ and $C_1^{bad} = 0$. Suppose the malicious router f_2 drops 100 good packets and injects 100 malicious packets. For each injected packet, f_2 needs to forge 2-bit MACs for both f_3 and the destination that "authenticate" the fabricated data content. However, since f_2 does not know the corresponding epoch symmetric keys of f_3 and the destination, f_2 can only guess the 2-bit MACs for its injected packets. Since the 2-bit MACs produced by the PRF are indistinguishable from random bits, f_2 can correctly guess each 2-bit MAC with probability $\frac{1}{4}$. Since f_2 must guess two correct MACs, each forged packet will be accepted by the destination with probability $\frac{1}{16}$. Suppose next that 26 of the 100 2-bit MACs that f_2 forged for f_3 happen to be valid with respect to the the malicious data content. f_3 thus computes $C_3^{bad} = 100 - 26 = 74$ and $C_3^{good} = 997 - 100$ (dropped legitimate packets) $+26$ (bogus but undetected packets) $= 923$. Similarly, we can analyze the counters for the destination in Figure 5.1, assuming 7 out of the 26 received bogus packets happen to be consistent with their 2-bit MACs at the destination.

5.2.3 Fault Localization and Guaranteed θ

At the end of each epoch, routers and the destination report their counter values to the source using a secure transmission approach (detailed in Section 5.3). The source can identify excessive packet drops between f_m and f_{m+1} if the C_{m+1}^{good} value of f_{m+1} is abnormally lower than that of f_m based on the drop detection threshold T_{dr} that is carefully set based on the customized acceptable per-link drop rate. Moreover, this scheme can successfully bound the total number of spurious packets with fabricated k-bit MACs that the adversary can inject, because at least one of the downstream recipient routers will detect the inconsistency of the k-bit MACs with a non-trivial probability, thus having a non-zero C^{bad} value. For example in Figure 5.1, although f_2 can claim any values for its own counters, no matter what values f_2 claims, the source can notice excessive packet loss and a large number of fake packets either between f_1 and f_2, or f_2 and f_3. Hence one of f_2's malicious

links will be detected by the source.

Once the source S bypasses all malicious links identified by ShortMAC, S can find a working path with no excessive packet corruption at any link, thus achieving a guaranteed successful forwarding rate θ. With secure fault localization, a source can find a working path after exploring at most Ω paths, where Ω is the number of malicious links in the network. In contrast, with only end-to-end path monitoring, a source may explore a number of paths exponential to Ω as we showed in Section 1.1.

5.3 ShortMAC Details

In this section we describe the ShortMAC protocol in detail, where the source can either guarantee that a high fraction θ of its data has been correctly forwarded if no malicious activities are detected, or can bypass the faulty links and find a working path after exploring a number of paths linear to the number of faulty links.[1] In the following, we first formalize the ShortMAC packet format and then detail the protocol.

5.3.1 ShortMAC Packet Format

A source node S adds a trailer to each data packet it sends:

$$trailer = \langle SN, \mathcal{M}_1, \ldots, \mathcal{M}_d \rangle, \tag{5.1}$$

where SN is a *per-path* sequence number to make each packet unique along the same path to prevent packet replay attacks, and $\mathcal{M}_i$ denotes the k-bit MAC computed for f_i, which is constructed in a recursive way starting from f_d:

$$\begin{aligned}
\mathcal{M}_d &\leftarrow PRF_{K_{sd}}(IP_{invar}||SN||TTL_d) \\
\mathcal{M}_{d-1} &\leftarrow PRF_{K_{s(d-1)}}(IP_{invar}||SN||TTL_{d-1}||\mathcal{M}_d) \\
&\ldots\ldots \\
\mathcal{M}_i &\leftarrow PRF_{K_{si}}(IP_{invar}||SN||TTL_i||\mathcal{M}_{i+1}||\ldots||\mathcal{M}_d)
\end{aligned} \tag{5.2}$$

[1] Recall that forwarding fault localization protocols protocols can only identify faulty links, rather than identifying the nodes [21]. However, given that a malicious node has a limited degree, after bypassing all its malicious links the source can eventually bypass that node.

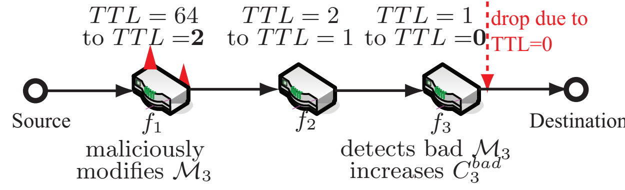


Figure 5.2: Illustration of framing attacks. f_1 is malicious.

where "||" denotes concatenation and $PRF_{K_{si}}(\cdot)$ denotes a PRF keyed by the symmetric key K_{si} shared between S and f_i. As previously discussed, the output of this PRF can be guessed correctly with probability no larger than $\frac{1}{2^k}$ by anyone without the secret key K_{si} [67]. In addition,

1) IP_{invar} denotes the invariant portion of the original IP packet that should not be changed at each router during forwarding, including the packet payload and IP headers excluding variable fields such as `TTL`, `RecordRoute` IP option, `Timestamp` IP option etc. If these invariant fields are unexpectedly changed during forwarding, each downstream router can detect inconsistency between the (modified) packet and embedded k-bit MAC with a non-trivial probability $1 - \frac{1}{2^k}$ and thus increase its C^{bad} counter.

2) TTL_i denotes the expected TTL value at router i. Without authenticating this field in the k-bit MAC, a malicious router can strategically lower the TTL field to cause packet drop at a remote downstream router due to zero TTL value, thus performing framing attacks. For example in Figure 5.2, if $\mathcal{M}_i$ in Eq.(5.2) had not authenticated the TTL field, f_1 can maliciously change the TTL value in the packets to 2, instead of decrementing it by 1. This causes the packets to be dropped at f_3, thus framing the link between f_2 and f_3.

3) $\mathcal{M}_i$ also authenticates the downstream $\mathcal{M}_{i+1}, \ldots, \mathcal{M}_d$, so that if a malicious router f_m changes any of these downstream k-bit MACs, f_i can observe the inconsistency in $\mathcal{M}_i$ with a probability $1 - \frac{1}{2^k}$ and increase its C_i^{bad} value. Otherwise, the protocol is vulnerable to framing attacks. For example in Figure 5.2, if $\mathcal{M}_i$ in Eq.(5.2) had not authenticated the downstream $k-$bit MAC field, f_1 can maliciously modify $\mathcal{M}_3$ in the packets which causes f_3 to detect inconsistent $\mathcal{M}_3$ with a non-trivial probability and increase C_3^{bad}, thus framing the link between f_2 and f_3.

5.3.2 Protocol Details

Formally, ShortMAC consists of Request, Report, Identify, Bypass and Send stages, described as follows.

Stage 1: Request with hop-by-hop reliable transmission

At the end of each epoch e_k (i.e., after sending every N data packets), the source S will send a request packet, denoted by `request`$= (S, p)$, along the path $p = (f_1, \ldots, f_d)$ used in epoch e_k. This `request` asks each router f_i and the destination f_d to report their counter values (C_i^{bad} and C_i^{good}) along the reverse of path p. Then S expects these counter reports in Acknowledgment (`ACK`) packets from all the nodes in p containing the requested information authenticated with each node's K_{si}.

Note that a spontaneous loss of `request` or `ACK` packets will prevent S from learning the counter values by certain routers in the previous epoch. To preclude such damage, we use the following **hop-by-hop reliable transmission** approach: when f_i forwards either a `request` or an `ACK` packet to its neighbor, f_i tries up to r times (e.g., $r = 5$) until it gets a confirmation from the neighbor. In this way, the failure of receiving a `request` or `ACK` packet can only indicate malicious drops – more precisely, with the probability of $1 - \rho^r$, where ρ is the natural loss rate of a link. Then thanks to the Onion ACK approach presented below, the source can immediately identify a malicious link that drops or modifies `request` or `ACK` packets; hence the `request` packets do not need to be authenticated by the source as we show below.

Stage 2: Report with Onion ACK

Upon receiving a `request`, f_i starts a timer whose value is the maximum round trip time from f_i to the destination.[2] At the same time, f_i constructs its local report $\mathcal{R}_i$:

$$\mathcal{R}_i = \left(f_i, p, C_i^{good}, C_i^{bad}\right) \tag{5.3}$$

[2]We can expect a reasonable upper bound of link latency in benign cases, which can be used to compute the maximum round trip time according to the hop count from f_i to the destination. Avramopolous et al. [17] first introduced the use of such a timer.

where f_i is the node id, p is the requested path, and C_i^{good} and C_i^{bad} are the counter values from the previous epoch. Each router finds C_i^{good} and C_i^{bad} corresponding to path p based on the source and destination IDs in p (assuming single path routing). Once the report is constructed:

Case 1. If f_i receives an `ACK` $\mathcal{A}_{i+1}$ from neighbor f_{i+1} before the timer expires, f_i further commits $\mathcal{R}_i$ into a new `ACK` $\mathcal{A}_i$ by combining the received $\mathcal{A}_{i+1}$ via an **Onion `ACK`** approach:

$$\mathcal{A}_i = \big(\mathcal{R}_i, \mathcal{A}_{i+1}, \mathrm{MAC}_{K_{si}}(\mathcal{R}_i || \mathcal{A}_{i+1})\big), \tag{5.4}$$

$\mathrm{MAC}_{K_{si}}(\cdot)$ denotes a message authentication code computed with K_{si}. Then, f_i forwards $\mathcal{A}_i$ to f_{i-1} toward S.

Case 2. If f_i receives no `ACK` packet from f_{i+1} before the timer expires, f_i will initiate a new `ACK` with its local report and send it to f_{i-1}.

The Onion `ACK` prevents the adversary from *selectively* dropping the `request` or the reports of a certain router f_i and framing a benign link l_i [97]. In Onion `ACK`, all the reports are *combined* and *authenticated* in one `ACK` packet at each hop so that a malicious node can only drop or modify the onion report from its *immediate* neighbors. Intuitively, if f_m drops or modifies the received `request` or Onion `ACK`, the source can receive the correct reports from $f_1, \ldots, f_{m-1}$ but not from $f_m, \ldots, f_d$; hence one of f_m's links will be pinpointed by the source node, in the identify stage described below.

After sending the local reports, each router f_i resets C_i^{good} and C_i^{bad} to zero, to be used for the next epoch along path p (if p is still used).

Stage 3: Identify

Upon receiving an Onion `ACK` $\mathcal{A}_1$ from f_1, S first iteratively retrieves $\mathcal{A}_1, \mathcal{A}_2, \ldots$ in order, until it either completes at d or fails at j ($j \neq d$). S can verify if a certain retrieved report $\mathcal{R}_i$ is valid by checking the embedded message integrity code $\mathrm{MAC}_{K_{si}}(\mathcal{R}_i || \mathcal{A}_{i+1})$. When the check fails at j ($j \neq d$), S will immediately identify l_j as faulty due to the use of reliable hop-by-hop transmission and Onion ACK. For example, if S receives no report it will identify l_1 as faulty ($j = 1$).

In addition, S extracts $\mathcal{R}_1, \ldots, \mathcal{R}_j$ in turn which include the C_i^{bad} and C_i^{good} values. A non-zero C_i^{bad} implies the existence of malicious packet injection between f_i and S. However, S cannot blame l_i simply whenever $C_i^{bad} > 0$, say, $C_i^{bad} = 1$. A possible scenario is that a malicious node f_{i-2} injects a fake packet, but the k-bit MAC intended for f_{i-1} "happens" to be consistent with the fake packet at benign node f_{i-1} (e.g., when $k = 2$, this can happen with probability 0.25). In this case, f_{i-1} will forward the fake packet which f_i may detect and thus increase C_i^{bad}. Similarly, due to natural packet loss, S cannot simply accuse link l_i when $C_i^{good} < C_{i-1}^{good}$. Therefore, we leverage two detection thresholds T_{in} and T_{dr}, where T_{in} is the injection detection threshold for the number of injected packets on each link, and T_{dr} is the drop detection threshold for the fraction of dropped packets on each link. As we will show in Section 5.5, these thresholds reduce false positives while limiting the adversary's ability to corrupt packets and ensuring a lower bound on the successful packet forwarding rate. The detection thresholds are used in two detection procedures:

1) check-injection**:** S checks the extracted $C_1^{bad}, C_2^{bad}, \ldots, C_j^{bad}$ values *in order*. If $C_i^{bad} \geq T_{in}$ for some i, then S identifies l_i as faulty and the check-injection procedure stops.

2) check-dropping**:** If no fault is detected by check-injection, S further checks the extracted C_1^{good}, C_2^{good}, ..., C_j^{good} values *in order*. If $C_i^{good} < (1 - T_{dr}) \cdot C_{i-1}^{good}$ (with $C_0^{good} = N$) holds for certain i, then S identifies l_i as faulty and the check-dropping procedure terminates.

Stage 4: Bypass and Send

If Stage 2 outputs any malicious link l_m, S selects a new path excluding the previously detected malicious links and sends its packets with ShortMAC authentication shown in Eq.(5.2). Each node f_i examines its corresponding k-bit MAC $\mathcal{M}_i$ in each packet to increase C_i^{good} or C_i^{bad} accordingly. In addition, each router remembers the last seen *per-path SN* embedded in the packets as shown in Eq.(5.1), and discards packets with older SN in that path.

5.4 Security Analysis

This section discusses ShortMAC's security against data-plane attacks by malicious routers. Section 5.5 provides theoretical proofs on ShortMAC's security. In our adversary model, a malicious

router can drop and inject data packets, `requests` and `ACK`s, and can send arbitrary counter values in its reports. We show that ShortMAC is secure against a single malicious router (say, f_m) as well as multiple colluding nodes.

Corrupting data packets. Dropping legitimate data packets by f_m will cause a discrepancy of the counter values between f_m and its neighbors. For example, if f_m correctly reports C_m^{good}, then $C_m^{good} - C_{m+1}^{good}$ will exhibit a large discrepancy; if f_m reports a lower C_m^{good}, then $C_{m-1}^{good} - C_m^{good}$ will exhibit a large discrepancy. Hence, either l_{m-1} or l_m will become suspicious. Moreover, if f_m injects/modifies packets, $\mathcal{M}_{m+1}$ will be inconsistent at f_{m+1} with high probability and cause a non-zero C_{m+1}^{bad}. Hence, both dropping and injection attacks can be detected as long as the source can learn the correct counter values in the `ACK` packets sent by the nodes between f_m and the destination, which is described next.

Corrupting `ACK`s or `requests`. Since the `requests` are not authenticated by S, f_m can modify the content of `requests` (such as the source ID and the path); however, this will result in S failing to receive the correct counter reports from f_{m+1} (or f_m) $, \ldots, f_d$ in p, thus causing l_{m+1} or l_m to be detected. f_m cannot selectively drop the `ACK` reports due to the use of Onion `ACK`. Instead, f_m can only drop the `ACK`s or `requests` from its *immediate* neighbors, which will again harm its incident links.

Replay, reorder, and traffic analysis attacks. To prevent replay and reorder attacks, each packet contains a per-path sequence number SN in Eq.(5.1) and each router discards packets with older SNs. Hence, the replayed and reordered packets will be dropped at the next-hop benign node without influencing the counter values of benign nodes. Note that because ShortMAC runs on a per-path basis and a SN is a *per-path* sequence number providing natural isolation across different paths, packets *along the same path* are expected to maintain the same order during forwarding as they were sent by the source in benign cases. On the other hand, if f_m falsely reports a large SN, f_{m+1} will drop the subsequent packets and l_m will be identified as malicious due to its high packet drop rate. Moreover, the per-path SN can prevent ShortMAC from *traffic analysis attacks*, where f_m attempts to find out the correct k-bit MAC of a packet m by re-sending m with different

k-bit MACs and observing whether the next-hop f_{m+1} forwards the packet. Such traffic analysis is ineffective because f_{m+1} can detect packets with the same SN and each packet is unique due to the use of the per-path SN, and thus f_m cannot send the same packet m with only the k-bit MAC changed.

DoS attacks. A malicious router f_m may launch bandwidth Denial-of-Service (DoS) attacks by generating an excessive amount of packets. However, this attack can be reduced to a packet injection attack and will be reflected by C^{bad}_{m+1}. A malicious router may also attempt to open many bogus flows with spoofed sources to exhaust other routers' state. We can borrow existing work to provide source accountability and reliable flow/path identification [12, 92]. Also note that in our adversary model we consider malicious *routers* which threaten the communication between benign hosts. We do not consider DDoS attacks launched by malicious hosts (botnets), which other researchers have strived to defend against [59, 92, 61]. Hence in our problem setting, a link under DDoS attacks thus exhibiting high loss rate is simply considered a faulty link under our adversary model. Meanwhile, the path setup phase in ShortMAC can be naturally integrated with capability schemes [92] for DDoS limiting, and the per-path counters may also be used for per-path rate limiting.

Collusion attacks. Each of the colluding routers can commit any of the misbehavior discussed above. We can prove by induction that in any case, one of the malicious links of one of the colluding nodes is guaranteed to be detected. A proof sketch is given below.

Consider the base case where two nodes f_m and $f_{m'}$ ($m < m'$) collude. Without loss of generality:

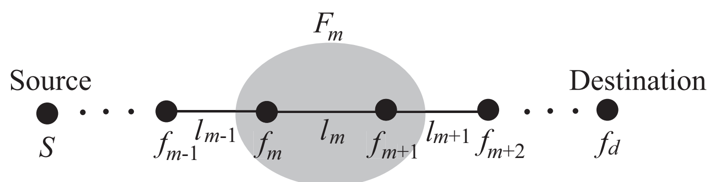


Figure 5.3: Security against colluding nodes – one base case with two adjacent colluding nodes f_m and f_{m+1} forming a virtual malicious node F_m.

1) When f_m and $f_{m'}$ are not adjacent (i.e., $m' > m+1$), the security analysis in Section 5.4 applies to f_m and one of f_m's malicious links will become suspicious if f_m misbehaves. This is because if f_m commits the above attacks, such misbehavior will be reflected in the benign neighbor f_{m+1}'s counters which cannot be biased by $f_{m'}$.

2) When f_m and $f_{m'}$ are adjacent ($m' = m+1$), these two nodes can be regarded as one single "virtual" malicious node F_m with neighbors f_{m-1} and f_{m+2}, as shown in Figure 5.3. (i) If f_m or f_{m+1} drops packets, a discrepancy will exist between C_{m-1}^{good} and C_{m+2}^{good}, no matter what values of C_m^{good} and C_{m+1}^{good} F_m claims. (ii) If f_m or f_{m+1} injects packets, C_{m+2}^{bad} will become non-zero and make l_{m+1} suspicious. In any case, an adjacent link of F_m (a malicious link) will become suspicious.

In the general case with n colluding nodes, we can first group adjacent colluding nodes into virtual malicious nodes as in Figure 5.3, resulting in non-adjacent malicious nodes (including virtual malicious nodes). Then we can show non-adjacent malicious nodes can be detected based on the above analysis.

Despite colluding attackers cannot corrupt packets more than the same thresholds as an individual attacker on any *single* link, they can choose to distribute packet dropping across multiple links. In this case, the total packet drop rate by colluding attackers increases (and is still bounded) linearly to the number of malicious links in the same path, as analyzed in Section 5.5.

5.5 Theoretical Results and Comparison

We prove the $(N, \delta)-$data-plane fault localization (Definition 3) and $(\alpha, \beta)_\delta-$forwarding security of ShortMAC (Definition 5), which in turn yield the $\theta-$guaranteed forwarding correctness (Definition 4). Proofs of the lemmas and theorems are provided in Appendix B.

Comparison of theoretical results. Before presenting the theorems, we first summarize and compare ShortMAC theoretical results with two recent proposals, PAAI-1 [97] and Stat. FL [21] (including two approaches denoted by *SSS* and *sketch*). Table 5.1 presents the numeric figures using an example parameter setting for intuitive illustration, while ShortMAC presents similarly distinct advantages in other parameter settings. In this example scenario shown in the table,

Protocol	ShortMAC	PAAI-1	SSS	Sketch
Detect. Delay (pkt)	3.8×10^4	7.1×10^5	1.6×10^8	$\approx 10^6$
Comm. (extra %)	$< 10^{-5}$	1	1	$< 10^{-5}$
Marking Cost (bytes)	2	0	0	0
Per-path State (bytes)	21	$2{\times}10^5$	4×10^3	≈ 500

Table 5.1: Theoretical comparison with PAAI-1 [97] and Stat. FL [21] (including two approaches *SSS* and *sketch*). Note that the details of *sketch* are not provided in the published paper [21], and the full version of [21] does not present the explicit bounds on detection delay. The above figures for *sketch* are estimated from their earlier work [36]. In this example scenario, $d = 5$, $\delta = 1\%$, $\rho = 0.5\%$, $T_{dr} = 1.5\%$, a symmetric key is 16 bytes, and ShortMAC uses 2-bit MACs. PAAI-1 specific parameters include the "packet sampling rate" set to 0.01, the end-to-end latency set to 25 ms, the source's sending rate set to 10^6 packets per second, each packet hash is 128 bits.

the guaranteed data-plane packet delivery ratio is $\theta = 92\%$. The communication overhead for a router in ShortMAC is 1 extra `ACK` for every 3.8×10^4 data packets in an epoch; the marking cost is 10 bits for the 2-bit MACs in a path with 5 hops, and the per-path state at each router is 21 bytes (16-byte symmetric key, 2-byte C^{good}, 1-byte C^{bad}, and 2-byte per-path SN). Though Barak et al. *proved the necessity* of per-path state for a *secure* fault localization protocol [21], such a *minimal* per-path state in ShortMAC is viable for both intra-domain networks with tens of thousands of routers and the Internet AS-level routing among currently tens of thousands of ASes.

We provide the intuition for ShortMAC's distinct advantages. PAAI-1 or Stat. FL used either low-rate packet sampling or approximation techniques for packet fingerprinting, both of which waste entropy contained in certain packet transmissions, thus resulting in long detection delay (e.g., the transmission results of non-sampled packets will not contribute to the detection phase). In contrast, ShortMAC counts *every* packet transmission thus achieving much faster detection rate. In addition, *secure* packet sampling requires additional packet buffering [97], and packet fingerprint takes considerable memory [21].

Lemma 13. ***Injection Detection:*** *Given the bound δ on detection false negative and false positive rates, the injection detection threshold T_{in} can be set to $T_{in} = \frac{2\ln\frac{2d}{\delta}}{q^4}$, where d is the path length and $q = \frac{2^k-1}{2^k}$ is the probability that a fake packet will be inconsistent with the associated k-bit MAC. The number of fake packets β an adversary can inject on one of its malicious links without being*

detected is limited to:

$$\beta = \frac{T_{in}}{q} + \frac{\sqrt{\left(\ln\frac{2}{\delta}\right)^2 + 8qT_{in}\ln\frac{2}{\delta}} + \ln\ \frac{2}{\delta}}{4q^2}. \tag{5.5}$$

In Lemma 14, we derive N, the number of data packets a source needs to send in one epoch to bound the detection false positive and false negative rates below δ. Due to natural packet loss, a network operator first sets an expectation based on her domain knowledge such that any benign link in normal condition should spontaneously drop less than ρ fraction of packets. We first describe how the drop detection threshold T_{dr} is set when N and δ are given. Intuitively, by sending more data packets (larger N), the *observed* per-link drop rate can approach more closely its *expected* value, which is less than ρ; otherwise, with a smaller N, the observed per-link drop rate can deviate further away from ρ, and the drop detection threshold T_{dr} has to tolerate a larger deviation (thus being very loose) in order to limit the false positive rate below the given δ. On the other hand, a small N is desired for *fast fault localization*. We define **Detection Delay** to be the minimum value of N given the required δ.

Lemma 14. ***Dropping Detection*** *and* (N,δ)*-* ***Data-Plane Fault Localization:*** *Given the bound* δ *on detection false positive and negative rates and drop detection threshold* T_{dr}*, the detection delay* N *is given by:*

$$N = \frac{\ln(\frac{2d}{\delta})}{2\left(T_{dr}-\rho\right)^2\left(1-T_{dr}\right)^d}, \tag{5.6}$$

where d *is the path length. Correspondingly, the fraction of packets* α *an adversary can drop on one of its malicious links without being detected is limited to:*

$$\alpha = 1-(1-T_{dr})^2 + \frac{\beta}{N(1-T_{dr})^d}. \tag{5.7}$$

In practice, T_{dr} can be chosen according to the expected upper bound ρ of a "reasonable" normal link loss rate such that a drop rate above T_{dr} is regarded as "excessively lossy".

Theorem 15. ***Forwarding Security and Correctness:*** *Given* T_{dr}*,* δ*, and path length* d*, we can*

achieve $(\alpha, \beta)_\delta-$*forwarding security where* α *is given by Lemma 14 and* β *is given by Lemma 13. We also achieve* (Ω, θ)*-Guaranteed forwarding correctness with* Ω *equal to the number of malicious links in the network, and*

$$\theta = (1 - T_{dr})^d - \frac{\beta}{N}. \tag{5.8}$$

where N *is derived from Lemma 14*

In Theorem 16, we analyze the protocol overhead with the following three metrics (we further analyze the *throughput* and *latency* in Section 5.7 via real-field testing):

1) The ***communication overhead*** is the fraction of extra packets each router needs to transmit.

2) The ***marking cost*** is the number of extra bits a source needs to embed into each data packet.

3) The ***per-path state*** is defined as the per-path extra bits that a router stores for the security protocol in *fast memory* needed for *per-packet* processing.[3]

Theorem 16. ***Overhead:*** *For each router, the communication overhead is one packet for each epoch of* N *data packets. The marking cost is* $k \cdot d$ *bits for the* k*-bit MACs where* d *is the path length. The per-path state comprises one* $\lg N$*-bit* C^{good} *counter, one* $\lg \beta$*-bit* C^{bad} *counter, one* $\lg N$*-bit last-seen per-path* SN*, and one epoch symmetric key.*

5.6 SSFNet-based Evaluation

In addition to analyzing the theoretical performance, we implement ShortMAC prototype on the SSFNet simulator [6] to study the detection delay and security of ShortMAC. Section 5.7 further investigates ShortMAC's throughput and latency. These experimental results provide *average-case* performance with various attack strategies to complement the theoretical results derived in the *worst case* scenario (due to multiple mathematical relaxations such as Hoeffding inequality) and constant dropping/injection rates.

[3]The buffering space needed for the Onion-ACK construction of `report` messages in ShortMAC is not a major concern, as the Onion-ACK is computed only once every epoch, which can be buffered in off-chip storage.

Evaluation scenario and attack pattern. Since ShortMAC provides a natural isolation across paths due to its per-path state, our evaluation focuses on a single path. Specifically, we present the result of a 6-hop path (routers f_1, f_2, f_3, f_4, f_5 and the destination f_6) since our experiment yields the same observation with other path lengths. We simulate both an (i) *independent packet corruption* pattern where a malicious node drops/injects each packet independently with a certain drop/injection rate, and (ii) *random-period packet corruption* pattern where the benign (non-attack) period T_b and attack period T_a (when the malicious node drops/modifies *all* legitimate packets) are activated in turns. The durations for both periods are randomly generated. For both attack patterns, we control the *average* packet drop/injection rates and observe that both attack patterns yield similar observations. Hence, in the following experiment, we only show the results for the independent packet corruption pattern. Also, we infuse natural packet loss rate ρ for each link to simulate natural packet loss, which is not provided by SSFNet. As Section 5.4 elaborates ShortMAC security against colluding attacks, we only show the representative results for a single malicious node f_3. For each simulation setting, we run the simulation 1000 times and present the average results.

Against various dropping attacks. Figure 5.4 depicts the detection delay N and error rates δ with per-link natural loss rate ρ as 0.5%, drop detection threshold T_{dr} as 1%, and a stealthy malicious drop rate as 2%.

We see that: (i) even against stealthy dropping attacks with a dropping rate as low as 2%, ShortMAC can successfully localize a faulty link in < 2000 packets with an error rate $\delta < 1\%$, which is orders of magnitudes faster than the worst-case theoretical bound (Lemma14). (ii) In addition, the FN rate is always no lower than the FP rate, because when a FP occurs (a benign link being falsely detected) the actual faulty link must have evaded detection for the current epoch (ShortMAC detects only one "faulty" link each epoch). (iii) When N is large, the FP and FN rates are almost identical, because the two rates are different only when no faulty link is detected (false positive is 0 while false negative is non-zero), which is unlikely to happen when N is large.

Figure 5.5 depicts different detection delays with different natural packet loss rates, demonstrating that larger $|T_{dr} - \rho|$ yields higher detection accuracy and lower detection delay.

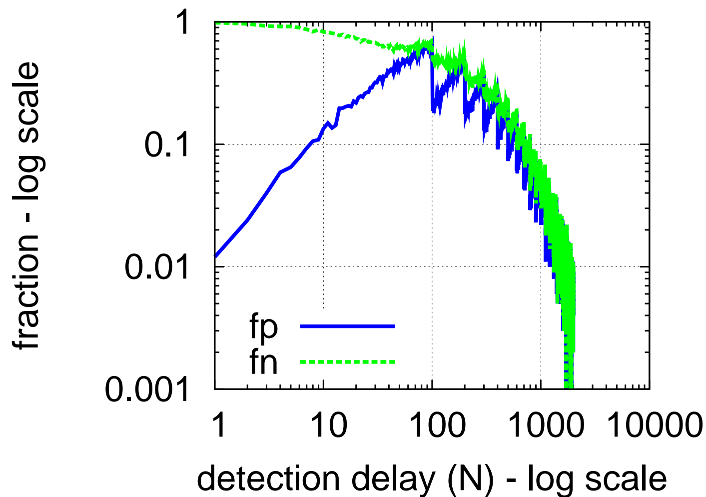


Figure 5.4: Natural loss. The malicious drop rate is 2%, $T_{dr} = 1\%$, and natural drop rate $\rho = 0.5\%$.

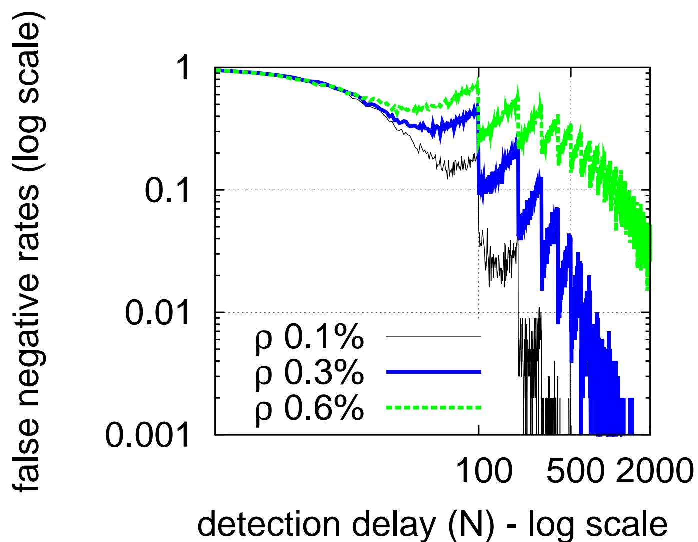


Figure 5.5: Dropping attacks. The malicious drop rate is 2%, and $T_{dr} = 1\%$.

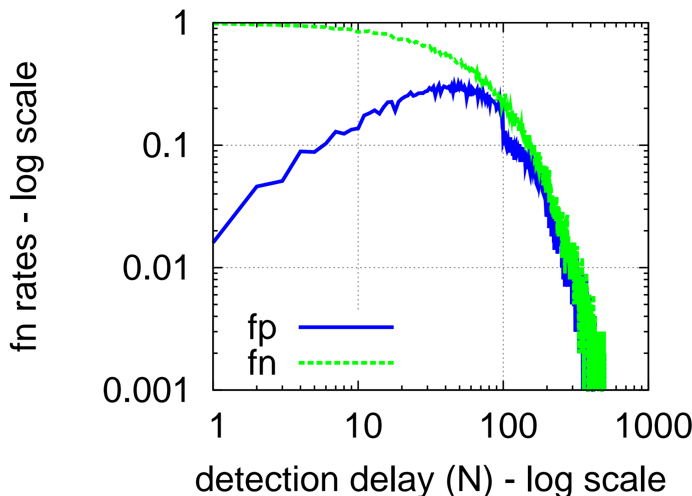


Figure 5.6: Injection attacks. The malicious injection rate is 2% using 2-bit MACs, natural loss rate $\rho = 0.5\%$, and $T_{dr} = 1\%$.

Against various injection attacks. Figure 5.6 shows the results when f_3 injects packets at a 2% rate (relative to the legitimate packet sending rate). It shows that the error rates stay below 1% in a few hundred packets, indicating that even with 2-bit MACs, an adversary can only inject up to around ten packets without being detected. We further investigate the effects of using different lengths of $k-$bit MACs, and Figure 5.7 shows that the detection delay and error rate dramatically diminish as k increases.

Against combined attacks. Figure 5.8 shows how the combinations of dropping and injection attack strategies (in our setting, dropping/injection rates are chosen between 2% – 5%) influence the protocol. We observe that the detection delay is mainly determined by the dropping detection process, which is much slower than the injection detection process. This also indicates that a malicious node cannot gain any advantage (and actually can only harm itself) by injecting bogus packets in attempt to bias the counter values.

Variance due to different malicious node positions. To investigate the influence of the position of the malicious node, we consider a path with 6 forwarding nodes $f_1, f_2, \ldots, f_6$ and place the malicious node at each position (1 to 6) in turn. We limit the error rate $< 1\%$ and obtain the

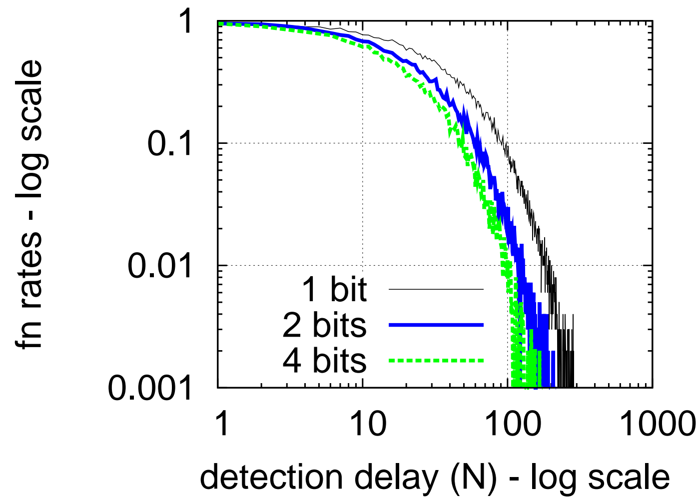


Figure 5.7: Effects of different k-bit MAC lengths on detection delay N and false negative rate δ. The malicious injection rate is 2%, $\rho = 0.5\%$, and $T_{dr} = 1\%$.

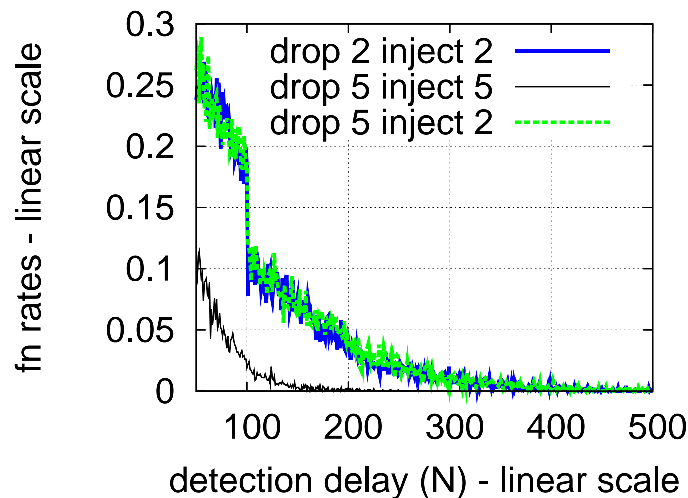


Figure 5.8: Combined attacks. "drop p inject q" denotes the use of p% dropping rate and q% injection rate at f_3.

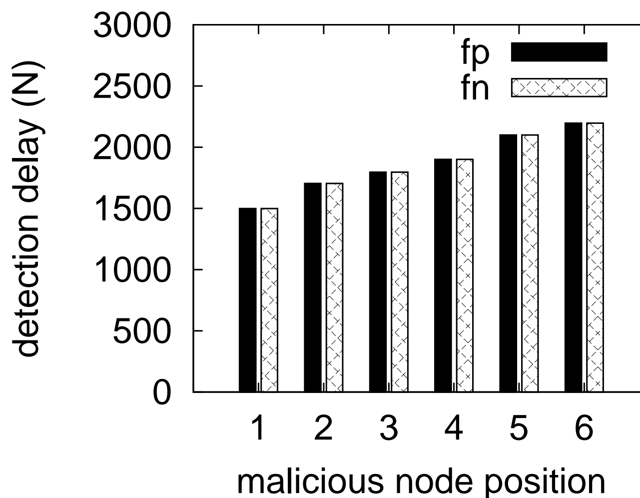


Figure 5.9: Variance on detection delay N in dropping attacks. $\delta < 1\%$, $T_{dr} = 1\%$, $\rho = 0.5\%$, and both malicious dropping and injection rates set to 5%.

corresponding detection delays. Figure 5.9 shows one representative scenario where both dropping and injection rates are 5%. We can see that (i) the dropping detection delay increases linearly when the malicious node is farther away from the source. This is because in the ShortMAC detection process, the source always inspects the closer links first and stops once the first "faulty" link is detected. The FP rate thus increases when more links exist between the source and the malicious node due to natural packet loss on each link. (ii) In contrast, the injection detection delay exhibits little variance (cannot be seen from the figure as the detection delay is determined by the dropping detection), which can also be theoretically proved.

Comparison with recently proposed protocols. For comparison, we simulate the full-ACK and PAAI-1 schemes presented in Chapter 4. Recall that full-ACK is a heavy-weight fault localization protocol requiring an Onion ACK packet from *every* forwarding node for *every* packet the source sent. In contrast, PAAI-1 employs packet sampling and only requires acknowledgments for the securely sampled packets to reduce communication overhead while retaining desired detection delay. Since both Full-ACK and PAAI-1 only consider packet dropping attacks, we compare their dropping detection delays along a path with 6 hops and f_3 as the malicious node. Figure 5.10

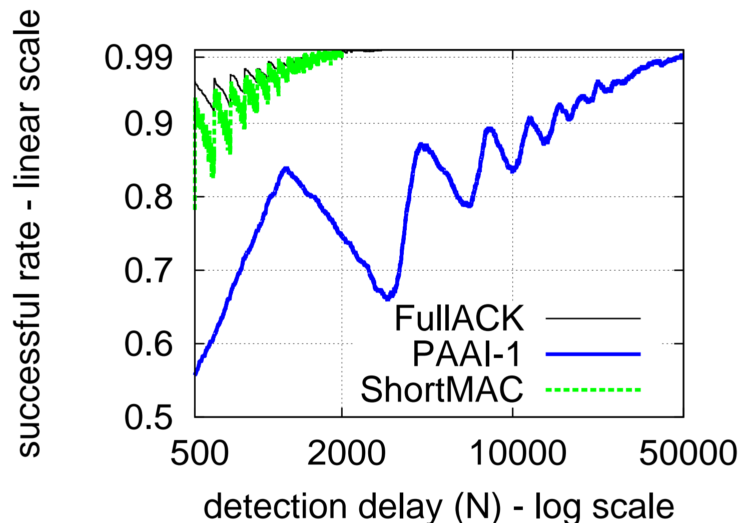


Figure 5.10: Comparison with PAAI-1 and Full-ACK. The natural packet loss rate $\rho = 0.5\%$ and drop detection threshold $T_{dr} = 1\%$.

	ShortMAC	Full-ACK	PAAI-1
Detect. delay	20 sec	20 sec	8.3 min
Communication	0.01%	100%	5.6%

Table 5.2: Comparison of ShortMAC, Full-ACK, and PAAI-1 with a source send rate of 100 packets per second.

shows the results when per-link natural packet loss rate $\rho = 0.5\%$ and drop detection threshold $T_{dr} = 1\%$. To make the comparison clear, we use a metric of *successful rate*, which equals to 1 - $max\{$FP rate, FN rate$\}$. The results show that the detection delays to achieve a successful rate $> 99\%$ for ShortMAC, Full-ACK, and PAAI-1 are 2000, 2000, and 5×10^4, respectively. Table 5.2 shows their detection delays in seconds/minutes and compares the extra communication overhead, based on the results from Figure 5.10 and with $\delta < 1\%$.

5.7 Linux Prototype and Evaluation

We implement ShortMAC source and destination nodes as user-space processes running on Ubuntu 10.04 32-bit Desktop OS. Even implemented in user-space on a standard desktop OS, our result

shows that the cryptographic operations of ShortMAC incur little communication degradation and negligible additional latency at gigabit line rate. It has also been demonstrated that using modern hardware implementation and acceleration the speed of PRF functions can be fundamentally improved [52].

Implementation details. Our ShortMAC processes listen to application packets via TUN/TAP virtual interfaces and appending k-bit MACs to the packets. We also implement ShortMAC routers using the Click Modular Router [51] running on Ubuntu 10.04 32-bit Desktop OS, which verify the k-bit MACs in each packet at each hop. To approach the realistic performance of commercial-grade routers, we implement the above elements on off-the-shelf servers with an Intel Xeon E5640 CPU (four 2.66 GHz cores with 5.86 GT/s QuickPath Interconnect, 256KB L1 cache, 1MB L2 cache, 12MB L3 cache, and 25.6 GB/s memory bandwidth) and 12G DDR3 RAM. The servers are equipped with Broadcom NetXtreme II BCM5709 Gigabit Ethernet Interface Cards.

Evaluation methodology. We evaluate ShortMAC's effects on communication throughput and computational overhead, especially due to the generation and verification of k-bit MAC using PRF operations. We utilize the widely used Netperf benchmark [4] for the ShortMAC throughput evaluation, and write our own micro-benchmark for accurate latency evaluation. We evaluate ShortMAC with varying packet sizes by configuring the interface Maximum Transmission Unit (MTU) sizes. We evaluate the throughput of a ShortMAC router and a ShortMAC source separately to better illustrate the throughput of each component, while the end-to-end path throughput can be easily derived by taking the minimum throughput of the two evaluation results. Then we evaluate the end-to-end latency with different path lengths ranging from 2 to 64. We also exploit the multi-core parallel processing at the source node via OpenMP API [5].

Summary of evaluation results. The evaluation results of our Linux software prototype demonstrate that both a ShortMAC router and source node can retain more than 92% of the *baseline throughput* (no ShortMAC operations are employed). Furthermore, the additional latency due to ShortMAC operations is negligible (tens of microseconds) even with a path length of 64 hops. The results further indicate the ShortMAC scheme is fully scalable as the number of processing cores

increases in a software-based implementation, while we anticipate hardware implementation of the MAC operations in ShortMAC can further boost the protocol throughput. Details of the evaluation results are as follows.

Router throughput with different PRF implementations. We first evaluate the throughput of a user-level ShortMAC router with different PRF implementations (i.e.,UMAC [85], HMAC-SHA1 [53], and AES-CMAC [83]) with the support of the new Intel AES-NI instructions [45]. The ShortMAC router connects a source machine and a destination machine, with the source sending TCP packets via Netperf as fast as possible to the destination to stress-test the router. For comparison, we use the Linux kernel forwarding throughput *without* ShortMAC operations as the base line. The ShortMAC router runs as a *single user-space* process without exploring parallelism, which already matches up the base line speed as shown below.

Figure 5.11 depicts the results with packet sizes from 100 to 1500 bytes, showing that UMAC-based PRF implementation yields the highest throughput, which retains more than 90% of the baseline throughput (e.g., 92% with 1.5KB packet size and 96% with 1KB packet size). With a small packet size of 100 bytes, both the baseline and ShortMAC throughput dropped substantially (similar to other public testing results [3]), because the network drivers used in our experiments are running under interrupt-driven mode, which hampers throughput when packet receiving rate is high. However, UMAC-based PRF still retains $\frac{53.84}{57.52}$=94% of the baseline throughput.

Source node throughput. We further evaluate the throughput of a ShortMAC source node with different path length d, where for each path length the source needs to perform $d-1$ UMAC-based PRF operations. Originally, it might seem that the ShortMAC source node represents the throughput bottleneck as the source needs to compute multiple k-bit MACs. However by parallelizing the ShortMAC operations on readily-available multi-processor systems, the throughput of a ShortMAC source node can *fully* cope with the base line rate even with a path length of 8. For comparison, we use the source node throughput *without* ShortMAC operations as the baseline. We evaluate two different parallelizations based on widely used OpenMP [5] API. Our first implementation (internal parallelism in short) uses multiple OpenMP threads to parallelize the computation

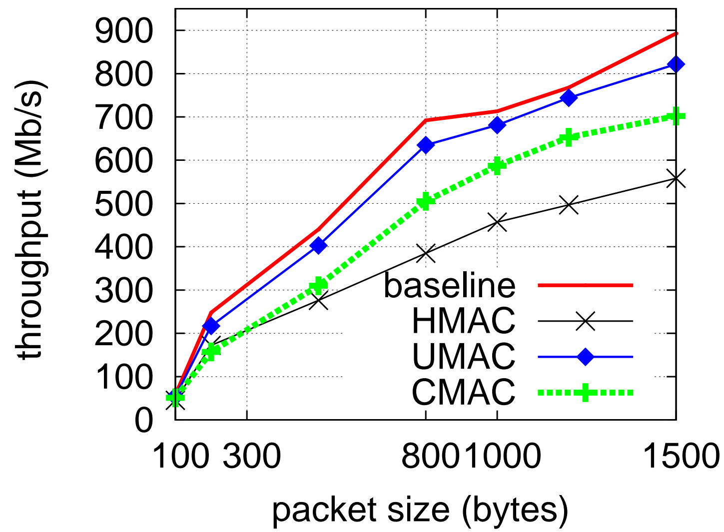


Figure 5.11: Router throughput.

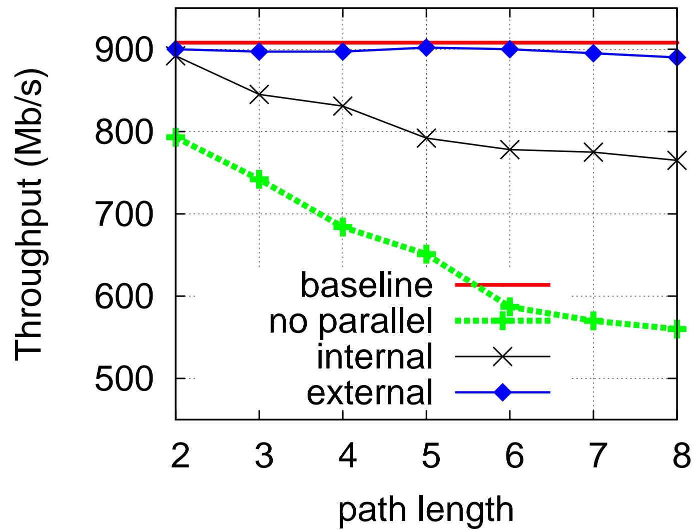


Figure 5.12: Source throughput.

of multiple k-bit MACs per packet. Our second implementation (external parallelism in short) assigns different packets to different OpenMP threads.

We evaluate the ShortMAC source throughput with various packet sizes, and observe that in all cases ShortMAC incurs negligible throughput degradation. Hence we only show the results with packet size set to 1500 bytes in Figure 5.12. We can see that external parallelism yields the best performance, which matches the baseline case where the source performs no ShortMAC operations.

ShortMAC latency. We also evaluate the additional latency incurred by a ShortMAC source node for computing the k-bit MACs with different path lengths and packet sizes; while the end-to-end latency can be derived base on our results. This additional latency in ShortMAC includes PRF computation, k-bit MACs appending, and TCP/IP checksum updating. We write our micro-benchmark to derive the additional time delay for the source to send each packet compared to the baseline case where the source does not compute any k-bit MAC nor updates the checksums.

Figure 5.13 and Table 5.3 show the results. We can see that the latency incurred by the checksum computation is stable. It does not increase with the packet size because in our implementation we employ incremental checksum update for the short MAC appended to the packet, instead of recomputing the checksum over the entire packet. We do not observe sharp increase of checksum latency with increasing path length either due to ShortMAC's efficient k-bit MAC authentication. In addition, the latency caused by the checksum computation is small compared to the latency introduced by UMAC-based PRF computation. The additional latency due to UMAC computation increases linearly to the path length under the same packet size, and also increases linearly to the packet size with a fixed path length due to the property of the UMAC algorithm. Finally, compared to the average end-to-end network latency which is on the order of milliseconds, the additional latency introduced by ShortMAC is negligible.

5.8 Discussion and Limitations

Incremental deployment. Although we argue it is feasible to upgrade all routers with ShortMAC within ISP/enterprise networks, we observe that partial deployment of ShortMAC can still

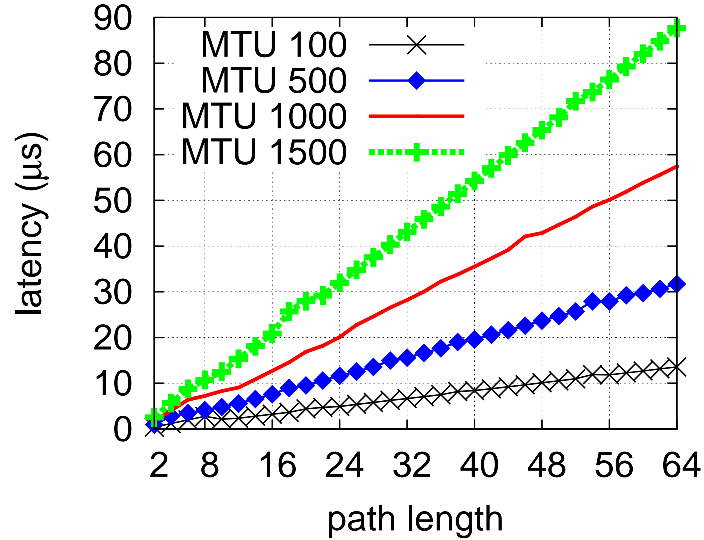


Figure 5.13: Source latency.

Path Length	Checksum (μs)	UMAC (μs)			
		100	500	1000	1500
2	0.0374	0.1771	0.4760	0.8892	1.4047
3	0.0378	0.3691	0.9557	1.7635	3.3025
4	0.0442	0.5239	1.4273	2.6357	4.0944
5	0.0415	0.7080	1.9018	3.5059	5.4566
6	0.0437	0.8723	2.3758	4.3839	6.8307
7	0.0445	1.0467	2.8530	5.2617	8.2019
8	0.0474	1.2206	3.3274	6.1285	9.5483

Table 5.3: ShortMAC source node latency breakdown (checksum updates and UMAC computation). All the data represent the average time of processing 50000 packets.

provide benefits and thus enables incremental deployment. Specifically, the ShortMAC routers form an *overlay network* on top of the physical network. In the overlay network, a “logical link” consists of the physical links between two ShortMAC routers. The fault localization protocol runs only on the ShortMAC routers and a data delivery fault will be localized to a logical link. Although in such settings the source node cannot exactly identify a faulty physical link, it can nevertheless localize the fault to a network area (a set of links between two ShortMAC routers) to facilitate further investigation. Furthermore, the more densely the ShortMAC routers are deployed, the more accurate the fault localization can be, which incentivizes incrementally deploying ShortMAC. However, one caveat for incremental deployment is that a discovery protocol for determining which routers support ShortMAC is needed, possibly through the use of explorer packets.

Interdomain deployment. Though ShortMAC mainly targets at intra-domain networks such as ISP and enterprise networks, ShortMAC may also be deployed in interdomain networks such as the Internet. In the interdomain setting, each Autonomous System (AS) can represent a node in ShortMAC; the fault localization runs at the AS level and localizes any data delivery fault between two ASes. To make ShortMAC applicable, different ASes need to establish secret keys (e.g., via Passport [60]), and the egress router of an AS needs to set the TTL value of each packet to the TTL value at the ingress router *minus one* to enable k-bit MAC verification (Section 5.3.1). Finally, a source AS needs to know the downstream AS path (which is readily available in BGP) which may dynamically change in the current Internet; however, the majority of AS paths are stable over minutes [78] thus facilitating ShortMAC fault localization. If an adversary were to constantly alter paths, it would essentially raise suspicion to itself, since path information is visible and the adversary needs to remain on the path to remain effective.

Topology changes and short-lived flows. Fault localization protocols inevitably require at least a threshold number of packets to be sent along the monitored path to obtain a statistically accurate detection in the presence of natural packet loss. Hence, monitored paths need to be stable over an epoch. Since ShortMAC incurs several orders of magnitude lower detection delay compared to related work [97, 21], ShortMAC can support topology or path changes and short-lived flows

much better than previous work. For example, as long as the path remains stable for transmitting around 2000 packets, the source can make an accurate fault localization. While path changes do happen during an epoch (e.g., due to link failures), the source will detect the old link where the path is switched away as faulty. At the same time, the source can also learn the routing updates about the path change, and by correlating the detection results with routing updates, the source may distinguish a benign path change and a malicious packet misrouting attack (in which case no corresponding routing updates will be received). However, the fault localization accuracy of ShortMAC decreases for dynamic paths that transmit far fewer than 2000 packets before path changes occur.

Multipath routing. A ShortMAC router maintains different counters for different paths, and need to know which counter to update given a certain packet (or which path the packet belongs to). If a source uses multiple paths simultaneously to reach a destination, the source and destination IDs alone are no longer sufficient to identify a path. Instead, the source needs to encode the path in the packets so that the routers know which counters to update. For example, in SCION routing [96], the source embeds the path into packet headers, which naturally supports ShortMAC.

5.9 Summary

In this chapter, we design, analyze, implement, and evaluate ShortMAC, an efficient path-based fault localization protocol, which enables a *theoretically proven* guarantee on data-plane packet delivery and substantially outperforms related protocols in the following aspects. First, ShortMAC achieves high security assurance even in the presence of strong adversaries in control of colluding malicious routers that can drop, modify, inject, and misroute packets at the forwarding paths; whereas a majority of existing fault localization protocols exhibit security vulnerabilities under such a strong adversary model. Second, compared to existing *secure* protocols, ShortMAC achieves several orders of magnitude lower detection delay and protocol overhead, which facilitates its practical deployment. Finally, we demonstrate that ShortMAC's efficient cryptographic operations, even if implemented in software, have negligible effects on the communication throughput via realistic

testing on Gigabit Ethernet links. We anticipate that ShortMAC probabilistic authentication and efficient fault localization can become a basic building blocks for the construction of highly secure and efficient network protocols.

The high efficiency of ShortMAC facilitates its practical deployment, and enables the construction of efficient secure routing protocols. We thus anticipate that ShortMAC can become a basic building block for the construction of highly secure and efficient network protocols. Though more efficient compared to PAAI, ShortMAC requires changes to the packet headers (for adding the k-bit MACs) while PAAI requires no changes to the packet headers. In addition, as a path-based protocol, ShortMAC still suffers several limitations as discussed and addressed in the next chapter.

Chapter 6

TrueNet

Though PAAI and ShortMAC strive to optimize the efficiency of fault localization, theoretically proven lower bounds have shown that path-based fault localization protocols in the *current* network infrastructure *inevitably* incur prohibitive overhead. We observe the current limits are due to a *lack of trust* relationships among network nodes. This chapter demonstrates that we can achieve much higher fault localization efficiency by leveraging trusted computing technology to design a *1-hop-based* fault localization protocol, TrueNet, with a small Trusted Computing Base (TCB). We also intend TrueNet to serve as a case study that demonstrates trusted computing's ability in yielding tangible and measurable benefits for secure network protocol designs.

6.1 Introduction

Barak et al. recently proved the lower bound overhead of path-based fault localization protocols in the *current* network infrastructure [21], which is impractical for large-scale ISP/enterprise/datacenter networks. Specifically, the lower bound states that a router *must* share some secret (e.g., cryptographic keys) with each source sending traffic traversing that router, making the key storage overhead at an intermediate router *linear in the number of end nodes*. In addition, path-based fault localization protocols run at the granularity of entire end-to-end paths, requiring each intermediate router to store *per-path state* and the paths to be long-lived (e.g., transmitting at least 10^6 packets, which would hinder agile load-balancing and traffic engineering) [21, 97]. These fundamen-

tal limitations exist in traditional network infrastructure due to the lack of *any trust relationships* among nodes. Hence, a source node needs to *directly* check or monitor all intermediate routers (thus sharing secret keys and state) in the routing path to ensure the routers behave correctly.

Furthermore, in existing secure fault localization protocols, a node n which detects a faulty link l can only remove l from n's *local* routing table but cannot share the detection result with other nodes, otherwise a potentially malicious n make false accusation of other benign links (*slander* attacks). This retards the *network-wide* detection/failure recovery process, and causes inconsistent routing tables at different nodes (faulty links excluded from the routing tables of some but not all nodes). Inconsistent routing tables violate the requirements of certain routing protocols such as link-state routing. The lack of trust among network nodes also inhibits the global sharing of local detection result.

In light of the fault localization limitations in current network infrastructures, we explore how trusted computing technology can enable a network architecture with intrinsic *trust of correct data delivery* among nodes with fundamentally better performance than the proven boundaries [21] in a traditional network architecture. Our key insight is that *remote code attestation* provided by trusted computing enables a node to verify if a remote communicating node runs a trusted (or expected) version of software/protocol via authenticated "code measurements". *Isolation* further ensures that critical code execution and data are isolated from all other code and devices on the local system. Jointly, these properties provide **transitivity of verification**, i.e.: *if A verifies B's code integrity (via attestation and isolation) and B verifies C, then A believes in C's code integrity as well without needing to verify C's code integrity, because A knows B's code has correctly verified C.* Transitivity of verification, when applied to secure network protocol designs, enables *each* node to perform verification and monitoring *only with 1-hop neighbors*, building a *chain* of verification over the *end-to-end* path with reduced overhead, i.e., only requiring *per-neighbor* (as opposed to per-node or per-path) state at each router. In short, transitivity of verification eliminates the need of establishing *direct point-to-point* validation between any two nodes in the network which incurs high storage overhead and obstructs key management.

Though useful, current trusted computing technologies are by no means a panacea when directly applied to the realm of computer networks. Although several researchers propose Trusted Platform

Module (TPM)-based protocols for securing general distributed systems (e.g., BIND [82]) and specific network applications (e.g., Not-a-Bot [40]), fundamental challenges render these approaches ineffective in securing data delivery at the network layer: (i) existing approaches cannot "attest" raw command-line configuration for which an expected "measurement" for remote attestation is hard to define, (ii) the extensive network stack would swell the size of the Trusted Computing Base (TCB) and it is challenging to abstract a small-sized, invariant "critical code", and (iii) a large ISP network can contain different routing instances with different implementation versions [57], which obstructs the use of a consistent "code measurement" for attestation.

The TrueNet design answers these challenges of applying trusted computing. Instead of strictly attesting the *semantics* of the huge, intertwined network stack itself, TrueNet attests the *behavior* of the network stack, i.e., whether it has successfully delivered the data or not. On one hand, the success of data delivery guarantees that *all* of the network-layer components have worked correctly, regardless of their implementation variations. On the other hand, if *any* of the network-layer components misbehaves, failures will arise in data delivery by which the faulty link(s) can be detected. Correspondingly, our approach in TrueNet is to monitor *1-hop* data delivery behavior (behavior of the network-layer protocol stack) with a small **monitoring module** as the critical code at each hop, and attest, isolate, and protect only the particular monitoring module with trusted computing. Thus, TrueNet requires only a small amount of critical code (the small monitoring module) as the TCB. Such a small TCB size (i) supports different network stack implementations and flexible protocol updates, (ii) makes the attestation of the small critical code efficient, and (iii) enables applying formal analysis [28] on the small critical code to ensure the TCB is indeed trustworthy.

The small TCB on each TrueNet router forms a *logical protected path* overlayed on the physical machines and an untrusted network stack between a source and destination, along which data delivery is monitored and ensured. As a result, TrueNet achieves efficient fault localization with **small router state** (only per-neighbor state), support for **dynamic/short-lived paths** (no requirements on the minimum number of packets transmitted along a path since monitoring is performed only between neighbors), and **global sharing** of detection results while eliminating slander attacks. As a proof of concept, we implement a TrueNet prototype in Linux using existing

trusted computing technology and a TPM, and demonstrate that TrueNet provides high throughput while achieving the desired security properties. We also launch real trace-based measurements to show that the router state in TrueNet is up to *five* orders of magnitude less than related work [21, 97].

Contributions. We design, implement, and evaluate TrueNet, which, assuming trusted hardware, achieves secure fault localization with properties (i.e., per-neighbor router state, dynamic path support, and global sharing of fault localization results while avoiding slander or framing attacks) that invalidate the previously proven performance boundaries in traditional networks [21]. TrueNet still provides benefits for partial adoption, enabling incremental deployment, and can be deployed in inter-domain settings with the recently proposed SCION architecture [96]. Finally, TrueNet explores the role trusted computing might play in securing network protocols, shows the possibility of using trusted computing to break traditional performance boundaries, and could spark future research.

6.2 Setting

Besides the problem formulation described in Chapter 2, we introduce additional assumptions and definitions for this chapter below.

Definition 17. *We denote by* $\delta_{AB} = \{\delta^d_{AB}, \delta^f_{AB}\}$ *the number of original packets dropped and misrouted* (δ^d_{AB})*,* and *the number of packets injected, modified, and reordered* (δ^f_{AB}) *on* l_{AB}*. A link* l_{AB} *is* faulty *if* δ_{AB} *is larger than a certain* accusation threshold $\{T_{dr}, T_{in}\}$ *set by the network administrator, i.e.:*

$$\delta^d_{AB} > T_{dr}, \quad \textbf{\textit{or}} \quad \delta^f_{AB} > T_{in}. \tag{6.1}$$

Definition 18. ***Aggregate fault localization*** *is achieved iff* given a routing path p, δ_{AB} *can be accurately learned for each link* l_{AB} *in* p*.* ***Per-packet fault localization*** *is achieved iff* given the routing path p *the failure of delivering a single packet in* p *can be immediately localized to a specific link in* p*.*

Adversary Model. We follow the trusted computing literature and assume the adversary can compromise the router OS, install malware on the routers, and launch remote software-based attacks; but the adversary cannot compromise hardware or manipulate the physical network infrastructure, nor defeat trusted computing primitives (code attestation and isolation). Such a remote attacker model is consistent with real-world router-based attacks. For example, most documented router compromises in ISP and enterprise networks are due to phishing [7] and remote exploitation of router software vulnerabilities [2, 13] and weak passwords [41] by remote hackers [87]. In addition, a majority of network operators in a recent security survey [1] listed *router misconfiguration*, which also falls under our software-based attack model, as an important cause of outages; and documented router software misconfiguration has led to network partitioning [55]. Finally, software-based attacks are usually more stealthy and large-scale than hardware-based attacks, since a hardware-based attacker usually needs physical proximity to targeted routers and will likely leave physical evidence, making the attack more auditable and less scalable.

The adversary controls multiple malicious routers which can drop, modify, inject, reorder, and misroute packets on links incident to malicious nodes in control. Furthermore, the adversary can launch *collusion* attacks where multiple malicious routers can coordinate and conspire to evade fault localization or incriminate a benign link. However, the adversary has polynomially bounded computational power and cannot break cryptographic primitives.

6.3 Fundamental Challenges

We further elaborate on the fundamental challenges in directly applying code attestation and isolation to secure data delivery in large-scale networks.

Large protocol stack. The network layer contains numerous interacting software components, i.e., (i) topology discovery, (ii) path selection from the topology, (iii) converting routing tables to forwarding tables, (iv) forwarding table lookup, etc. The incorrect operation of *any* of these components will hamper the correctness of the eventual network data delivery; therefore, straightforward attestation of the entire protocol stack would require attesting tens of thousands of lines of code.

For example, the IPv4 subsystem in the Linux 2.6.37 kernel contains more than 66K lines of code, and the IP-related elements in the Click modular router [51] contain more than 15K lines of code. This swells the TCB size and thus broadens the surface for potential vulnerabilities.

Diverse implementations and complex dependencies. In practice, there can be many co-existing protocol implementations and instances [57] within the same large ISP or enterprise network. Furthermore, due to the intrinsic and obscure interactions between network-layer components, it is highly challenging to distill an invariant, small, infrequently updated critical code as TCB to be attested.

Securing raw user input/configuration. In addition to the network protocol stack, data delivery also depends on human command-line input and configuration. Unfortunately, user configurations are hard to attest due to the flexibility of the configuration language, but can be utilized by the attackers to launch attacks to sabotage data delivery. Since the current Cisco IOS provides rich command-line interfaces to drop and alter packets, an attacker can cause damage without even modifying the network stack.

Hence in this paper, we strive to address these challenges by ascertaining *the minimal, invariant critical code for securing network data-plane packet delivery*, along with its minimal configuration parameters.

6.4 Design Building Blocks

Remote attestation, isolation, and sealed storage are the high-level primitives that trusted computing offers pertaining to our purpose of securing network data delivery.

Trusted computing primitives. By remotely *attesting* a selected piece of "critical code", a node X can verify if a remote node Y is executing the expected, correct version of the critical code. In conjunction with *isolation*, attestation can ensure that the execution of the critical code occurs untampered by any potentially present malicious code including the OS. Specifically, with attestation of the 1-hop monitoring module as the critical code in TrueNet, a node X can convince

another node Y that X is indeed executing the correct monitoring module in an isolated fashion. Furthermore, sealed storage binds a piece of sensitive data to a particular piece of software, ensuring that only the software that originally sealed the data accesses it. In TrueNet, sealed storage can seal a monitoring module's secret keys so that only the *same* monitoring module can access the secrets.

These trusted computing primitives have been widely deployed on commodity computers [39, 44]. In the remainder of the paper, we first use these trusted computing primitives *conceptually* for presenting the TrueNet protocol. Then we delineate and implement a TrueNet router architecture incorporating the trusted computing primitives in Sections 6.10 and 6.11.

Security properties. Remote attestation and sealed storage can be used to set up *secure channels* and *transitivity of monitoring results* as the security properties leveraged by TrueNet for efficiently achieving fault localization.

1) Secure channel: The above trusted computing primitives enable a monitoring module MM_A to generate and convey its public key to a remote MM_B [70], based on which MM_A and MM_B can establish a shared secret key. By performing cryptographic operations using the secret keys *sealed* and only known by the trusted monitoring modules at network routers, a compromised router OS or malware cannot impersonate the monitoring module by forging signatures or performing encryption/decryption based on those sealed keys. This builds a secure communication channel among the monitoring modules at different routers.

2) Transitivity of monitoring results: End-to-end monitoring can now be achieved via a chain of 1-hop monitoring between every two adjacent neighbors while eliminating slander and collusion attacks. This is because if a node X verifies via code attestation that its neighbor Y is executing the correct monitoring module MM_Y, X knows that the monitoring results reported by MM_Y are correct, and that MM_Y is correctly monitoring Y's neighbor, which recursively ensures the entire end-to-end path is being correctly monitored.

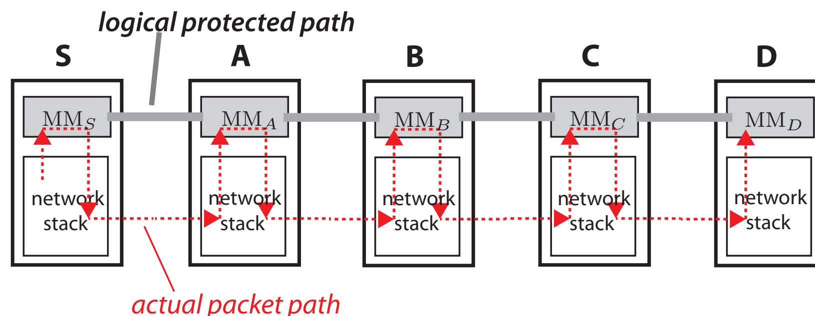


Figure 6.1: An example topology to illustrate the operation of TrueNet. The solid line represents the logical protected path of packets implemented by the secure channels between the trusted monitoring modules.

6.5 TrueNet Overview

We give an overview of TrueNet with Figure 6.1 as an example topology. The shaded areas denote the monitoring modules isolated and protected by trusted computing at each router and thus reside in the TCB. *A router's network stack (including the OS, network interfaces, and other related programs) is untrusted.*

The logical protected path. In TrueNet, each packet is *supposed* to pass through the monitoring module MM_i at each hop i. The MMs on the logical path are protected by trusted computing mechanisms and are thus trusted. The dashed line in Figure 6.1 depicts the **actual packet path** comprising the physical machines and network stack, originated from node S and destined to D. In contrast, the *secure channels* between adjacent trusted monitoring modules along the actual packet path form a **logical protected path** overlayed on the untrusted network stack. Every two neighboring MM_A and MM_B on the logical protected path share a secret key K_{AB} that is sealed by and only accessible to the same MM_A or MM_B. Nodes (i.e., monitoring modules) in the logical protected path can thus communicate with *secrecy* and *authenticity* using the shared and sealed secret keys, and the untrusted network stack cannot inject or forge authenticated messages in the logical protected path. Nodes in the logical protected path can also attest to each other that the MMs are indeed intact and trusted.

The formation of this logical protected path requires only *per-neighbor* key storage yet greatly

facilitates secure fault localization. Specifically, each MM_i maintains a local data structure (e.g., a counter) to reflect the reception of each packet as the "packet footprint". In this way, each packet should leave a certain footprint at each hop's monitoring module *iff* the packet is successfully delivered along the logical protected path. Later by comparing the packet footprints left at every two neighbors MM_A and MM_B in the logical protected path, it either confirms that the packets have been successfully delivered (if the footprints match) or some problem occurs between MM_A and MM_B (if the footprints do not match). The secrecy and authenticity properties of the logical protected path ensure that the footprints reported by each MM_i will not be forged or injected by a malicious network stack or malware.

Localizing a faulty *link*. Note that TrueNet detects a faulty *link* between two adjacent MMs, instead of a specific malicious router. In this way, MMs do not rely on the untrusted network stack or NIC to correctly deliver packets to the MMs: if the NIC or network stack of a router M drops or modifies packets before sending to MM_M, faults will be localized between MM_M and its neighboring MMs. For example in Figure 6.1, if the malicious OS or a malware in router A corrupts or drops the packet before it reaches MM_A, then the footprint that packet leaves at MM_A will differ from that at MM_S, thus causing link l_{SA} to be detected as we show shortly.

Small TCB. The TCB in TrueNet only includes the trusted computing primitives and the protected MM. Due to the challenges outlined in Section 6.3, it is impractical to include the entire network stack and NIC in the TCB or for code attestation. Due to those challenges, one *cannot* simply use attestation to determine if the local OS or NIC is compromised and stop any malicious system.

TrueNet fault localization phases. From a high level, TrueNet consists of setup, 1-hop monitoring, and global accusation phases as sketched below.

1) Setup: During protocol setup, an administration entity of the network installs a public/private key pair, a public key K_{admin} of the administration entity, and a neighbor list to each node. Every two neighbors A and B establish a shared secret key K_{AB}, which is used to authenticate the messages exchanged between MM_A and MM_B in the logical protected path. The administration

entity signs the neighbor list along with a version number using its private key K^{-1}_{admin}. The node private key, MAC key and K_{AB} are sealed by and only accessible to the local monitoring module.

2) 1-hop monitoring: To implement the secure channel between neighboring MMs in the logical path, a MM_A computes a Message Authentication Code (MAC) for each packet sent to the next-hop MM_B in the logical protected path using K_{AB}. By verifying the MAC, MM_B can be convinced that its neighbor A is running the correct monitoring module *otherwise K_{AB} cannot be retrieved for authentic MAC generation*. Similarly, by authenticating the footprint reports, a node can be convinced that its neighbors are telling the correct footprints *and* having correctly monitored their neighbors in the logical protected path, *otherwise the sealed key cannot be retrieved for authenticating the reports*. This *chain of 1-hop monitoring* ensures all links in a logical protected path have been correctly monitored.

TrueNet provides two types of 1-hop monitoring primitives in the monitoring modules, namely, *per-packet monitoring* and *aggregate monitoring* for achieving per-packet fault localization and aggregate fault localization, respectively. These two monitoring approaches differ in the footprint data structure and how frequently footprints are compared between neighbors. In per-packet monitoring, a monitoring module MM_B maintains an identifier (e.g., a sequence number) for each received packet with a correct MAC computed by MM_A, and sends back an acknowledgment (ACK) to MM_A for each received packet from MM_A *immediately*. In aggregate monitoring in contrast, MM_B increments a *counter* if a packet received from the neighbor MM_A contains a correct MAC computed by MM_A. Then MM_B exchanges the counters with its neighbor across the logical protected paths *periodically*. Hence, aggregate monitoring reduces the communication overhead and tells *how many* packets have been dropped or corrupted between every two neighbors in the logical protected paths, while per-packet monitoring provides more fine-grained and immediate information about *which packets* have been corrupted between two neighbors in the logical protected paths, enabling instant *failure recovery* (e.g., by immediately retransmitting the corrupted packets at the *network layer* on a per-link basis). In both monitoring approaches, MMs add additional *per-neighbor* sequence numbers for the data packets, which are used to prevent replay and reordering attacks and identify dropped packets.

3) Global accusation: A monitoring module MM_A constantly asks for the footprint reports from

each neighbor MM_B to learn δ_{AB}. If MM_A observes an abnormally large δ_{AB} on a link l_{AB} in the logical protected path, MM_A sends out an accusation message to its 1-hop neighbors in the logical protected path which can verify and accept the message based on authentic MACs. Similarly, the neighbors of MM_A in the logical protected path further tell their neighbors about the accusation. This process recursively achieves network-wide *trustworthy broadcasting* (Section 6.8). Hence, *all* the network nodes remove faulty links from their routing tables upon identification. Such consistency of routing tables further accelerates network-wide failure recovery, enabling the use of link-state routing which remains the de facto routing protocol for contemporary intra-domain networks.

Small router state and support for dynamic paths. Note that in any phase, attestation and authentication are only performed between two neighbors; thus each node only maintains *per-neighbor* state. Such 1-hop operations also eliminate the need for long-lived and stable paths, facilitating load balancing.

The following sections detail each phase of TrueNet.

6.6 TrueNet Setup

In the setup phase, a local network administrator remains responsible for setting up and updating a router with appropriate cryptographic keys and its neighbor list as follows.

Day Zero setup. The first time a router i physically joins a network, the network administrator (i) launches a monitoring module MM_i on router i and ensures that MM_i is securely loaded and protected by the trusted computing primitives on router i. (ii) The administrator installs a public key K_{admin} of the administration entity of the network into MM_i and ensures that MM_i has correctly loaded and protected K_{admin} for verifying future messages from the administrator. (iii) The administrator creates and installs a public/private key pair K_i/K_i^{-1} and a neighbor list NL_i for router i, along with a version number and a signature created using its private key K_{admin}^{-1}. The private key K_i^{-1} is sealed and only accessible to MM_i. (iv) Each router i exchanges a secret key K_{ij} with each of its neighbors j using their public/private key pairs [70]. K_{ij} is sealed and only

	Source				**Router A**	
S1)	$\mathrm{OS}_S \rightarrow \mathrm{MM}_S$:	packet m				
S2)	MM_S `genPkt`:	$\mathcal{M}_S \leftarrow m, N_{SA}^S, \mathrm{MAC}_{K_{SA}}(m\|\|S\|\|N_{SA}^S)$				
S3)	MM_S `awaitACK`:	store N_{SA}^S, start timer				
S4)	MM_S `incrSN`:	$N_{SA}^S \leftarrow N_{SA}^S + 1$				
S5)	$\mathrm{MM}_S \rightarrow \mathrm{OS}_S$:	$\mathcal{M}_S$	$\overset{\mathcal{M}_S}{\Longrightarrow}$	A1)	$\mathrm{OS}_A \rightarrow \mathrm{MM}_A$	$\mathcal{M}_S$
				A2)	MM_A `validatePkt`:	if $\mathcal{M}_S$ invalid, accuse l_{SA}
				A3)	MM_A `genACK`:	$ack_{AS} \leftarrow A, N_{SA}^A, \mathrm{MAC}_{K_{SA}}(A\|\|N_{SA}^A)$
				A4)	MM_A `incrSN`:	$N_{SA}^A \leftarrow N_{SA}^S$
S6)	$\mathrm{OS}_S \rightarrow \mathrm{MM}_S$:	ack_{AS}	$\overset{ack_{AS}}{\Longleftarrow}$	A5)	$\mathrm{MM}_A \rightarrow \mathrm{OS}_A$:	ack_{AS}
S7)	MM_S `verifyACK`:	if ack_{AS} invalid, accuse l_{SA}		A6)	MM_A `updatePkt`:	$\mathcal{M}_A \leftarrow m, N_{AB}^A, \mathrm{MAC}_{K_{AB}}(m\|\|A\|\|N_{AB}^A)$
				A7)	MM_A `awaitACK`:	store N_{AB}^A, start timer
				A8)	MM_A `incrSN`:	$N_{AB}^A \leftarrow N_{AB}^A + 1$
				A9)	$\mathrm{MM}_A \rightarrow \mathrm{OS}_A$:	$\mathcal{M}_A \Longrightarrow$ further sent to router B

Table 6.1: TrueNet per-packet monitoring. Shaded instructions are functions of the monitoring module MM_i which is in the TCB. $\mathrm{MAC}_K(m)$ denotes a Message Authentication Code (MAC) computed over m using the symmetric key K.

accessible to MM_i and MM_j, and is used for constructing the secure channel between MM_i and MM_j.

Incremental updates. After Day Zero setup, the administration entity uses the public key K_{admin} to authenticate all its update messages to the routers (e.g., when updating NL_i or K_i). These control messages from the administration entity will be protected by per-packet monitoring as we describe below. The MMs run at routers are responsible for verifying the authenticity of these updates messages using K_{admin}. The neighboring nodes i and j can periodically update their shared secret key K_{ij}. However, this paper omits the details of handling these updates due to space limitation.

6.7 TrueNet 1-Hop Monitoring

Given an end-to-end communication path p, 1-hop monitoring in TrueNet ensures that the data sent by the source will be correctly delivered to the destination along p, otherwise a faulty link in p that tampers with correct data delivery will be localized and accused. Thus, we assume the source node can learn path p (e.g., from link-state routing, source routing, or recent centralized routing protocols like 4D [37], SANE [27] or ETHANE [26]), which is a common requirement for all existing secure fault localization schemes. We first detail each of per-packet and aggregate monitoring, and then discuss their usage scenarios in Section 6.7.3.

6.7.1 Per-packet Monitoring

We use Figure 6.1 as an example to illustrate TrueNet per-packet monitoring. Table 6.1 shows the interactions between the source S and the first hop router A for transmitting and protecting a single packet. Subsequent routers in path p will perform identical operations as router A.

Packet generation. Upon receiving a packet m with path p embedded from the network stack (OS_S) of the source S, the trusted monitoring module MM_S wraps the packet into $\mathcal{M}_S$ with a *per-neighbor* sequence number N^S_{SA} for the next-hop router A, and a MAC computed over m and N^S_{SA} with the secret key K_{SA} shared between MM_S and MM_A (Table 6.1 S2). Meanwhile, router A

maintains a per-link sequence number N_{SA}^{A} remembering the last sequence number for the packets sent from S to A. Note that only one MAC for the next hop is attached (as opposed to attaching one MAC for each router in the path), because the transitivity of verification provided by trusted computing enables the chaining of trusted 1-hop verifications to achieve end-to-end guarantees.

As it transmits the packet, MM_S starts a timer, expecting to receive an ACK from the next-hop receiver MM_A within the allocated time, allowing MM_S to determine whether MM_A successfully received the packet. For this purpose, N_{SA}^{S} is temporarily stored as the packet identifier until the timer expires (Table 6.1 S3). MM_S then increments N_{SA}^{S} for the next packet to be sent to prevent packet replay and reordering attacks (Table 6.1 S4), and sends $\mathcal{M}_S$ back to OS_S, which in turn forwards $\mathcal{M}_S$ to router A.

Packet reception. Each received packet is expected to be passed through the monitoring module at each hop. At router A, MM_A first validates the received packet $\mathcal{M}_S$ via `validatePkt` (Table 6.1 A2), which includes checking the sequence number, the next hop, and the MAC as follows:

1) `validatePkt` first checks if the per-neighbor sequence number N_{SA}^{S} contained in $\mathcal{M}_S$ matches the locally stored per-neighbor N_{SA}^{A} value. If the values differ, indicating a replay, re-ordering, or packet injection, `validatePkt` terminates (skipping the following checks) and returns "invalid".

2) `validatePkt` then retrieves the next hop from path p embedded in $\mathcal{M}_S$, and checks if the local router A is indeed the next hop in p for the current communication flow. An inconsistency indicates the previous router's OS used a wrong interface (packet misrouted), and `validatePkt` terminates returning "invalid".

3) `validatePkt` finally checks the MAC in $\mathcal{M}_S$, and returns "invalid" if the MAC is incorrect.

If `validatePkt` outputs "valid", MM_A generates an ACK including N_{SA}^{A} as the packet identifier with a MAC (Table 6.1 A3), which MM_S awaits. MM_A then increments the local per-neighbor sequence number N_{SA}^{A} (Table 6.1 A4) to prevent packet replay and reordering attacks. If `validatePkt` returns "invalid", MM_A believes that forwarding misbehavior occurs between MM_S and MM_A (denoted by l_{SA}). MM_A generates an accusation if the failure rate remains high with efficient trustworthy broadcasting (Section 6.8), or signals MM_S in the ACK for instant failure recovery as we show shortly.

Packet forwarding. If the packet validation succeeds, the original MAC embedded in the received packet $\mathcal{M}_S$ is replaced with a new one computed for the next hop MM_B using the sealed secret key K_{AB} shared between MM_A and MM_B; and the per-neighbor sequence number is also replaced with the one (N_{AB}^A) for traffic between MM_A and MM_B (Table 6.1 A6). Right before the updated packet $\mathcal{M}_A$ departs MM_A, MM_A also starts a timer and expects an authenticated ACK from the next-hop MM_B (Table 6.1 A7). Finally, MM_A increments the per-neighbor sequence number N_{AB}^A for the next-hop B to prevent packet replay and reordering attacks (Table 6.1 A8).

ACK reception and failure recovery. Upon receiving an ACK ack_{AS} from a neighbor router A (Table 6.1 S6), MM_S checks if the corresponding packet identifier (N_{SA}^S in this case) is still stored indicating the timer has not expired. Then MM_S checks if the MAC is correct. If any check fails, MM_S can either re-transmit the particular corrupted packet up to r times for *instant failure recovery*, or globally accuses l_{SA} for failing to deliver any of the $r+1$ packets corresponding to N_{SA}^S via trustworthy broadcasting. The number of re-transmissions r is introduced and set to tolerate spontaneous packet loss. E.g., assuming an upper bound ρ (probability) of packet loss rate and an upper bound ϵ of allowed false positive rate, we should set $r \geq \frac{\ln \epsilon}{\ln \rho} - 1$.

Optimization. Similar to the TCP acknowledgment mechanism, a sender MM can send data packets *asynchronously* to the ACKs within a certain *sliding window* of w packets, before the ACKs for previous packets have been received. Accordingly, a receiver node can send *one single* ACK for all the w packets in the previous sliding window to reduce communication overhead.

6.7.2 Aggregate Monitoring

In aggregate monitoring, packet forwarding at each hop is divided into consecutive **monitoring intervals**, which are *asynchronous* among network nodes. A monitoring interval **from** A **to** B refers to the aggregate monitoring for packets *sent from A to B* in that interval.

Different from per-packet monitoring where MM_S starts a timer and expects an *immediate* ACK from MM_A for *each packet* sent from MM_S to MM_A, in aggregate monitoring, MM_S increments a local *monitoring counter* C_{SA}^S for each packet sent to A. Our key observation is that, due to

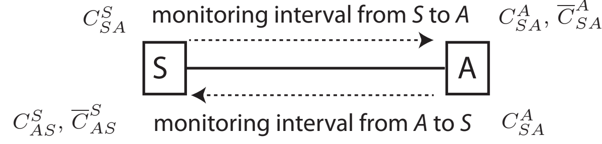


Figure 6.2: Router state in TrueNet aggregate monitoring: three counters for each neighbor.

packet authentication by 1-hop MACs, *packet count becomes a verifiable measure of the packet payload as well*, because a modified packet payload will result in an invalid MAC and cause the packet to be dropped without polluting the counter. Correspondingly, MM_A also increments a local monitoring counter C^A_{SA} for each *valid* packet received from MM_S; and increments another per-neighbor counter $\overline{C}^A_{SA}$ for each *invalid* packet received from A, as Figure 6.2 depicts. These counters can later be compared to reflect $\delta_{SA} = \{\delta^d_{SA}, \delta^f_{SA}\}$, i.e.:

$$\delta^d_{SA} = |C^S_{SA} - C^A_{SA}|, \;\; \delta^f_{SA} = \overline{C}^A_{SA} \tag{6.2}$$

Similarly, MM_A sets a counter C^A_{AB} for the next hop B, and this process recursively builds a trusted chain of 1-hop aggregate monitoring over the entire end-to-end path, while each node only has per-neighbor state (monitoring counters).

Periodically, neighbors exchange local monitoring counters in a "request-and-reply" manner to learn δ_{AB} for each link l_{AB} and accuse any link with δ_{AB} larger than a pre-set accusation threshold. Specifically, each monitoring interval consists of sending N packets (e.g., 10^4 packets). MM_S counts the number of packets sent in each monitoring interval I from S to A. Each time N packets have been sent indicates the end of interval I, and MM_S generates a counter request $\mathcal{R}_{SA}$ including the requester S, the next-hop requestee A, the interval number I to prevent replay attacks, and a MAC computed for the next hop MM_A. Then similar to per-packet monitoring, MM_S stores I and C^S_{SA}, starts a timer to wait for the counter report from MM_A, increments the interval number I, and zeros C^S_{SA} for the next interval. Finally, the request $\mathcal{R}_{SA}$ and the report $\mathcal{A}_{SA}$ proceed in the same way as in per-packet monitoring. Based on the received $\mathcal{A}_{SA}$, MM_S can calculate δ_{SA} (Equation 6.2) and accuse a faulty link if any.

6.7.3 Per-Packet vs. Aggregate Monitoring

Per-packet monitoring enables instant fault localization and failure recovery by re-transmitting the corrupted packets immediately, at the cost of an additional ACK per packet (or per w packets in a sliding window) on each link. Aggregate monitoring reduces the communication overhead by sending one counter report for all the packets in each monitoring interval (with N packets), at the cost of additional fault localization delay (one monitoring interval).

In TrueNet, per-packet monitoring is used to protect critical *control-plane messages*, e.g., the router configuration messages from the network administrator to each router as we mentioned earlier, global accusation message via trustworthy broadcasting as we show in Section 6.8, or flow setup packets in TCP. Accordingly, aggregate monitoring would be used to protect *line-rate* data packets for the sake of lower overhead, and the network can rely on *transport layer* protocols (such as TCP) for retransmitting and recovering the lost or corrupted packets on an end-to-end basis.

6.8 TrueNet Trustworthy Broadcasting

TrueNet trustworthy broadcasting achieves *reachability*, *integrity*, and *trustworthiness* of the broadcasted message. Specifically, when a certain node O broadcasts a certain message m, (i) every node in the network will receive the message as long as the malicious nodes do not cause a *graph partition* in the network topology (**reachability**), (ii) the broadcast message received by each node is the same as the original one (**integrity**), (iii) and the broadcast message is trusted, e.g., the accused link is indeed faulty (**trustworthiness**).

TrueNet trustworthy broadcasting is built on top of per-packet monitoring to achieve the above security properties. When a node O originates a broadcast message m, it uses per-packet monitoring (Table 6.1) to convince O's neighbors that the message has not been modified from the original one thus preserving integrity, and the message is generated by the correct monitoring module thus preserving trustworthiness. Figure 6.3 shows an example of how a broadcast message propagates using per-hop monitoring (not showing the ACKs). The per-packet authenticated ACK in per-packet monitoring assures a sender that its neighbors have received the correct message thus achieving reachability, also run the correct monitoring modules, and thus will faithfully keep broadcasting

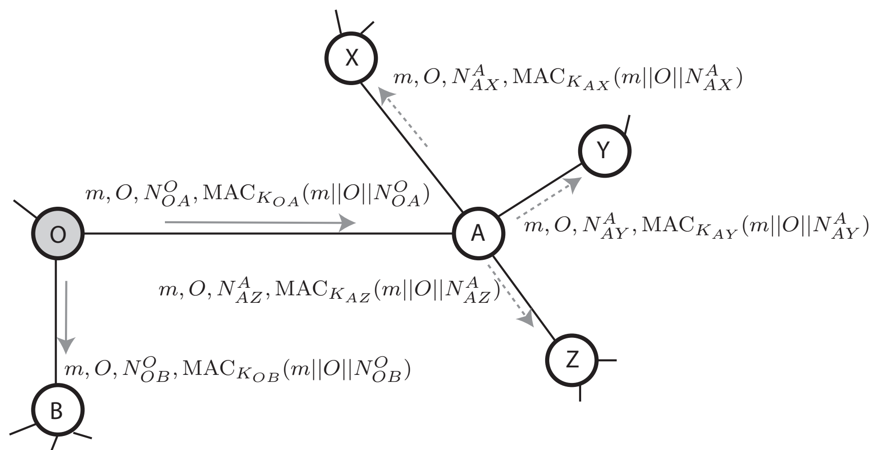


Figure 6.3: TrueNet trustworthy broadcasting example. Node O is the originator of the broadcast message and other nodes use per-packet monitoring to protect the broadcast message.

the message to their neighbors and so on.

Duplicate suppression. Numerous methods exist to ensure that the broadcast message traverses each link only a single time. Due to limited space we defer detailed protocol design and analysis to future work. However, a simple method for suppressing duplicate broadcast messages is for each MM to keep state to detect duplicate messages it may later receive. To recover the state, messages can contain time stamps and nodes can be loosely time synchronized, thus only requiring storage for the maximum clock skew plus the maximum duration for the message to reach all nodes.

Global accusation. Once a node's MM detects faults, the MM generates an accusation and disseminates it *inside* certain network-wide, *periodic* beacon messages, such as the periodical routing updates (or link state announcements in link state routing) or keep-alive messages between neighbors. In TrueNet, each router R's MM_R expects to receive every neighbor's beacon after every t seconds, otherwise MM_R accuses its neighbor which does not send a beacon on time (hence a malicious router OS cannot prevent the locally generated accusations from being sent to its neighbors). A beacon from a neighbor MM_N contains any accusation generated by MM_N and is protected using per-packet monitoring. If a beacon from MM_N contains an accusation, this beacon automatically becomes a broadcast message and is further propagated using the trustworthy broadcasting.

6.9 TrueNet Fault Localization Analysis

This section analyzes TrueNet fault localization delay, security and overhead, while Section 6.11 presents real-field implementation and evaluation.

6.9.1 Fault Localization Delay

The fault localization delay in per-packet monitoring equals the packet re-transmission time r: only when all $r+1$ packets fail to be delivered (failure recovery fails) will a link accusation be made. Theorem 19 states the lower bound of r. During aggregate monitoring, at the end of a monitoring interval, a router A can learn the *accurate* δ_{AB} for each local link l_{AB} (thus achieving aggregate fault localization), *regardless* of the interval length N (number of packets sent in that interval). Hence, the value of N is set based on the desirable tradeoff between detection delay and communication overhead. For example, a smaller N enables faster detection but increases the number of counter reports (one report required for every N packets). Furthermore, since faulty links are defined and detected based on the accusation threshold, the value of N is also determined by the accuracy of the threshold-based faulty link accusation. Specifically, a too small N will introduce considerable noise in the observed link loss rate, given by $\frac{\delta_{AB}^d}{N}$, due to the existence of spontaneous packet loss. Theorem 19 states the lower bound of N for achieving a sufficiently high accusation accuracy.

Theorem 19. *Suppose the natural packet drop rate is ρ on a link, the accusation threshold $T_{dr} = \rho + \epsilon$ where $T_{dr} \in (0,1)$*[1]*, and the allowed false positive and negative rate is σ. Then the fault localization delay or packet re-transmission time for failure recovery in per-packet monitoring is at least $r = \frac{\ln \sigma}{\ln \rho} - 1$. The fault localization delay or a monitoring interval length is at least $N = \frac{\ln(\frac{2}{\sigma})}{(T_{dr}-\rho)^2}$.*

Proof. We assume each link has a natural drop rate ρ.

Per-packet monitoring. The probability that a benign link "naturally" drops all $r+1$ packets (including r re-transmissions), or the false positive fp, is given by $fp = \rho^{r+1}$. Since we require $fp \leq \sigma$, we have $r \geq \frac{\ln \sigma}{\ln \rho} - 1$.

[1]To simplify the mathematical formula, we denote T_{dr} as a *fraction* of packets dropped, instead of the absolute number of dropped packets as the original T^d denotes.

Aggregate monitoring. We study how many packet transmissions are required to estimate the drop rate of a single link l_{ij} within a certain *accuracy interval*. Suppose that the true value of the drop rate of l_{ij} is θ^*_{ij}, and the estimated drop rate of l_{ij} is θ_{ij}. We compute the number of packets needed to achieve a (ϵ, σ)-accuracy for θ_{ij}:

$$Pr(|\theta_{ij} - \theta^*_{ij}| > \epsilon) < \sigma \tag{6.3}$$

i.e., with probability $1 - \sigma$ the estimated θ_{ij} is within $(\theta^*_{ij} - \epsilon, \theta^*_{ij} + \epsilon)$. We define each time a data packet is sent over link l_{ij} as a random trial, and thus each monitoring interval has N random trials. Then using *Hoeffding's inequality*, we have:

$$Pr(|\theta_{ij} - \theta^*_{ij}| > \epsilon) < 2e^{-2N\epsilon^2} \tag{6.4}$$

Then by Equation 6.3, we have:

$$2e^{-2N\epsilon^2} \leq \sigma \Rightarrow N \geq \frac{\ln(\frac{2}{\sigma})}{2\epsilon^2} \tag{6.5}$$

Since $\epsilon = T_d - \rho$, we further have: $N \geq \frac{\ln(\frac{2}{\sigma})}{2(T_d - \rho)^2}$ □

Finally, the *network-wide* faulty link detection process is accelerated in TrueNet since a faulty link detected by one node will be removed from the routing tables of all other nodes; whereas in existing protocols a node cannot share others' accusation because of slander attacks.

6.9.2 Security analysis

TrueNet achieves per-packet and aggregate fault localization via per-packet and aggregate monitoring, respectively. Recall that the adversary can drop, modify, inject, replay, re-order, and misroute packets at links under control.

Per-packet fault localization. Packet dropping, modification, and injection attacks between MM_A and MM_B will cause MM_A or MM_B to fail to generate authentic ACKs for the original packets; thus the link l_{AB} that corrupts the packets will be localized. Packet replay and re-ordering

attacks from MM_A to MM_B will cause packets to be dropped at MM_B thanks to the use of per-neighbor sequence numbers, because MM_B stores and only expects a packet with the *most recent* per-neighbor sequence number. Finally, packet misrouting attacks are impossible because the source embeds the expected path p in the packets, and routers will perform next-hop checking based on the path and will drop any packets that are misrouted.

Aggregate fault localization. Without loss of generality, we consider a monitoring interval from A to B for example. Upon receiving the counters C_{AB}^{B} and $\overline{C}_{AB}^{B}$ from B (otherwise MM_A can immediately accuse l_{AB} for not sending a correct counter report), MM_A can first be convinced that the counter values were reported by the correctly running MM_B and *are thus correct*. Then MM_A can estimate δ_{AB} and detect any fault. Similar to the analysis of per-packet fault localization above, packet dropping will increase δ_{AB}^{d}, and packet modification, injection, replay, re-ordering, and misrouting will increase δ_{AB}^{f}.

We give one interesting note about packet misrouting attack using Figure 6.1 as an example topology. The malicious node B can first misroute the packets to a colluding neighbor C' (not shown in the figure), which then *transparently* forwards the packet back to C (the legitimate next hop of B in path p) *without passing the packet through* $\mathit{MM}_{C'}$. TrueNet treats this as a legitimate case which does *not* violate aggregate fault localization, because *in the logical protected path* the packets still traverse from MM_B to MM_C in order. This packet detouring is only possible between *colluding neighbors* which can be treated as *one logical* malicious entity, and is akin to detouring packets inside the same malicious router.

6.9.3 Overhead Analysis

Storage overhead. We focus on the router state required for *per-packet* processing which needs to reside in on-chip memory or cache and usually becomes the system scalability bottleneck. A router state in TrueNet includes (i) per-neighbor secret keys (e.g., 16 bytes per neighbor) for both per-packet and aggregate monitoring, and (ii) *three* monitoring counters (e.g., 3×8 bytes) in aggregate monitoring as Figure 6.2 shows. Since per-packet monitoring is used for infrequent

(compared to the link rate) packets, such state can be either stored in the adequate off-chip DRAM, or stored in a small cache (storing up to w packets in a sliding window at any time).

Communication overhead. The extra communication overhead in TrueNet per-packet monitoring includes one ACK per packet or per sliding window with w packets. The communication overhead in aggregate monitoring is one counter report per monitoring interval (e.g., with 10^4 packets). When per-packet monitoring is only used for protecting infrequent (compared to the line rate) control messages such as flow setup in TCP and link-state routing updates, the extra communication overhead amortized on each data packet is small.

6.10 TrueNet Router Architecture

We present a TrueNet router architecture leveraging a *dedicated hypervisor* and TPM chip to implement the trusted computing primitives (remote attestation, isolation, and sealed storage), and modern mainstream router hardware to speed up time-critical operations in TrueNet.

Anatomy of a TrueNet router. Modern routers commonly use a switch-based router architecture with fully distributed processors [20] and the network interfaces perform almost all the critical data-path operations for a normal packet. Figure 6.4 shows the architecture of a TrueNet router, where the shaded components are those added in a TrueNet router but not present in a standard modern router and also constitute the TCB for TrueNet. As Figure 6.4 shows, each TrueNet router is equipped with a TPM chip and CPUs with hardware virtualization support (e.g., AMD SVM [10], or Intel TXT [44]), and installs a dedicated hypervisor such as TrustVisor [69]. The dedicated hypervisor isolates MM from the rest of the router system (e.g., router OS, peripheral devices, etc.), enables remote attestation and sealed storage with the support of TPM chip, and protects MM's execution integrity, data integrity and secrecy. Similar to TrustVisor [69], the TPM operations are only needed when the dedicated hypervisor boots to ensure the hypervisor's integrity, while afterwards the dedicated hypervisor performs attestation and storage sealing to improve the efficiency.

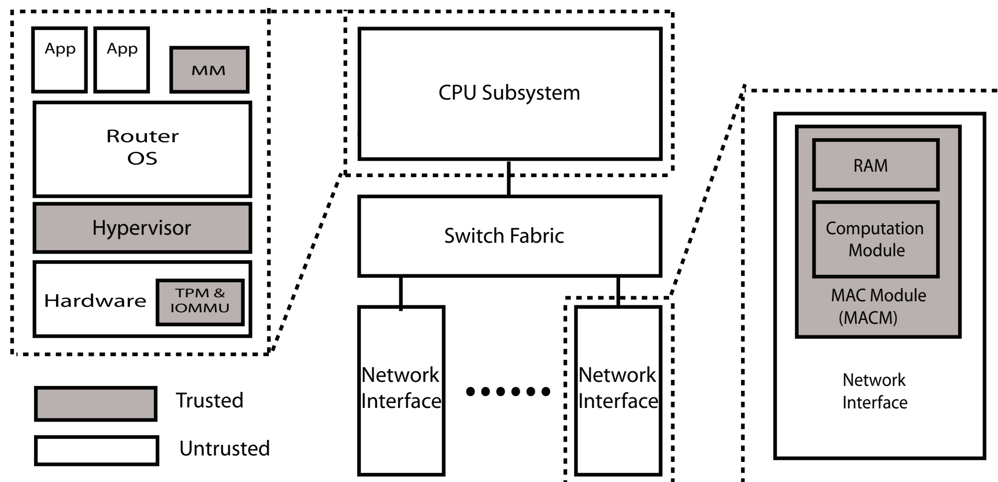


Figure 6.4: TrueNet router architecture.

For better performance, we anticipate on every network interface, there is a *trusted* hardware MAC Module (MACM) to perform the MAC operations in MM as described earlier. A MACM has a piece of private memory space and a high-speed MAC computation module. The private memory of MACM is mapped to the main memory residing in the CPU subsystem, and shared with the local MM. The dedicated hypervisor also protects this piece of main memory from the rest of the CPU subsystem, so that only the MM can read from and write to this main memory region. However, MACM can also be implemented inside the software MM as we described earlier, which we used for our prototyping (Section 6.11.1).

Software monitoring module MM. A MM handles all *control-plane* operations that are not time-critical, or infrequent in a TrueNet system. First, the local MM negotiates secret keys with the MMs on the neighboring routers, and writes the secret keys into the main memory region that maps the private memory of MACM. Secret key negotiation only happens periodically according to the cryptographic key lifetime. Secondly, MMs on the source nodes also handle packets originating from the connected end-hosts by adding the entire routing path into the packets (Section 6.7) for 1-hop monitoring. Thirdly, the MM is also responsible for generating accusations to be broadcasted in the beacon messages. In addition, MM also periodically checks the locally stored timers for awaiting ACKs from neighbors to detect and remove any expired entries.

Dedicated MAC module. The dedicated MAC module (MACM) is responsible for all data-plane operations to achieve high packet processing throughput. A MACM verifies the MAC in the packet, validates the correct presence of the local router in the embedded path, computes the new MAC using the shared secret key for the next hop router, updates per-neighbor sequence numbers and monitoring counters, and attaches the new MAC to the packet on a per-packet basis. To achieve high throughput in MAC computation, We can use parallelizable MAC algorithms such as XOR-MAC [22], XECB-MAC [34], PMAC [23], or high speed hardware implementations [93, 81, 62, 86] which can obtain more than 62.6 Gbps throughput.

6.11 Implementation and Evaluation

In this section, we evaluate both TrueNet's *computational* overhead based on our Linux prototype of a TrueNet router and TrueNet's *storage* overhead based on real-world ISP topologies and traffic traces. We show that even when implementing MACM *inside the software MM*, a TrueNet router can achieve gigabit line rate with only commodity multi-core support, and the state in a TrueNet router is up to five orders of magnitude less than in related work [97, 21].

6.11.1 Prototype and Computational Overhead

We implement a TrueNet router prototype in Linux with TPM chip to evaluate per-packet cryptographic computational overhead of a TrueNet router. We show the performance of a TrueNet intermediate router which performs two MAC operations per packet (verification of the previous-hop MAC and generation of the next-hop MAC) *inside the software MM*. We observe that the TrueNet per-packet cryptographic operations, even implemented in TrueNet software module MM without any hardware acceleration, can fully cope with gigabit link-rate processing of data packets, and are fully scalable to higher performance with more CPUs. We anticipate the dedicated hardware MACM (Section 6.10) can further boost the TrueNet router throughput.

Platform. We performed all experiments on off-the-shelf servers with one Intel Xeon E5640 CPU (four 2.66 GHz cores, 256KB L1 cache, 1MB L2 cache, 12MB L3 cache), 12G DDR3 RAM with

25.6 GB/s memory bandwidth. This CPU supports new Intel AES-NI instructions [45] for high speed AES computation. The servers are equipped with TPM chips and Broadcom NetXtreme II BCM5709 Gigabit Ethernet Interface Cards, and runs Ubuntu 10.04 32-bit Desktop OS.

Prototype. In our TrueNet prototype, we modify TrustVisor [69] as our dedicated hypervisor. We run Ubuntu Linux OS on top of our hypervisor and implement a TrueNet intermediate router as a multi-threaded user-space process. A TrueNet router process includes the secure software module MM and untrusted network stack. The untrusted network stack consists of two threads: a receiver thread that listens to network packets via TUN/TAP virtual interfaces and puts received packets to an input packet queue, and a forwarder thread in charge of sending the packets in the output packet queue to their appropriate next-hop routers. Multiple MMs run as child threads, constantly poll the input packet queue, copy the new incoming packets to a shared output packet queue, and perform MAC computations. We use the CMAC-AES-128 MAC algorithm to leverage the new AES-NI instructions on Intel CPUs.

Our software module MM performs similar per-packet cryptographic operations as the hardware module MACM proposed in Section 6.10 in software manner, while maintaining same security guarantees. The MM child threads are running inside the secure and isolated execution environment provided by dedicated hypervisor ever since threads start. The dedicated hypervisor also protects the memory region of input packet queue as accessible by both the untrusted network stack and MMs, and the output packet queue as writable by MMs but only readable by untrusted network stack. This memory configuration assures MM's execution integrity. Finally, the TPM securely boots and late-launches the dedicated hypervisor to guarantee its integrity, as described in the TrustVisor proposal [69].

Throughput and Latency Breakdown. We tested the throughput of our software TrueNet router prototype using the widely adopted network performance benchmarking tool Netperf [4]. Figure 6.5 shows the test result. The baseline performance in the figure is obtained by using a main thread to receive packets, *two* MM threads to move packets to the output packet queue *without any other operations*, and one forwarder thread to send packets out to next-hop routers. For

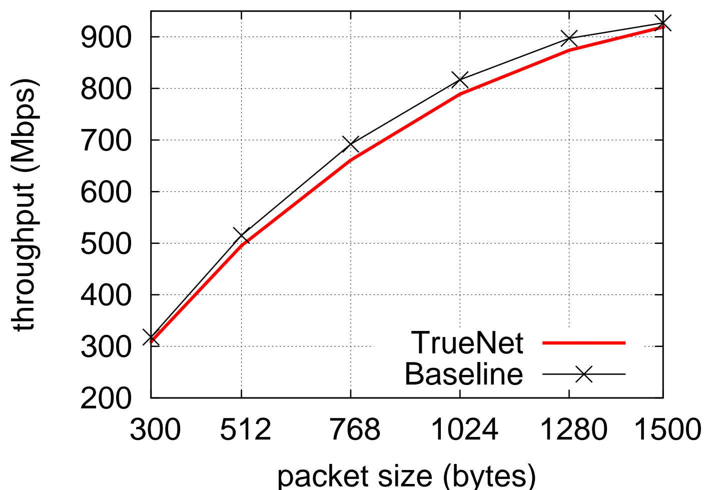


Figure 6.5: TrueNet router throughput.

	Packet Size (Byte)			
	1500	1000	500	100
MAC Computation	4.2	2.9	1.6	0.4
Others	1.3	1.1	0.7	1.1
Total	5.5	4	2.3	1.5

Table 6.2: TrueNet software module MM's latency overhead breakdown. All the data is the average time (microseconds) in 50000 packet processing trials.

TrueNet prototype, the test setting is similar to baseline performance test with the only difference that MM threads perform TrueNet packet validation and MAC computations for every packet. As Figure 6.5 shows, TrueNet prototype incurs *negligible* throughput degradation when compared with the baseline throughput (maximum degradation in our test is (817-789)/817=4.5% when packet size is 1024 bytes, most degradation rates are under 2%).

We also shows a latency overhead breakdown of executing software module MM's per-packet process. From Table 6.2, we know that, leveraging the new AES-NI instruction, MAC computations are highly efficient (on average 3 CPU cycles per packet byte). In our prototype, AES key setup time is negligible since each TrueNet router only needs to hold one session key per neighboring router in a session key life time, and we can pre-compute all AES sub-keys.

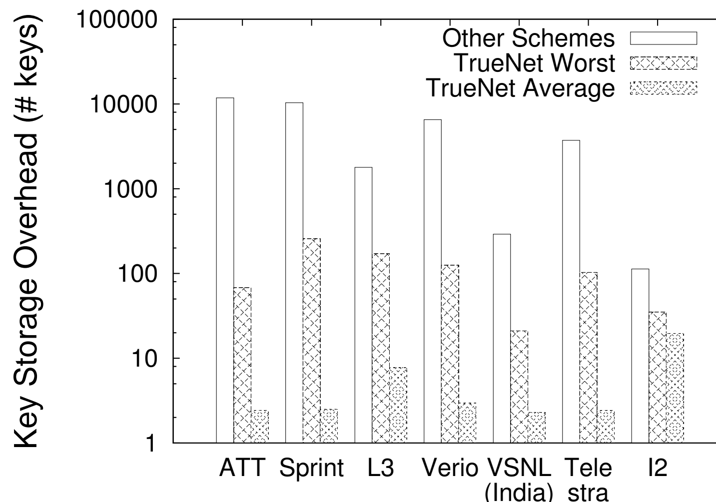


Figure 6.6: Key storage overhead of a single router on ISP topologies.

6.11.2 Storage Overhead Measurement

TrueNet's ability to deliver strong security properties (instant failure recovery with per-packet fault localization, global accusation, etc) with less state than previous attempts [21, 97] follows logically. Still, measurements under real-world conditions provide an exact assessment of TrueNet's strength.

Rocketfuel-based measurements. The Rocketfuel topologies [84] of various top-tier ISPs extend from the ISPs' peering routers to approximately the first hop within a customer's network. We count the node degree for each router in the topology to assess TrueNet's overhead and compare it to the number of nodes in the network, representing the recently proposed Statistical fault localization [21] and PAAI [97]'s key storage overhead. Figure 6.6 suggests that TrueNet incurs on average two orders of magnitude less overhead in the worst case (considering the maximum node degree in the topologies), and three order less overhead for the average case (considering the average node degree).

Internet2-based measurements. The Internet2 provides similar topology data for its core routers, which Figure 6.6 also illustrates (labeled as "I2"). Since this topology only includes core routers, TrueNet does not deliver the orders of magnitude less overhead achieved with the Rocketfuel

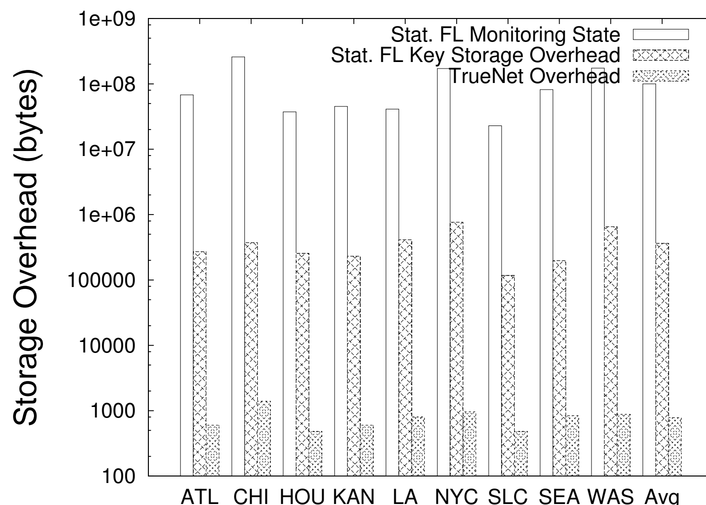


Figure 6.7: Overhead comparison based on Internet2 topology and traffic traces.

topologies, providing an 83% savings in the average case and 69% in the worst case. Conveniently, the Internet2 also provides Netflow data, allowing for measurement of TrueNet's and Statistical fault localization's monitoring state overhead. These Netflow files capture 1/100 packets seen over a five minute interval. In Statistical fault localization, the router incurs an around 500-byte "secure sketch" [36] for each path (identified as each unique source and destination in our measurement). In contrast, a TrueNet router maintains three counters (24 bytes) for each neighbor. Figure 6.7 shows that TrueNet requires approximately five orders of magnitude less monitoring state overhead. Additionally, these flow data allow for a more accurate estimation of key storage overhead in Statistical fault localization (number of sources with traffic concurrently traversing the same router), also shown in Figure 6.7 (the key storage overhead in TrueNet is still one key per neighbor).

6.12 Discussion

6.12.1 Incremental Deployment

Although we argue it is feasible to upgrade all routers with trusted computing primitives within *a single* administrative domain, we note that partial deployment of TrueNet can still benefit the early

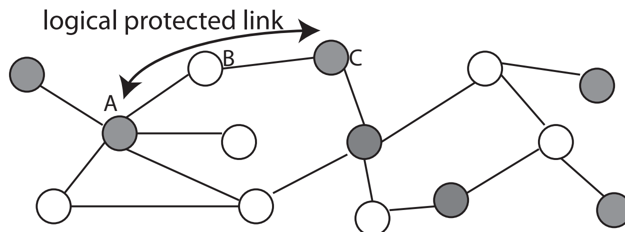


Figure 6.8: Incremental deployment of TrueNet. The shaded nodes have deployed TrueNet and form logical trust links between each other.

adopters. Specifically, when only a subset of routers in a network are equipped with TrueNet, the monitoring modules still constitute logical protected paths where a *logical protected link* between two MMs may consist of multiple physical links. Figure 6.8 shows an example where the shaded nodes have deployed TrueNet and a logical protected link consists of l_{AB} and l_{BC}. Hence, fault localization is still achieved on each logical protected link (though not an exact physical link), which helps localizing the failure to a bounded region and facilitates network diagnosis. Furthermore, the more densely the MMs are deployed, the more accurate the failure localization can be, which incents incrementally deploying TrueNet.

6.12.2 Interdomain Deployment

TrueNet mainly targets intra-domain networks such as ISP and enterprise networks, where sophisticated hardware attacks can be precluded since the *remote* attacker (the adversary model we considered) does not have physical access to the routers. However, it is ineffective to deploy TrueNet in the *current* inter-domain setting where each Autonomous System (AS) represents a node in TrueNet, because a selfish or malicious AS has physical access to its routers and can thus subvert the hardware (e.g., TPM chips) upon which trusted computing primitives rely. Fortunately, the recently proposed SCION [96] inter-domain architecture groups the ASes into different *trust domains*, within which strong contractual or legislative regulation can be enforced. Hence, an AS tampering with the hardware can be legally penalized by the containing trust domain. This architecture naturally enables the wide deployment of TrueNet (or trusted computing primitives in general) across different ASes within a trust domain. Meanwhile, TrueNet also serves as an example of how to technically achieve enforceable accountability within a trust domain in SCION.

6.13 Summary

In this chapter, we demonstrate that trusted computing enables transitivity of verification and eliminates the need of establishing *direct point-to-point* trust between any two nodes in the network which incurs high storage overhead and obstructs key management. TrueNet employs only a small TCB to achieve secure fault localization with small router state, dynamic path support, and global accusation that are proven impossible in traditional networks. Though achieving much smaller protocol overhead compared to path-based fault localization approaches (PAAI and ShortMAC), TrueNet requires special hardware support (such as TPM chips and hardware virtualization) and is vulnerable to hardware attacks. In the next chapter, we present a 1-hop-based fault localization protocol with small overhead without relying on trusted computing.

Chapter 7

DynaFL

Like PAAI and ShortMAC, most existing *secure* fault localization protocols are *path-based*, which assume that the source node *knows the entire outgoing path* that delivers the source node's packets and that the path is *static* and *long-lived*. However, these assumptions are incompatible with the dynamic traffic patterns and agile load balancing commonly seen in modern networks. To cope with real-world routing dynamics, we propose the first secure *neighborhood-based* fault localization protocol, DynaFL, with *no* requirements on path durability or the source node knowing the outgoing paths. DynaFL aims to localize data-plane faults to a 1-hop neighborhood, instead of a specific link. Unlike TrueNet, DynaFL requires no special hardware support. Through a core technique we named *delayed function disclosure* , DynaFL incurs little communication overhead and a small, constant router state independent of the network size or the number of flows traversing a router. In addition, each DynaFL router maintains only a single secret key, which is 100 to 10000 times fewer than in path-based fault localization protocols based on our measurement results.

7.1 Introduction

Existing fault localization protocols that are secure against sophisticated packet dropping and modification attacks [17, 97, 21] require that the sender *know the entire path* that delivers the source node's packets, and that the path be *long-lived* (e.g., stable over transmitting 10^8 packets [21]) to obtain a statistically accurate fault localization. However, recent measurement studies [16, 38, 31]

show that a considerable fraction of current network flows are short-lived "mice" and routing paths are highly dynamic. Furthermore, emerging enterprise and datacenter networks call for more agile load balancing and dynamic routing paths. For example, a recently proposed datacenter routing architecture, VL2 [38], employs Valiant Load Balancing to spread traffic uniformly across network paths via random packet deflection. In this case, the actual routing path is determined *on the fly during forwarding* and thus cannot be predicted and known by the sender. Given the conflict between the "static-path" assumption and the "dynamic-path" reality, researchers have concluded that *existing* fault localization protocols are impractical for widespread deployment in large-scale networks with dynamic traffic patterns [21].

In addition, in existing secure fault localization protocols, a router must share some secret (e.g., cryptographic keys) with each source node sending traffic traversing that router, making the key storage overhead at an intermediate router linear in the number of end nodes. The proliferation of key copies shared by routers with all end nodes under non-uniform (and generally poor) administration also increases the risk of key compromise thereby enabling undetected attacks. In existing secure fault localization protocols, a router also needs to maintain per-path state for each path traversing that router, making the fault localization unscalable for large-scale networks.

We aim to bridge the current gap between the security of fault localization against strong adversaries and the ability to support dynamic traffic patterns in modern networks such as ISP, enterprise, and datacenter networks. More specifically, the desired fault localization protocol should be secure against sophisticated packet dropping, modification, fabrication, and delaying attacks by colluding routers, while retaining the following properties:

- **Path obliviousness:** A source node or a router does not need to know the outgoing/downstream path.
- **Volatile path support:** The fault localization protocol requires no duration time for a forwarding path.
- **Constant router state:** A router does not need to maintain per-path, per-flow, or per-source state.

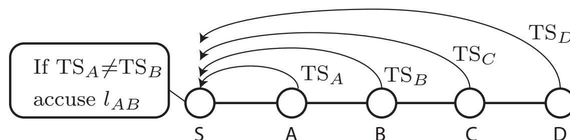


Figure 7.1: Path-based fault localization. TS$_r$ denotes the traffic summary generated by router r. For brevity, "TS$_A$≠TS$_B$" refers to "TS$_A$ deviates from TS$_B$ more than a certain threshold".

- **O(1) key storage:** A router only manages a small number of keys regardless of the network size.

Path obliviousness and volatile path support together enable agile (e.g., *packet*-level) load balancing and dynamic routing paths (e.g., Valiant load-balanced paths). These two properties also *decouple* the data-plane fault localization from routing, thus enabling it to support a wide array of routing protocols. Finally, constant router state provides scalability in large-scale networks and O(1) key storage reduces the security risk due to key compromise.

We observe that the "static-path" assumption in existing secure fault localization protocols stems from the fact that those fault localization protocols operate on entire end-to-end paths (*path-based*), to localize the fault to *one specific link*. As Figure 7.1 shows, each router maintains a certain "traffic summary" (e.g., a counter, packet hashes, etc.) for *each* path that traverses the router (thus requiring per-path state), and sends the traffic summary to the source node S of each path. S can then detect a link l as malicious if the traffic summaries from l's two adjacent nodes deviate greatly, as Figure 7.1 illustrates. Hence, S needs to know the entire path topology to compare traffic summaries of adjacent nodes, and needs to send a large number of packets over the same path so that the deviation in traffic summaries can reflect a statistically accurate estimation of link quality. Finally, to authenticate the communication between the source and each router in the path, a router needs to share a secret key with each source that sends traffic through that router.

In this paper, we explore *neighborhood-based* fault localization approaches, where a router r's data-plane faults (if any) can be detected by checking the consistency (or conservation) of the traffic summaries generated by the *1-hop neighbors* of r (denoted by $\mathbb{N}(r)$ in Figure 7.2). That

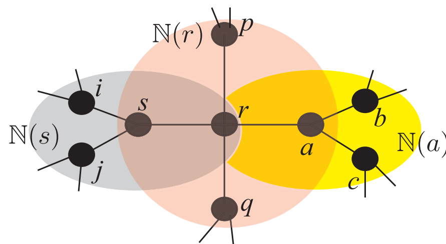


Figure 7.2: A neighborhood example.

is, in benign cases, the packets *sent to* r will be consistent with the packets *received from* r by all of r's neighbors as reflected in their traffic summaries. In this way, the fault localization is independent of routing paths and only depends on 1-hop neighborhoods, thus supporting arbitrary routing protocols and dynamic load balancing. Additionally, each router in a neighborhood-based approach only needs to maintain state for each neighbor. In summary, neighborhood-based fault localization localizes faults *to a specific 1-hop neighborhood* to reduce further investigation, to *trade localization precision for practicality* in modern networks with dynamic traffic patterns.

Though promising, neighborhood-based fault localization is susceptible to sophisticated packet modification and collusion attacks due to several security and scalability challenges. For example, for the sake of scalability, the traffic summary cannot be a copy of all the original packets (or even their hashes), but have to be a compact representation of the original packets via a certain **fingerprinting function** $\mathcal{F}$. On one hand, if $\mathcal{F}$ generates traffic summaries at different nodes *without* using different secret keys, a malicious router can predict the outputs of $\mathcal{F}$ at other nodes and tactically modify packets such that the outputs of $\mathcal{F}$ will stay the same as with the original packets. On the other hand, if $\mathcal{F}$ at different nodes uses *different* secret keys, we cannot compare and run consistency check over different nodes' traffic summaries. To address these challenges, we propose DynaFL, a protocol that employs a core technique called *delayed function disclosure* , which discloses the *same* key for computing $\mathcal{F}$ to different routers *after* they have forwarded the packets. To further minimize the protocol overhead, DynaFL employs a secure sampling mechanism also based on the delayed function disclosure, so that a malicious router cannot know if a packet is sampled or not at the time it forwards (corrupts) the packet. Finally, a router in DynaFL only shares a secret key with a centralized controller, thus achieving O(1) key storage.

Contributions. Our contributions are three-fold:

1. We raise the importance of pursuing a secure fault localization design to cope with *dynamic traffic patterns* in real-world operational networks with a small, constant router state and key storage.

2. To the best of our knowledge, DynaFL is the first secure *neighborhood*-based fault localization protocol that achieves path obliviousness and volatile path support, *and* is secure against both packet loss and sophisticated packet modification/injection attacks.

3. In addition, a DynaFL router requires only about 4MB per-neighbor state based on our AMS sketch [11] implementation, whereas path-based fault localization protocols require per-path state. We also show through measurements that the number of keys a router needs to manage in path-based fault localization protocols is 100 to 10000 times higher than that in DynaFL (which is a single key shared with a centralized controller). Finally, our simulation results demonstrate DynaFL's small detection delay and negligible communication overhead.

7.2 Setting

Besides the problem formulation described in Chapter 2, we introduce additional notation and definitions for this chapter below.

Notation. We denote the 1-hop neighborhood (or neighborhood, for brevity) of a node s as $\mathbb{N}(s)$, as Figure 7.2 illustrates. For a particular packet traversing a neighborhood $\mathbb{N}(s)$, the neighbor sending that packet to node s is called an **ingress node** in $\mathbb{N}(s)$ for that packet, and the node receiving that packet *from* s is called an **egress node**. We term a sequence of packets as a *packet stream* $\mathbb{S}$. Particularly, we denote the packet stream sent from node i to node j as $\mathbb{S}_{ij}$, and this packet stream is *seen* by nodes i and j as $\mathbb{S}_i^{\rightarrow j}$ and $\mathbb{S}_j^{\leftarrow i}$, respectively. The **difference** of two packet streams $\mathbb{S}$ and $\mathbb{S}'$, denoted by $\Delta(\mathbb{S}', \mathbb{S}')$, refers to the number of packets in one packet stream but not in the other, without considering the variant IP header fields such as the TTL and checksum.

Network Setting. We consider a network with dynamic traffic patterns and a relatively static network topology, which is best exemplified by today's ISP, enterprise, and datacenter networks. To provide maximum flexibility to support various routing protocols, and even packet-level load balancing, we pose *no* restriction on the routing protocols and load balancing mechanisms used in the network. We assume a *trusted* **administrative controller** (AC) in the network, which shares a pairwise secret key with each node in the network. As we will show later, the AC is mainly in charge of analyzing the traffic summaries gathered from different nodes and localizing any neighborhood with data-plane faults. Finally, we require nodes in the network be *loosely* time-synchronized, e.g., on the order of milliseconds. Loose time synchronization represents a common requirement for detecting packet delaying attacks [71, 14, 15] and nowadays even high-precision clock synchronization is available given the advent of GPS-enabled clocks and the adoption of IEEE 1588 [46].

7.2.1 Problem Formulation

Our goal is to design a practical and secure *neighborhood-based* fault localization protocol to identify a suspicious *neighborhood* (if any) that contains at least one malicious node. Recall that practicality translates to *path obliviousness*, *volatile path support* and *constant router state* as stated in Section 7.1. We further adopt the (α, β, δ)**-accuracy** [36] to formalize the security requirements as below:

- If more than β fraction of the packets are corrupted by a malicious node m, the fault localization protocol will raise a neighborhood containing m *or* one of its colluding nodes as suspicious with probability at least $1 - \delta$.
- In benign cases, if no more than α fraction of the packets are spontaneously corrupted (e.g., dropped) in a neighborhood, the fault localization protocol will raise the neighborhood as suspicious with probability at most δ.

The thresholds α and β are introduced to tolerate spontaneous failures (e.g., natural packet loss) and are set by the network administrator based on her experience and expectation of network performance.

Neighborhood-based fault localization enables the network administrator to scope further investigation to a 1-hop neighborhood to find out which router is compromised. It is also possible to further employ dedicated monitoring protocols, which only need to monitor a small region (the identified neighborhood) of the network to find the specific misbehaving router.

7.3 Challenges and Overview

In this section, we first describe the high-level steps of a general neighborhood-based fault localization and then explain the security challenges in the presence of strong adversaries. Finally, we present the key ideas in DynaFL that address these challenges.

7.3.1 High-Level Steps

The general steps a neighborhood-based fault localization takes are (i) *recording* local traffic summaries, (ii) *reporting* the traffic summaries to the AC, and (iii) *detecting* suspicious neighborhoods by the AC based on the received traffic summaries, as we sketch below. Though intuitive, these general steps face several security vulnerabilities and scalability challenges as Section 7.3.3 will show.

Recording. We divide the time in a network into consecutive *epochs*, which are synchronous among all the nodes including the AC in the network. For each neighbor r, a node s locally generates traffic summaries, denoted by $\mathrm{TS}_s^{\rightarrow r}$ and $\mathrm{TS}_s^{\leftarrow r}$, for the packet streams $\mathbb{S}_{sr}$ and $\mathbb{S}_{rs}$ in each epoch, respectively. Figure 7.3 depicts the router state in a toy example.

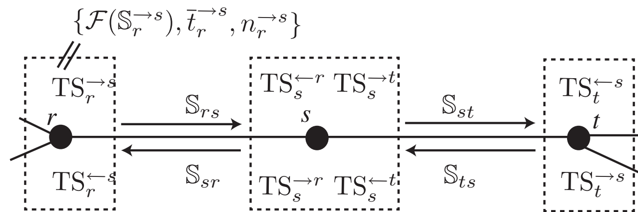


Figure 7.3: Router state for traffic summaries.

The traffic summary recorded by a node s should reflect both the packet contents and the

arrival/departure time seen at node s to enable the detection of malicious packet corruption and delay. For the sake of scalability, the traffic summary can not simply be an entire copy of all the original packets (or their hashes using a *cryptographic* hash function such as SHA-1 which provides one-wayness and collusion resistance) and their timing information. Instead, we use a *fingerprinting function* $\mathcal{F}$ to reflect the *aggregates* of packet contents to save both the router state and bandwidth consumption for reporting the traffic summaries to the AC. We denote the fingerprint for a packet stream $\mathbb{S}_{rs}$ generated by r as $\mathcal{F}(\mathbb{S}_r^{\rightarrow s})$, as Figure 7.3 depicts. In addition, as Figure 7.3 shows, for a packet stream $\mathbb{S}_{rs}$ (or $\mathbb{S}_{sr}$), the traffic summary of node r also contains the *average* departure time $\overline{t}_r^{\rightarrow s}$ (or arrival time $\overline{t}_r^{\leftarrow s}$) and the total number of packets $n_r^{\rightarrow s}$ (or $n_r^{\leftarrow s}$) in $\mathbb{S}_{rs}$ (or $\mathbb{S}_{sr}$) seen in the current epoch to enable the detection of packet delay attacks.

Reporting. At the end of each epoch, each node s sends its local traffic summaries to the AC.

Detection. After receiving the traffic summaries at the end of an epoch, the AC runs a consistency check over the traffic summaries in each neighborhood. A large inconsistency of the traffic summaries in a certain neighborhood $\mathbb{N}(s)$ indicates that $\mathbb{N}(s)$ is suspicious.

7.3.2 The Fingerprinting Function $\mathcal{F}$

Before we present the instantiation of $\mathcal{F}$, we first describe the general properties that $\mathcal{F}$ should satisfy. To enable the AC to detect suspicious neighborhoods, $\mathcal{F}$ should generate traffic summaries with the following two properties:

Property 1. *Given any two packet streams $\mathbb{S}$ and $\mathbb{S}'$, the "difference" between $\mathcal{F}(\mathbb{S})$ and $\mathcal{F}(\mathbb{S}')$ can give an estimation of the difference between $\mathbb{S}$ and $\mathbb{S}'$, denoted by:* $\Delta(\mathcal{F}(\mathbb{S}), \mathcal{F}(\mathbb{S})') \rightsquigarrow \Delta(\mathbb{S}, \mathbb{S}')$.

Defining the "difference" between $\mathcal{F}(\mathbb{S})$ and $\mathcal{F}(\mathbb{S}')$ is $\mathcal{F}$-specific, as we show shortly.

Property 2. *Given any two packet streams $\mathbb{S}$ and $\mathbb{S}'$, $\mathcal{F}(\mathbb{S} \cup \mathbb{S}') = \mathcal{F}(\mathbb{S}) \cup \mathcal{F}(\mathbb{S}')$.*

The $\cup$ operator on the left-hand side denotes a union operation of the two packet streams $\mathbb{S}$ and $\mathbb{S}'$. The $\cup$ operator on the right-hand side denotes a "combination" of $\mathcal{F}(\mathbb{S})$ and $\mathcal{F}(\mathbb{S}')$, which is $\mathcal{F}$-specific and defined shortly.

These two properties enable the conversion *from checking packet stream conservation to checking the conservation of traffic summaries in a neighborhood*. In other words, these two properties enable nodes to simply store the compact packet fingerprints instead of the original packet streams while still enabling the AC to detect the number of packets dropped, modified, and fabricated between two packet streams from their corresponding fingerprints.

Specifically, during the detection phase, the AC only needs to compare the difference between (i) the *combined* traffic summaries for packets *sent to* node s in $\mathbb{N}(s)$, i.e., $\cup_{i\in\mathbb{N}(s)} \mathcal{F}(\mathbb{S}_i^{\rightarrow s})$, and (ii) the *combined* traffic summaries for packets *received from* node s in $\mathbb{N}(s)$, i.e., $\cup_{i\in\mathbb{N}(s)} \mathcal{F}(\mathbb{S}_i^{\leftarrow s})$. By Properties 1 and 2:

$$
\begin{aligned}
&\Delta(\bigcup_{i\in\mathbb{N}(s)} \mathcal{F}(\mathbb{S}_i^{\rightarrow s}), \bigcup_{i\in\mathbb{N}(s)} \mathcal{F}(\mathbb{S}_i^{\leftarrow s})) \\
&= \Delta(\mathcal{F}(\bigcup_{i\in\mathbb{N}(s)} \mathbb{S}_i^{\rightarrow s}), \mathcal{F}(\bigcup_{i\in\mathbb{N}(s)} \mathbb{S}_i^{\leftarrow s})) \quad \text{based on Property 2} \\
&\rightsquigarrow \Delta(\bigcup_{i\in\mathbb{N}(s)} \mathbb{S}_i^{\rightarrow s}, \bigcup_{i\in\mathbb{N}(s)} \mathbb{S}_i^{\leftarrow s}) \quad \text{based on Property 1}
\end{aligned}
\tag{7.1}
$$

Note that $\Delta(\cup_{i\in\mathbb{N}(s)} \mathbb{S}_i^{\rightarrow s}, \cup_{i\in\mathbb{N}(s)} \mathbb{S}_i^{\leftarrow s})$ reflects the discrepancy between packets sent to and received from node s, and a large discrepancy indicates packet dropping, modification, and fabrication attacks in $\mathbb{N}(s)$.

Sketch for $\mathcal{F}$. The p^{th}moment estimation sketch [9, 30, 88] (as used by Goldberg et al. [36] for *path-based* fault localization) serves as a good candidate for $\mathcal{F}$. More specifically, p^{th}moment estimation schemes use a random linear map to transform a packet stream into a short vector, called the sketch, as the traffic summary. In *benign* cases, packets, if viewed as 1.5KB (the Maximum Transmission Unit) bit-vectors, are "randomly" drawn from $\{0,1\}^{1536\times 8}$. Hence, different packet streams will result in different sketches *with a very high probability (w.h.p.)*. Goldberg et al. [36] also extensively studied how to estimate the number of packets dropped, injected, or modified between two packet streams from the "difference" of two corresponding sketch vectors, thus satisfying Property 1. Specifically, the difference $\Delta(\mathcal{F}(\mathbb{S}), \mathcal{F}(\mathbb{S})')$ (used in Property 1) between two sketch

vectors is defined as:

$$\Delta(\mathcal{F}(\mathbb{S}), \mathcal{F}(\mathbb{S})^{'}) = ||\mathcal{F}(\mathbb{S}) - \mathcal{F}(\mathbb{S})^{'}||_p^p \tag{7.2}$$

where $||x||_p^p$ denotes the p^{th}moment of the vector x. We can further prove (see Appendix C) that the sketch satisfies Property 2 and the combination of $\mathcal{F}(\mathbb{S})$ and $\mathcal{F}(\mathbb{S})^{'}$ used in Property 2 is defined as:

$$\mathcal{F}(\mathbb{S}) \cup \mathcal{F}(\mathbb{S})^{'} = \mathcal{F}(\mathbb{S}) + \mathcal{F}(\mathbb{S})^{'} \tag{7.3}$$

where + denotes the addition of two vectors.

7.3.3 Challenges in a Neighborhood-based fault localization

From Property 1, we can further derive the following conditions on the fingerprinting function $\mathcal{F}$. Given any two packet streams $\mathbb{S}_r$ and $\mathbb{S}_t$ seen at nodes r and t, respectively, a fingerprinting function computed by r and t should satisfy:

$$\text{if } \mathbb{S}_r = \mathbb{S}_t, \mathcal{F}(\mathbb{S}_r) = \mathcal{F}(\mathbb{S}_t) \tag{7.4}$$

$$\text{if } \mathbb{S}_r \neq \mathbb{S}_t, \mathcal{F}(\mathbb{S}_r) \neq \mathcal{F}(\mathbb{S}_t) \text{ w.h.p.} \tag{7.5}$$

The first condition ensures the consistency of traffic summaries (more precisely, sketches in the traffic summaries) in the benign case when the packet streams are not corrupted between nodes r and t. The second condition ensures that if packet corruption happens between nodes r and t, inconsistency of the traffic summaries will be observed, which will then enable the estimation of packet difference in the corresponding packet streams (Property 1). However, these two conditions tend to be contradicting and lead to the following dilemma.

$\mathcal{F}$ without different secrets. If the random linear map in $\mathcal{F}$ (which can be implemented as a hash function [21]), is *not* computed with different secret keys by different nodes, a malicious node can predict the $\mathcal{F}$ output of *any other* node for *any* packet. Since $\mathcal{F}$ maps a set of packets (or

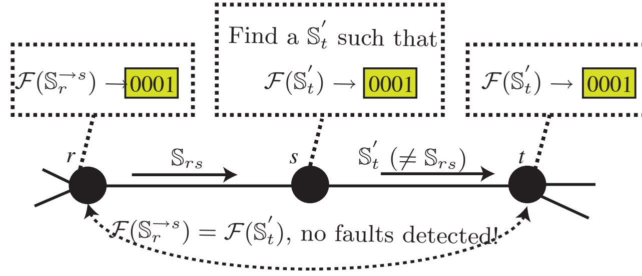


Figure 7.4: An example of stealthy packet modification attacks when nodes do not use different secret keys for computing $\mathcal{F}$. For simplicity, the sketch vector is represented as a '0-1' bit vector. The malicious node s modifies the packet stream in such a way that the modified packet stream $\mathbb{S}_t^{'}$ still results in the same sketch vector as $\mathbb{S}_{rs}$ at node t.

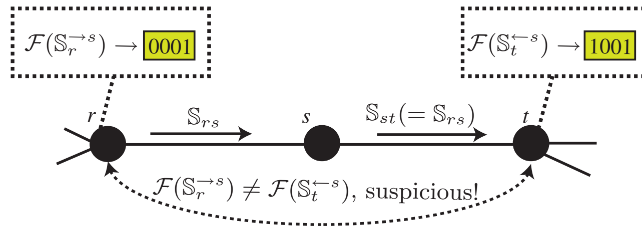


Figure 7.5: Illustration of the difficulty in using different secret keys when computing $\mathcal{F}$. The sketch vector is represented as a '0-1' bit vector for simplicity. In this example, nodes r, s and t use different secret keys when computing the Sketch to generate their traffic summaries.

their 160-bit cryptographic hashes) to a much smaller sketch, *hash collisions* will exist where two different packets produce the same $\mathcal{F}$ output (since sketch is not proven to preserve the collision resistance property of the cryptographic hash function). Hence, a malicious node can leverage such collisions to modify packets such that the modified/fabricated packets will produce the same $\mathcal{F}$ output at other nodes, violating the condition in (7.5). Figure 7.4 depicts such an example.

$\mathcal{F}$ with different secrets. If nodes compute $\mathcal{F}$ with different secret keys to satisfy the condition in (7.5), it is hard for the AC to perform a consistency check among the resulting sketches. For example, even the same packet stream would result in different sketches at different nodes, thus violating the condition in (7.4). Figure 7.5 depicts such an example. Since the sketch is only a *compact and approximate* representation of the original packet stream, the AC cannot revert the received sketches to the original packet streams to check packet stream conservation.

Scalability vs. sampling. Even with $\mathcal{F}$ for packet fingerprinting, a traffic summary over a huge number of packets can become too bandwidth-consuming to be sent frequently to the AC (e.g., every 20 milliseconds). For example, the number of packets for an OC-192 link (10Gbps) can be on the order of 10^7 per second in the worst case, which swells the size of a sketch to hundreds of bytes to bound the false positive rate below 0.001 [36] and may require several KB/s bandwidth for the reporting by *each* node. Packet sampling represents a popular approach to reducing bandwidth consumption, where each node only samples a *subset* of packets to feed into $\mathcal{F}$ for generating the traffic summaries. To enable a consistency check of the traffic summaries in a neighborhood, all nodes in a neighborhood should sample the *same* subset of packets, and the challenge is how to efficiently decide which subset of packets all nodes should agree to sample. For security, the sampling scheme must ensure that a malicious node cannot predict whether a packet to be forwarded will be sampled or not. Otherwise, the malicious node can drop any non-sampled packets without being detected.

The problem is further complicated by the presence of *collusion attacks* in our strong adversary model as well as our *path obliviousness* requirement. Several existing sampling schemes are broken when applied to our setting. For example, in Symmetric Secure Sampling (SSS) [36], the packet sender and receiver use a shared Pseudo-Random Function (PRF) $\mathcal{P}$ to coordinate their sampling. Imported to our setting, e.g., using the neighborhood example in Figure 7.5, nodes r and t share a secret key K_{rt} and a PRF $\mathcal{P}$, compute $\mathcal{P}$ with K_{rt} for each packet, and sample the packet if the PRF output is within a certain range. In this way, node s *itself* cannot know whether a packet is sampled or not. However, this approach fails in our setting. Take the topology in Figure 7.5 for example:

- If s and r collude, r can inform s of which packets are sampled, so that s can safely drop non-sampled packets and not be detected.

- Due to the dynamic traffic pattern, an ingress node r of a neighborhood $\mathbb{N}(s)$ does not know which egress node a packet will traverse in $\mathbb{N}(s)$ (if s has more neighbors than r and t, there exist multiple possible egress nodes than t). Hence, r does not know which PRF or secret key to use for packet sampling, given that r shares a different secret key with each node in $\mathbb{N}(s)$.

7.3.4 DynaFL Key Ideas

In DynaFL, nodes temporarily store the cryptographic hashes (which are collision-resistant) for all packets received/sent *per neighbor* in an epoch. At the end of each epoch e, nodes use *epoch sampling* to decide if packets in the epoch are to be fingerprinted; if so, nodes generate the traffic summaries and report them to the AC. This reduces both the communication overhead for sending the traffic summaries to the AC and the computational overhead for generating and checking the traffic summaries. Specifically, nodes first use the *same* per-epoch **sampling key** K_s^e (described shortly) for computing a PRF $\mathcal{P}$ to determine if the current epoch is "selected"; if and only if the current epoch is selected, nodes will use $\mathcal{F}$ with the *same* per-epoch **fingerprinting key** K_f^e (described shortly) to map packets into per-neighbor traffic summaries. Using the same K_s^e and K_f^e enables consistency checking over the traffic summaries from different nodes.

To address the packet modification attacks and collusion attacks mentioned earlier, nodes do *not* know the per-epoch K_s^e and K_f^e until the *end* of each epoch e, after they *have forwarded* (or possibly corrupted) packets in epoch e. Thus, when a packet is to be forwarded (or corrupted), a malicious node does not know K_s^e and K_f^e, and thus cannot predict whether this epoch is selected for sending traffic summaries, and if selected, what the sketch output will be for this packet. To achieve this property, in DynaFL, the trusted AC periodically sends the per-epoch K_s^e and K_f^e via **function disclosure messages** to all nodes at the end of each epoch in a reliable way (described later) and nodes use the received K_s^e and K_f^e to select epochs and fingerprint packets that have already been forwarded or corrupted.

A malicious node may first attempt to locally hold all the packets in an epoch e, and only forward or corrupt packets at the end of e when the malicious node learns K_s^e and K_f^e, thus being able to launch the sophisticated packet modification and selective packet corruption attacks as mentioned earlier. However, since the traffic summaries also include the average departure/arrival time of the sent/received packets, the malicious node will be detected with packet delay misbehavior in the detection phase.

Sections 7.4, 7.5, and 7.6 detail the recording, reporting, and detection phases in DynaFL, respectively. Section 7.7 presents the security analysis and Section 7.8 evaluates DynaFL's performance through measurements and simulations.

7.4 Recording Traffic Summaries

Nodes in DynaFL generate their traffic summaries by first temporarily storing all the received and sent packets for each epoch along with aggregate timing information. Then upon receiving the keys disclosed by the AC, nodes determine if the current epoch is selected with a keyed PRF $\mathcal{P}$, and if so, fingerprint the cached packets with keyed $\mathcal{F}$. The technical challenges in the recording phase are how to deal with *imperfect* time synchronization among nodes and packet transmission delay, and how to efficiently protect the function disclosure message from adversarial corruption. We explain how DynaFL solves these challenges in turn below.

Definition 20. *The epoch IDs are labeled as 0, 1, 2, ... If the current network time is t, then the current epoch ID is $\lfloor \frac{t}{l} \rfloor$, where l is the epoch length.*

7.4.1 Storing Packets

In the "ideal" case (with perfect time synchronization and no packet transmission delay), nodes simply need to store packets for the single "current" epoch and at the end of each epoch send the traffic summaries to the AC for that epoch. However, in practice, routers need to determine which epoch an incoming packet belongs to (or whether a received packet belongs to the current epoch or a previous, outdated epoch). One might attempt to let routers map received packets into epochs based on their local packet arrival time. However, this approach would introduce large errors for the following reasons:

- Though all the nodes in the network are *loosely* time-synchronized, e.g., ± 1 millisecond, the epoch intervals at different nodes may still be misaligned by up to a few milliseconds. This misalignment will result in a considerable number of packets being attributed to different epochs at different nodes, thus causing inconsistencies in the corresponding packet fingerprints.
- Due to the network transmission delay, a packet sent by a source at epoch e may arrive at another node at a different epoch $e + i$. In other words, a packet may have been received by an ingress node but not the egress node of a neighborhood at the end of an epoch when

nodes need to generate their packet fingerprints, thus producing inconsistencies in the traffic summaries.

To deal with imperfect time synchronization, the source in DynaFL embeds a *local* timestamp when sending each packet. Such a timestamp can be added as an additional flow header, using the TCP timestamp, or in the IP option field that all routers can process efficiently. Any router in the forwarding path will determine the corresponding epoch for each packet based on the embedded timestamp. In this way, we ensure that all routers put each packet in the same epoch for updating the traffic summaries. For example, if the timestamp embedded by the source is t_s and the epoch length is L, then all routers will map the packet into epoch $\lfloor \frac{t_s}{L} \rfloor$.

To eliminate traffic summary inconsistencies due to packet transmission delay, we also need to ensure that when generating traffic summaries for a certain epoch e, packets that are sent and not corrupted in epoch e are received by *all* the nodes in the forwarding paths. To this end, if the epoch length is set to L and the expected upper bound on the *one-way* packet transmission delay in the network is D, each router stores packets sent in the current epoch e *as well as* in previous $\lceil \frac{D}{L} \rceil$ epochs, denoted by $e-1, e-2, \ldots, e-\lceil \frac{D}{L} \rceil$. We call these epochs **live epochs**. Then at the end of an epoch e, nodes will generate and send to the AC the traffic summaries for the *oldest* live epoch $e-\lceil \frac{D}{L} \rceil$, in which the packets have either traversed all nodes in their forwarding paths or been corrupted. The periodic function disclosure messages that the AC sends synchronize the current epoch ID and the oldest live epoch ID for which traffic summaries are needed for reporting.

Hence, a node s maintains the following data structures for each neighbor r for each epoch, as Figure 7.6 also shows.

- The packet cache $\mathbb{C}_s^{\leftrightarrow r}$ temporarily stores hashes for packets in both $\mathbb{S}_s^{\rightarrow r}$ and $\mathbb{S}_s^{\leftarrow r}$ that are seen in a live epoch (using a cryptographic hash function such as SHA-1). Each entry contains the packet hash and a bit indicating if the packet belongs to $\mathbb{S}_s^{\rightarrow r}$ or $\mathbb{S}_s^{\leftarrow r}$.

- The router stores the *sum* of packet departure timestamps $t_s^{\rightarrow r}$ seen in $\mathbb{S}_s^{\rightarrow r}$ and the sum of packet arrival timestamps $t_s^{\leftarrow r}$ seen in $\mathbb{S}_s^{\leftarrow r}$ in a live epoch with millisecond precision.

- Finally, the router stores the total number of packets $n_s^{\rightarrow r}$ seen in $\mathbb{S}_s^{\rightarrow r}$ and $n_s^{\leftarrow r}$ seen in $\mathbb{S}_s^{\leftarrow r}$ in a live epoch.

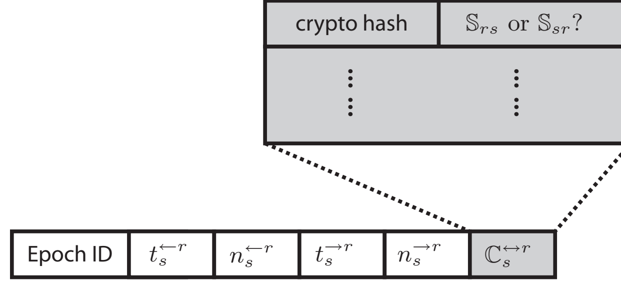


Figure 7.6: Router per-neighbor state details.

Among these data structures, $t_s^{\leftarrow r}$, $t_s^{\rightarrow r}$, $n_s^{\leftarrow r}$, and $n_s^{\rightarrow r}$ require small constant storage, around 8 or 4 bytes for each. $\mathbb{C}_s^{\leftrightarrow r}$ will be used for packet fingerprinting. The size of $\mathbb{C}_s^{\leftrightarrow r}$ depends only on the epoch length L and link bandwidth, but not the number of flows/paths traversing node s. As Section 7.8.1 shows, with an epoch length of 20 milliseconds and one-way network latency of 20 milliseconds, each router line-card requires only around 4MB of memory for an OC-192 link, which is readily available today.

For simplicity's sake, we use $\mathbb{C}_s^{\rightarrow r}$ and $\mathbb{C}_s^{\leftarrow r}$ to denote the packets cached for $\mathbb{S}_s^{\rightarrow r}$ and $\mathbb{S}_s^{\leftarrow r}$ by node s, respectively.

7.4.2 Secure Function Disclosure

At the end of each epoch e, the AC discloses the sampling key $K_s^{e-\lceil \frac{D}{L} \rceil}$ and fingerprinting key $K_f^{e-\lceil \frac{D}{L} \rceil}$ to all nodes in the network via a *function disclosure message* $\mathtt{d}_{\mathtt{AC}}$, and requests the traffic summaries for the oldest live epoch $e - \lceil \frac{D}{L} \rceil$. Obviously, $\mathtt{d}_{\mathtt{AC}}$ itself needs to be protected from data-plane attacks (dropping, modification, fabrication, or delaying) by a malicious node during end-of-epoch broadcasting. It might be tempting to let the AC use digital signatures to authenticate $\mathtt{d}_{\mathtt{AC}}$ in order to address malicious modification and fabrication; however, frequently generating and verifying the signatures on a per-epoch basis can be expensive (e.g., an epoch can be as short as 20 milliseconds and signature generation and verification time could be on the order of milliseconds).

Fortunately, the function disclosure message $\mathtt{d}_{\mathtt{AC}}$ is transmitted at the end of each epoch synchronously among all the nodes. If a malicious node s drops $\mathtt{d}_{\mathtt{AC}}$, the AC will fail to receive the traffic summaries of certain neighbors of s, thus detecting $\mathbb{N}(s)$ as suspicious. For example in Figure 7.7, if s drops $\mathtt{d}_{\mathtt{AC}}$ instead of forwarding it to its neighbor r, node r cannot fingerprint

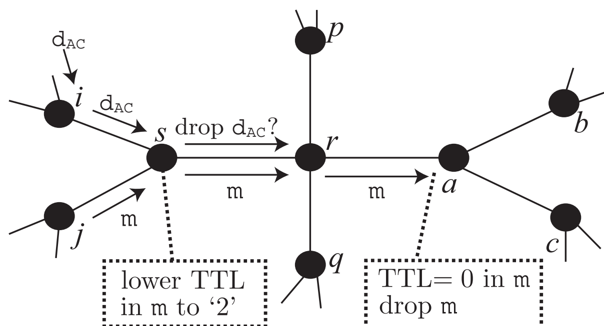


Figure 7.7: Possible attacks in the recording phase. A malicious node s may attempt to drop the function disclosure message $\mathtt{d}_{\mathtt{AC}}$, or manipulate the TTL value to cause packets to be dropped at a remote place (node a in this example), thus framing a remote neighborhood ($\mathbb{N}(a)$ in this example).

the packets to generate traffic summaries, thus failing the consistency check of traffic summaries in $\mathbb{N}(s)$. As we show in Section 7.5, the AC expects to receive traffic summaries within a short amount of time after each epoch ends; delaying $\mathtt{d}_{\mathtt{AC}}$ more than that amount of time is effectively equivalent to dropping $\mathtt{d}_{\mathtt{AC}}$ and causes the malicious node's neighborhood to be detected. Thus, the remaining problem is to prevent the modification and fabrication of $\mathtt{d}_{\mathtt{AC}}$, which is equivalent to authenticating $\mathtt{d}_{\mathtt{AC}}$ to all nodes in the network *without* the use of digital signatures. Section 7.7 further elaborates why the authentication of $\mathtt{d}_{\mathtt{AC}}$ is needed for security purposes.

In DynaFL, time in the network is loosely time-synchronized and divided into consecutive epochs; the authentication of $\mathtt{d}_{\mathtt{AC}}$ is required only once per epoch. This setting is naturally aligned with that of the TESLA broadcast authentication [77], which authenticates broadcast messages ($\mathtt{d}_{\mathtt{AC}}$ in our case) using only Message Authentication Codes (MACs) with keys derived from a one-way hash chain. As Figure 7.8 shows, the AC applies a one-way hash function H repeatedly on the root key K_r to derive a set of epoch authentication keys, and uses key K_e to compute a MAC for authenticating $\mathtt{d}_{\mathtt{AC}}$ in epoch e. The AC publishes K_0 through the network so that nodes can verify if any given epoch key is indeed derived from the genuine one-way hash chain. Then $\mathtt{d}_{\mathtt{AC}}$ in epoch e includes (i) the current epoch ID e, the oldest live epoch ID $j = e - \lceil \frac{D}{L} \rceil$ to be examined, sampling and fingerprinting keys, a MAC computed with K_e for the current epoch, and (ii) the key K_j for computing the MAC in a *previous* epoch j, by which nodes can verify the authenticity of $\mathtt{d}_{\mathtt{AC}}$ in

Figure 7.8: One-way hash chain example.

epoch j (verification delayed by $\lceil \frac{D}{L} \rceil$ epochs). That is:

$$\begin{aligned} \mathtt{d}_{\mathtt{AC}} = & e||j||K_s^j||K_f^j|| \\ & \mathrm{MAC}_{K_e}(e||j||K_s^j||K_f^j)|| \\ & K_j \end{aligned} \tag{7.6}$$

where $||$ denotes concatenation. Section 7.7 describes the reason for disclosing the key for epoch $j = e - \lceil \frac{D}{L} \rceil$ instead of epoch $e-1$.

Furthermore, DynaFL creates a spanning tree in the network rooted at the AC, along which $\mathtt{d}_{\mathtt{AC}}$ is delivered to each node. Since DynaFL uses a *pre-generated, static* spanning tree for the broadcast messages, there is no need for dynamic path support when protecting $\mathtt{d}_{\mathtt{AC}}$.

7.4.3 Sampling and Fingerprinting

Given the disclosed K_s^j and K_f^j at the end of an epoch e, each node t first uses the sampling PRF $\mathcal{P}$ with K_s^j, denoted by $\mathcal{P}_{K_s^j}$, to determine if the oldest live epoch j is selected. If so, node t then uses the fingerprinting function $\mathcal{F}$ to map the cached packet hashes in each per-neighbor stream into a sketch vector, i.e., $\mathcal{F}_{K_f^j}(\mathbb{C}_t^{\rightarrow r})$ or $\mathcal{F}_{K_f^j}(\mathbb{C}_t^{\leftarrow r})$, computed with the given K_f^j. Finally, node t generates *two* traffic summaries $\mathrm{T}_t^{\rightarrow r}$ and $\mathrm{T}_t^{\leftarrow r}$ for a neighbor r:

- $\mathrm{T}_t^{\rightarrow r}$ for packet stream $\mathbb{S}_t^{\rightarrow r}$ includes a fingerprint $\mathcal{F}_{K_f^j}(\mathbb{C}_t^{\rightarrow r})$, average packet departure time $\overline{t}_t^{\rightarrow r} = \frac{t_t^{\rightarrow r}}{n_t^{\rightarrow r}}$, and the total number $n_t^{\rightarrow r}$ of packets seen in $\mathbb{S}_t^{\rightarrow r}$ in epoch j;

- $\mathrm{T}_t^{\leftarrow r}$ for packet stream $\mathbb{S}_t^{\leftarrow r}$ includes a fingerprint $\mathcal{F}_{K_f^j}(\mathbb{C}_t^{\leftarrow r})$, average packet arrival time $\overline{t}_t^{\leftarrow r} = \frac{t_t^{\leftarrow r}}{n_t^{\leftarrow r}}$, and the total number $n_t^{\leftarrow r}$ of packets seen in $\mathbb{S}_t^{\leftarrow r}$ in epoch j.

Figure 7.9 summarizes the fault localization-related packet processing inside a DynaFL router. We detail $\mathcal{P}$ and $\mathcal{F}$ in the following.

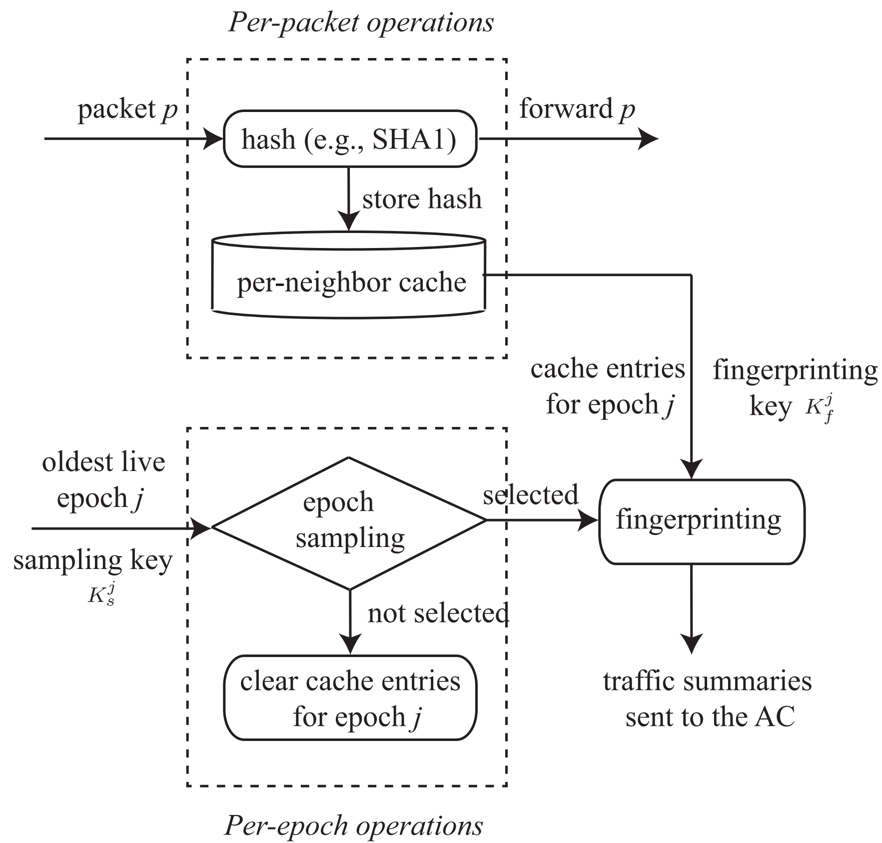


Figure 7.9: fault localization-related packet processing inside a DynaFL router.

Implementing $\mathcal{P}$. Specifically, $\mathcal{P}$ maps an epoch ID to a n-bit integer uniformly distributed in $[0, 2^n - 1]$. Given a *sampling rate* $\lambda \in (0, 1)$, a node computes $\mathcal{P}_{K_s^j}$ over the epoch ID j that is being examined, and epoch j is selected iff:

$$\mathcal{P}_{K_s^j}(j) < \lambda \cdot 2^n \tag{7.7}$$

In this way, on average a fraction λ of the epochs will be selected. Since nodes use $\mathcal{P}$ with the same K_s^j for epoch sampling, in benign case, nodes will select the same set of epochs, thus ensuring the consistency of the traffic summaries in a neighborhood.

Implementing $\mathcal{F}$. We use the second-moment sketch computed with K_f^j as a case study to implement $\mathcal{F}$, and analyze the size of the sketch vector to achieve Property 1 with the (α, β, δ)-accuracy. We assume 10^7 packets per second in the worst case for an OC-192 link with an epoch length of L (seconds). Then, the number of packets η in a sampled epoch is $\eta = L \cdot 10^7$. Using the classical Sketch due to Alon et al. [11] for example, the storage requirement for the sketch is given by:

$$M \times \log_2 \sqrt{2\eta \ln(\frac{200N}{\delta})} \tag{7.8}$$

$$\begin{aligned} &\text{where } M > \frac{12}{\epsilon^2}\frac{1}{3-2\epsilon}\ln\frac{1}{\delta} \\ &\text{and } \epsilon = \frac{\beta-\alpha}{\beta+\alpha}. \end{aligned} \tag{7.9}$$

In Section 7.8.1 we derive numeric values for the size of the sketch vector based on the epoch length L.

Dealing with TTL attacks. Certain fields in the IP header, such as the TTL, checksum, and some IP option fields, will change at each hop. Both sampling and fingerprinting in DynaFL need to properly deal with these variant fields to avoid both false positives and false negatives. Take the TTL field for instance hereinafter (though the arguments apply similarly to other variant fields). On the one hand, if $\mathcal{P}$ and $\mathcal{F}$ are computed over the entire packets including the TTL field, even

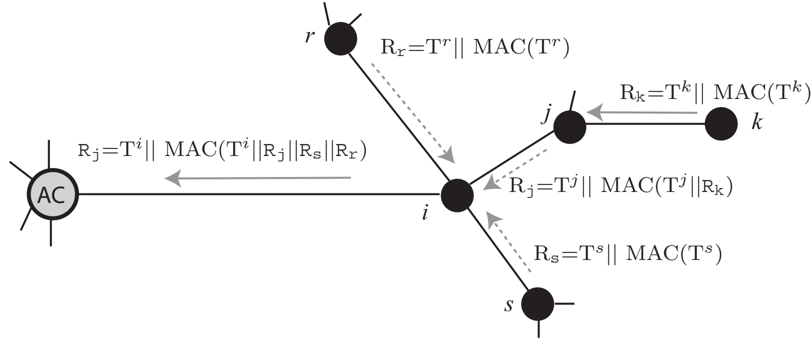


Figure 7.10: Example of secure transmission of traffic summary reports. For brevity, we denote the traffic summaries of a node i as T^i and omit the secret key for the MAC notation.

in the benign case the same packet stream will leave different traffic summaries (or precisely, the sketch vectors) at ingress and egress nodes. On the other hand, if $\mathcal{P}$ and $\mathcal{F}$ are computed over the entire packets excluding the TTL field, a malicious node can modify the TTL field at liberty without affecting the traffic summaries. Figure 7.7 depicts an example TTL attack, where the malicious node s lowers the TTL value to 2 in the packets and causes the packets to be dropped at the 2-hop-away downstream node a, thus framing neighborhood $\mathbb{N}(a)$.

To address the TTL attacks, when computing $\mathcal{P}$ and $\mathcal{F}$, each node r performs either of the following:

- For a packet received from a neighbor, node r computes $\mathcal{P}$ and $\mathcal{F}$ over the entire packet *including* the TTL field.

- For a packet sent to a neighbor, node r computes $\mathcal{P}$ and $\mathcal{F}$ over the packet, but with the TTL field additionally decreased by 2 (equal to the TTL value at the 2-hop-away egress node in $\mathbb{N}(r)$).

In this way, node r in Figure 7.7 simply uses the TTL value as contained in the packets received from s when computing $\mathcal{F}$ and $\mathcal{P}$, since the ingress nodes in $\mathbb{N}(s)$ (nodes i and j) must have computed $\mathcal{F}$ and $\mathcal{P}$ with an adjusted TTL value equal to that at node r.

The TTL value in a packet is also decremented by one for every second the packet is buffered at a router. Holding a packet longer than one second at a router is treated as a packet delaying attack and will be detected due to the use of the above construction.

7.5 Reporting Traffic Summaries

If an epoch is selected, after fingerprinting, a node t generates two traffic summaries $\mathrm{T}_t^{\rightarrow r}$ and $\mathrm{T}_t^{\leftarrow r}$ for each neighbor r, and sends them to the AC in a traffic summary report denoted by $\mathcal{R}_t$. The challenge in the recording phase is to protect the traffic summary reports from being corrupted.

In DynaFL, nodes form a static spanning tree rooted at the AC for sending the traffic summaries. Given the spanning tree, the goal is to protect the traffic summary reports $\mathcal{R}_t$s from different nodes destined to the AC. Although $\mathcal{R}_t$s are also subject to data-plane attacks, they are transmitted over *static* and *pre-generated* paths in the spanning tree. Hence, dynamic traffic is no longer a concern, thus substantially simplifying the problem. Specifically, DynaFL utilizes an *Onion Authentication* approach to protect the transmission of $\mathtt{d}_{\mathtt{AC}}$ along each path in the spanning tree. In a nutshell, within a short timer at the end of each epoch, each node t needs to send its traffic summary report $\mathcal{R}_t$ to the AC, and $\mathcal{R}_t$ is authenticated with a MAC computed using a pairwise secret key shared between node t and the AC. The traffic summary reports from different nodes are sent in an *onion* fashion. For example in Figure 7.10, $\mathcal{R}_j$ includes the report $\mathcal{R}_k$ of node k. In this way, DynaFL efficiently protects $\mathtt{d}_{\mathtt{AC}}$ without the use of expensive asymmetric cryptography. Section 7.7 gives a more detailed security analysis of such an Onion Authentication approach.

7.6 Detection

The AC performs consistency checks for each neighborhood $\mathbb{N}(r)$ based on the received traffic summaries. However, since an epoch may only have a small number of packets, detecting a suspicious neighborhood based on the consistency checks for *individual* epochs can introduce a large error rate. Take an extreme case for example: if in a certain epoch a neighborhood $\mathbb{N}(r)$ only transmits a single packet and the packet was spontaneously lost, concluding that the packet loss rate is 100% and $\mathbb{N}(r)$ is suspicious would be inaccurate.

To deal with this problem, we still perform the consistency checks and estimate the discrepancy for individual epochs, but make the detection based on the *aggregated* discrepancies over a set of E epochs (called **accumulated epochs**), so that the total number of packets over the E epochs is more than a certain threshold N to give a high enough accuracy (e.g., $> 99.9\%$) on the detection

results. Section 7.8 studies the value of N. Therefore, the AC stores the traffic summaries for each neighborhood and makes detection when the total number of packets N is reached. More specifically, let $n_x^{\leftarrow y}(e)$ and $n_x^{\rightarrow y}(e)$ denote the $n_x^{\leftarrow y}$ and $n_x^{\leftarrow y}$ in the traffic summary for epoch e, respectively; for a certain neighborhood $\mathbb{N}(r)$, whenever

$$max\{\sum_e \sum_i n_i^{\rightarrow r}(e), \sum_e \sum_i n_i^{\leftarrow r}(e)\} > N \tag{7.10}$$

(where $i \in \mathbb{N}(r)$ and e iterates over all the accumulated epochs), indicating N is reached, the AC performs the following checks to inspect if $\mathbb{N}(r)$ is suspicious:

1. Flow conservation. The AC first extracts $n_i^{\rightarrow r}(e)$ and $n_i^{\leftarrow r}(e)$ for each node i in $\mathbb{N}(r)$ for each epoch e, and calculates the difference between the number of packets sent to r and the number of packets received from r over all the E accumulated epochs. If the ratio of the difference to the total number of packets in all the E accumulated epochs is larger than a threshold β, i.e.:

$$\frac{|\sum_e \sum_i n_i^{\rightarrow r}(e) - \sum_e \sum_i n_i^{\leftarrow r}(e)|}{max\{\sum_e \sum_i n_i^{\rightarrow r}(e), \sum_e \sum_i n_i^{\leftarrow r}(e)\}} > \beta \tag{7.11}$$

then the AC detects $\mathbb{N}(r)$ as suspicious. The threshold β is set based on the administrator's expectation of the natural packet loss rate; e.g., in the simulations in Section 7.8 we set β to be four times of the natural packet loss rate in a neighborhood.

2. Content conservation. The AC then extracts the sketches in the traffic summaries in $\mathbb{N}(r)$, and estimates the discrepancy δ_f between the sketches for packets sent to r and the sketches for packets received from r. The AC detects $\mathbb{N}(r)$ as malicious if δ_f is larger than a certain threshold, i.e.,:

(7.12)

$$\delta_f > \frac{2\alpha\beta}{\alpha+\beta} \times max\{\sum_e \sum_i n_i^{\rightarrow r}(e), \sum_e \sum_i n_i^{\leftarrow r}(e)\}$$

where

$$\delta_f = || \cup_{i\in\mathbb{N}(r)} \mathcal{F}_{K_f^j}(\mathbb{C}_i^{\leftarrow r}) - \cup_{i\in\mathbb{N}(r)} \mathcal{F}_{K_f^j}(\mathbb{C}_i^{\rightarrow r})||_2^2$$

It has been proven [36] that the above threshold can satisfy the (α, β, δ)-accuracy defined in Section 7.2.1.

3. Timing consistency. Finally, the AC extracts the difference between the average packet departure time and arrival time, and concludes that $\mathbb{N}(r)$ is suspicious if the difference is larger than the expected upper bound on the 2-hop link latency.

7.7 Security Analysis

We show that DynaFL is secure against all attacks that are possible in the misbehavior space given our adversary model. By our definition, a malicious router can drop, modify, fabricate, and delay packets. In addition, a malicious router can attack data packets, function disclosure messages $\mathtt{d}_{\mathrm{AC}}$, and reporting messages. We first show DynaFL's security against a single malicious node and then sketch DynaFL's security against colluding nodes.

Security against corrupting the data packets. Dropping, modifying, and fabricating data packets in a neighborhood $\mathbb{N}(m)$ will cause inconsistencies between sketches in $\mathbb{N}(m)$ as mentioned earlier. Delaying data packets in $\mathbb{N}(m)$ will cause abnormal deviation between average packet arrival/departure timestamps in $\mathbb{N}(m)$. If a malicious router changes the timestamps in data packets embedded by the source nodes, it is equivalent to modifying packets and packets may be mapped to different epochs, in which case such an attack will manifest itself by causing inconsistencies in the sketches of a neighborhood containing the malicious router.

Security against corrupting $\mathtt{d}_{\mathrm{AC}}$. As we mentioned earlier, if a malicious node m drops the $\mathtt{d}_{\mathrm{AC}}$, some nodes adjacent to m will fail to send the correct traffic summaries to the AC, thus causing a neighborhood containing m to be detected. We note that the authentication of $\mathtt{d}_{\mathrm{AC}}$ is needed. Otherwise, a malicious node can replace the sampling and fingerprinting keys with its own fake keys, by which the malicious node can predict the output of other nodes's sketches and perform packet modification attacks. In addition, if the epoch IDs in $\mathtt{d}_{\mathrm{AC}}$ were not authenticated, a malicious node can replace the oldest live epoch ID in $\mathtt{d}_{\mathrm{AC}}$ for which the traffic summaries are requested with the

current epoch ID. In this way, inconsistencies of traffic summaries can be detected for some *benign* neighborhood due to the packet transmission delay as Section 7.4.1 describes. With the (delayed) authentication of $\mathtt{d}_{\mathtt{AC}}$, any attempt to modify $\mathtt{d}_{\mathtt{AC}}$ will be detected (after $\lceil \frac{D}{L} \rceil$ epochs).

It is noteworthy that the $\mathtt{d}_{\mathtt{AC}}$ sent at the end of epoch e cannot simply disclose the MAC secret key K_{e-1} for the previous epoch $e-1$. This is because at the time K_{e-1} is disclosed, the $\mathtt{d}_{\mathtt{AC}}$ sent at the end of epoch $e-1$ may still have not reached certain nodes. Hence, a malicious node which has already received K_{e-1} might send K_{e-1} to a downstream colluding node via an out-of-band channel, so that the colluding node can break the authenticity of the $\mathtt{d}_{\mathtt{AC}}$ sent in epoch $e-1$. Hence, at the end of an epoch e, we disclose the MAC key for epoch $e - \lceil \frac{D}{L} \rceil$ to ensure the $\mathtt{d}_{\mathtt{AC}}$ sent in epoch $e - \lceil \frac{D}{L} \rceil$ has reached all the nodes in the network.

Security against corrupting the reporting messages. First, due to the use of the Onion Authentication, a malicious node m cannot *selectively* drop the reporting messages of a *remote* (non-adjacent) node r, to frame a neighborhood containing node r. Since all the accumulated reporting messages are "combined" at each hop, m can only drop the reporting messages from its *immediate* neighbors, which will manifest a neighborhood containing m as suspicious.

Security against colluding attacks. We illustrate DynaFL's security against colluding attacks via a toy example shown in Figure 7.11. We show that for a malicious node m which actually corrupts packets, *as long as one benign node exists in* $\mathbb{N}(m)$*, a neighborhood containing either* m *or one of its colluding nodes will be detected*. The key observation is that since the traffic summaries are sent to the AC and the AC performs the detection, *each node can only claim one traffic summary per selected epoch*. To simplify the analysis while still unveiling the intuition, we only consider the number (but not the payload) of packets sent by each node, as shown in Figure 7.11. Suppose nodes c and d are colluding, and node d drops 50 packets. As long as node e is benign in $\mathbb{N}(d)$, to cover the misbehavior of d, the colluding node c has to send a traffic summary to the AC falsely claiming it sent "50" packets to d (and thus received "50" packets from node b). However, this claim will make the neighborhood $\mathbb{N}(b)$ suspicious since the benign node a will claim it sent 100 packets to b.

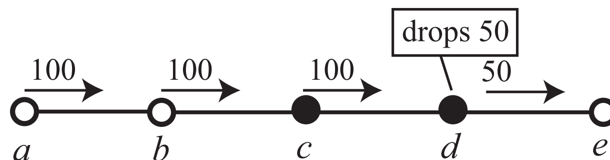


Figure 7.11: Example of DynaFL's security against colluding nodes. A number denotes the packet count each node sends.

7.8 Performance Evaluation

In this section, we analyze the protocol overhead and study the detection efficiency of DynaFL via measurements and simulations, with our implementation of the classic Sketch [11] in C++.

7.8.1 Storage Overhead

DynaFL incurs only per-neighbor state while existing secure path-based fault localization protocols require per-source and per-path state. In this section, we quantify the per-neighbor storage overhead of a DynaFL router r, which primarily includes the packet cache and the sketch for each neighbor s.

Sketch size. We derive numeric values of the sketch size based on Equations 7.8 and 7.9, using an example setting where the average packet size is 300 bytes and the link's capacity is 10 Gbps (an OC-192 link). Furthermore, we consider $\delta = 0.001$ and $\beta = 2\alpha$ for the (α, β, δ)-accuracy, i.e., the false positive rate and false negative rate of the sketch-based detection are limited under 0.001. Figure 7.12 plots the result, from which we can see that a sketch with fewer than 500 bytes can already yield a desirable accuracy.

Cache size and per-neighbor storage overhead. We now study the cache size for temporarily storing packet hashes in live epochs, which, together with the sketch size analyzed above, constitutes the per-neighbor storage overhead of a DynaFL router. We denote the upper bound of one-way network latency as D, epoch length as L, and the number of packets per second as η. Using 20-byte

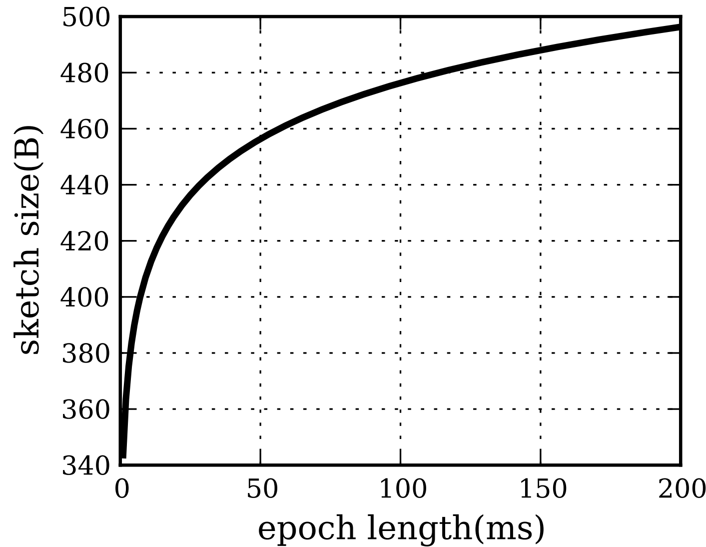


Figure 7.12: Sketch size for an OC-192 link with the average packet size of 300 bytes and $\delta = 0.001$.

packet hashes, the cache size is given by:

$$\lceil \frac{D}{L} + 1 \rceil \times 20 \cdot \eta \cdot L \tag{7.13}$$

We omit the 1-bit indicator for each packet hash entry to indicate which packet stream the packet belongs to (see Figure 7.6). Assuming the per-neighbor sketch size is 500 bytes, one-way latency $D = 20$ms, and the average packet size is 300 bytes for an OC-192 link, we derive the per-neighbor storage overhead of a DynaFL router with different epoch lengths shown in Figure 7.13. We can observe that, with an epoch length of 20ms, only around 4MB is required per-neighbor. The "humps" exist in the curve due to the use of the ceiling function in Equation 7.13.

7.8.2 Key Management Overhead

One distinct advantage DynaFL presents is that each router in DynaFL shares only one secret key with the AC, whereas in path-based fault localization protocols it is *necessary* for each router to share a secret key with each source node in the network in the worst case [21], which dramatically complicates the key management and broadens the vulnerability surface. To quantify DynaFL's

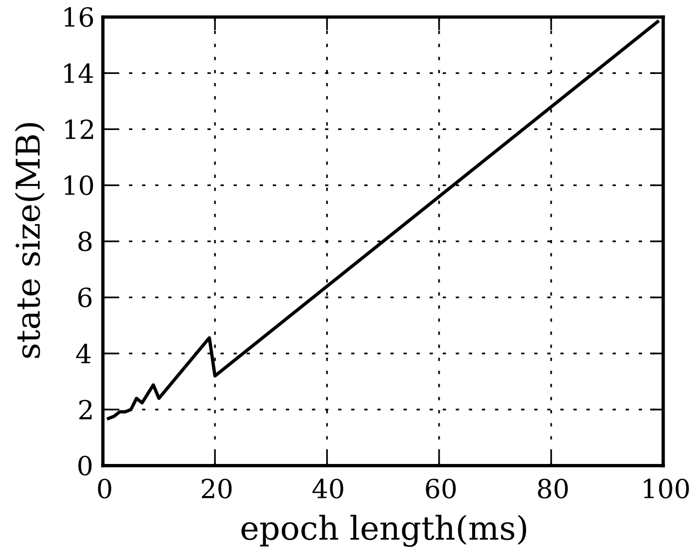


Figure 7.13: Router per-neighbor for an OC-192 link with the average packet size of 300 bytes and one-way network latency as 20ms.

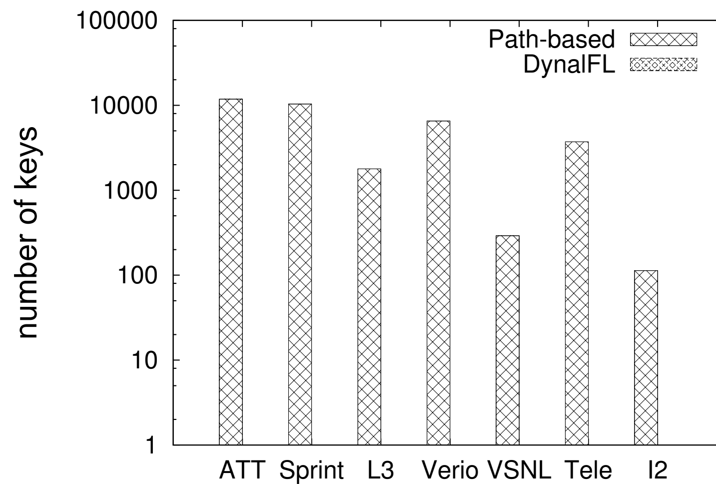


Figure 7.14: Key management overhead at each router. A router in DynaFL always requires just one key shared with the AC (hence not visible in the figure).

advantage over path-based fault localization protocols, we leverage the measured ISP topologies from the Rocketfuel dataset [84] and the topology from Internet2 [8]. Figure 7.14 shows the maximum number of keys each router needs to manage in path-based fault localization protocols; and a router in DynaFL always requires only one secret key shared with the AC (thus invisible in the figure). We can see that the number of keys a router needs to manage in path-based fault localization protocols is 100 to 10000 times higher than that in DynaFL.

7.8.3 Bandwidth Overhead

We analyze the bandwidth consumption on each link by the reporting traffic summaries based on the measured ISP topologies from the RocketFuel dataset [84]. Recall that the reporting messages are transmitted along a spanning tree rooted at the AC. Hence, the bandwidth consumption by the reporting messages on a link is determined by the number of children below that link and the degrees of the children.

For each ISP topology, we first select a "central" node as the AC, which is the node in the network that has the highest fraction of all shortest paths that pass through that node. Then, we create a minimum spanning tree rooted at the central node (or the AC) for transmitting reporting messages to the AC. We consider the epoch length L=20ms, a per-neighbor traffic summary as 500 bytes, and the epoch sampling rate is 1%. Hence, on average, each node only sends one reporting packet in every two seconds. Figure 7.15 plots the results for ISPs with AS numbers 1221, 1239, 1755, 3257, 3967, and 6461. From the results, we can see that the fraction of bandwidth used for reporting traffic summaries on a link is small for all topologies (e.g., between 0.002% and 0.012% for an OC-192 link).

7.8.4 Detection Delay

As Section 7.6 states, the AC performs consistency checks and detects any anomalies only when the total number of packets over multiple epochs is accumulated more than a certain threshold N in order to give a high enough accuracy (e.g., >99.9%) on the detection results. Hence, the number of packets N characterizes the detection delay of the fault localization protocol. We fully implement the classic Sketch due to Alon et al. [11] in C++ with a four-wise hash function, and

perform simulations to study N.

Since in DynaFL, neighborhoods are inspected by the AC *independently*, we also perform simulations for independent neighborhoods with different sizes. Since we showed DynaFL's security against colluding attacks in Section 7.7, we emulate a single malicious node in our simulations. Our setting is as follows. The natural packet loss rate in a neighborhood is 0.001 and the detection thresholds for both flow conservation and content conservation are $\beta = 2\alpha = 0.004$. Figure 7.16 depicts the false positive rates in benign cases where no malicious routers exist in the neighborhood. We can see that with $N > 5000$ packets, the false positive rate is under 1%.

Figure 7.17 shows the false negative rates with a malicious router which only drops packets with a probability of 0.005. Figure 7.18 plots the false negative rates with a malicious router which both drops and modifies packets with a probability of 0.005, respectively. We can see that the sketch-based approach is effective in detecting packet modification attacks, since by modifying packets the malicious router is detected faster in Figure 7.18 than in Figure 7.17.

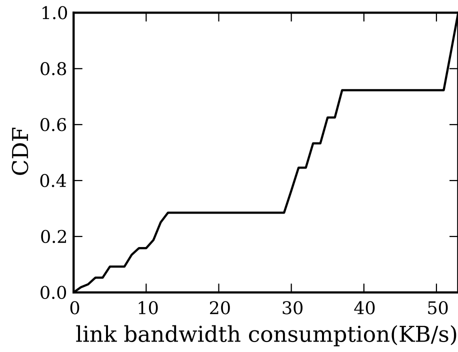
7.9 Summary

After identifying the fundamental limitations of previous path-based fault localization protocols, we explore a neighborhood-based FL approach. We present DynaFL, which utilizes delayed function disclosure, a novel technique that enables secure yet efficient checking of packet content conservation.

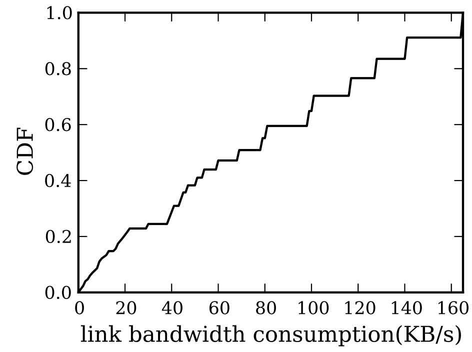
While existing path-based FL protocols aim to identify a specific faulty *link* (if any), DynaFL localizes fault to a coarser-grained 1-hop neighborhood, to achieve four distinct advantages. First, DynaFL does not require any minimum duration time of paths or flows in order to detect data-plane faults as path-based FL protocols do. Thus, DynaFL can fully cope with short-lived flows which are popularly seen in modern networks. Second, in DynaFL, a source node does not need to know the exact outgoing path while path-based FL protocols require so. Hence, DynaFL can support agile (e.g., packet-level) load balancing such as VL2 routing [38] for datacenter networks. Third, a DynaFL router only needs around 4MB per-neighbor state based on our classic Sketch implementation, while a router in a path-based FL protocol requires per-path state. Finally, a

DynaFL router only maintains a single secret key shared with the AC, while a router in a path-based FL protocol needs to manage 100 to 10000 secret keys in measured ISP topologies.

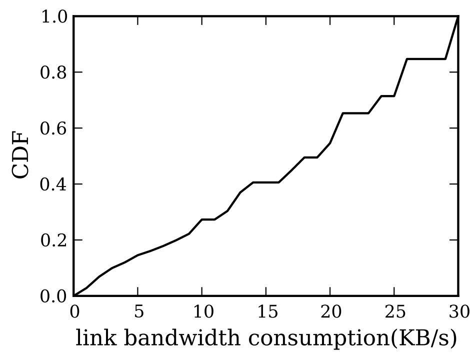
We anticipate that our work can spark future endeavors in designing practical network FL protocols for modern networks such as ISP, enterprise, and datacenter networks.



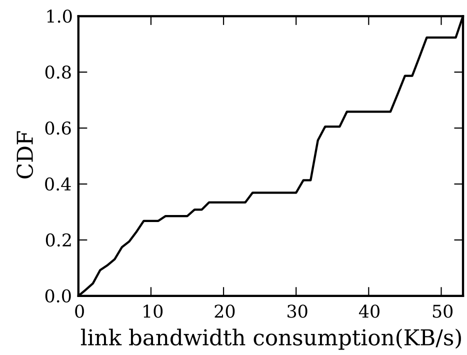
(a) ISP 1221



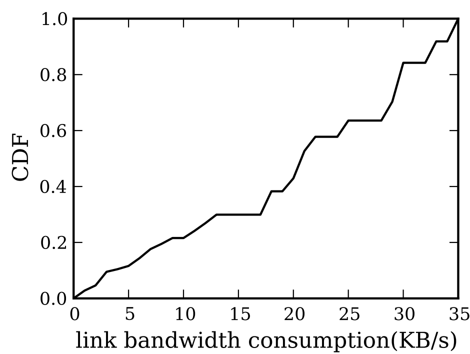
(b) ISP 1239



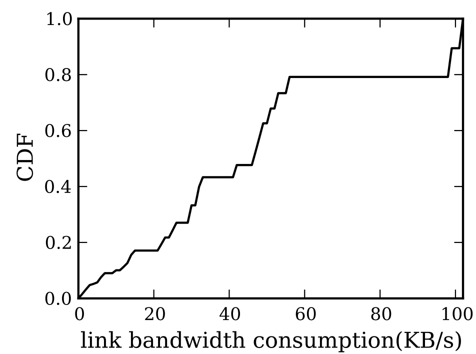
(c) ISP 1755



(d) ISP 3257



(e) ISP 3967



(f) ISP 6461

Figure 7.15: CDF of per-link bandwidth consumption for the reporting messages in DynaFL.

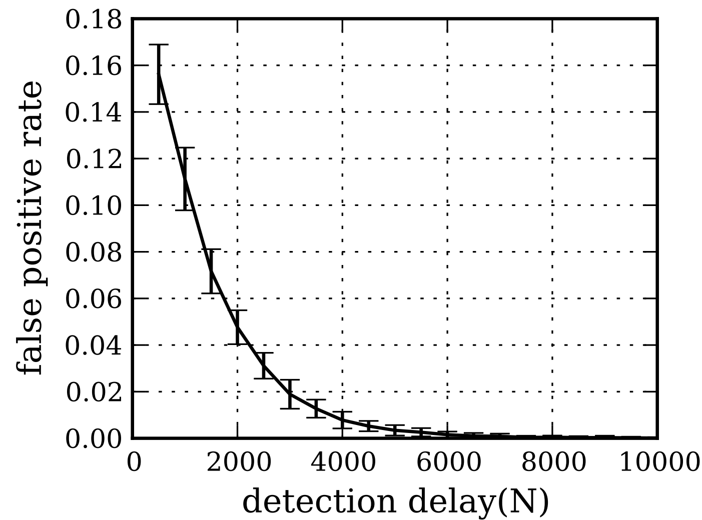


Figure 7.16: False positive rates with no malicious activity in a neighborhood with different numbers of nodes. The natural packet loss rate in a neighborhood is 0.001 and the detection thresholds for both flow conservation and content conservation are $T_d = \beta = 2\alpha = 0.004$.

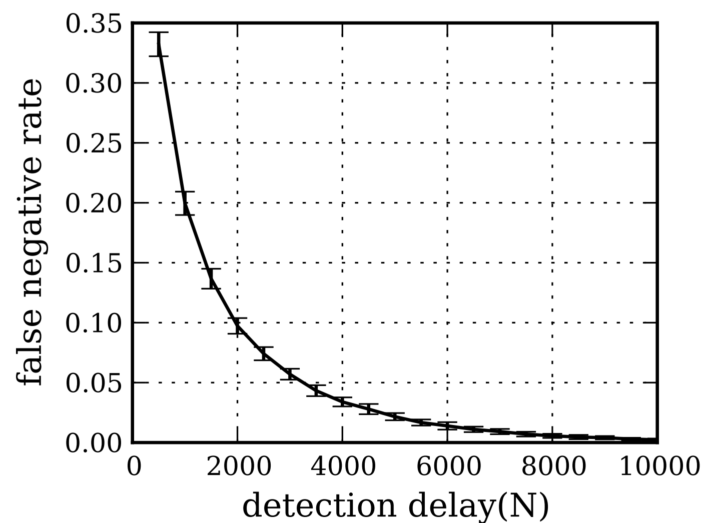


Figure 7.17: False negative rates in a malicious neighborhood with five nodes, where the malicious node *only drops* packets. The natural packet loss rate in a neighborhood is 0.001, the detection thresholds for both flow conservation and content conservation are $T_d = \beta = 2\alpha = 0.004$, and the malicious packet dropping rate is 0.005.

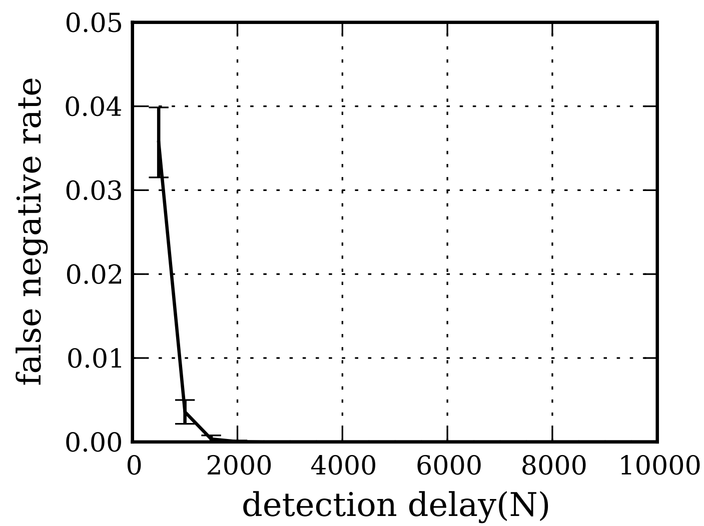


Figure 7.18: False negative rates in a malicious neighborhood with five nodes, where the malicious node both drops packets and modifies packets. The natural packet loss rate in a neighborhood is 0.001, the detection thresholds for both flow conservation and content conservation are $T_d = \beta = 2\alpha = 0.004$, the malicious packet dropping rate is 0.005, and the malicious packet modification rate is 0.005.

Chapter 8

Related Work

In this chapter, we summarize the related work and discuss their limitations. We first discuss the deficiencies of existing fault *detection* and fault localization protocols, and then briefly discuss existing work utlizing trusted computing for solving other network security problems.

8.1 Detecting the Presence of Data-Plane Attacks

A related line of research aims to only detect the presence of data-plane attacks, without providing the ability to *localize* the attacks [18, 36, 68].

For example, the protocol due to McCune et al. [68] aims to detect the presence of "Denial-of-Message" attacks in sensor networks, where a malicious sensor node drops broadcast messages sent by a base station. The base station solicits authenticated acknowledgments from a randomly selected subset of sensor nodes unpredictable to the attacker. Hence, the failure to receive authenticated acknowledgments from certain selected sensor nodes indicates the presence of packet dropping attacks. However, the protocol does not provide a mechanism to localize malicious sensor nodes that drop the messages.

Both Stealth probing [18] and PQM [36] employ secure probing techniques to detect data-plane attacks and monitor the forwarding quality for an entire end-to-end paths. These end-to-end path monitoring schemes are commonly used in conjunction with secure multi-path routing which aims to mitigate data-plane attacks. For example, both ACR [90] and Sprout [33] rely on end-to-

end monitoring to evaluate path quality and to avoid low-quality paths by switching to fail-over paths. However, being too lenient and oblivious to the faulty elements, such monitor-and-switch methods suffer from long path exploration delays, because without localizing the faulty elements, further explored paths may still contain the same faulty elements, thus resulting in exponential path exploration complexity in the worst case.

Realizing the importance of localizing data-plane attacks, researchers have recently proposed several approaches for network fault localization. As we show below, the known secure fault localization protocols are all path-based and suffer from either security vulnerabilities or high protocol overhead.

8.2 Vulnerabilities of Existing Fault Localization Schemes

Perlman first described the idea of acknowledgment-based approaches to detect data-plane adversaries and achieve robust routing in the presence of Byzantine failures [76]. However, details of how to achieve *secure* fault localization are not presented. We summarize the major security pitfalls of recent fault localization protocols as follows.

Evading and framing attacks. In ODSBR [19] and Secure Traceroute [74], the source node monitors the end-to-end loss rate of the path; and only when the observed loss rate exceeds a certain threshold, the source starts probing specific nodes in the path soliciting acknowledgments for the *subsequent* packets the source sends. However, a malicious node can safely drop packets when the probing is not activated, while behaving "normally" when probing is invoked. Hence, the source can never catch the malicious nodes nor bound the malicious dropping rate, unless the probing is always activated which incurs high overhead. In addition, ODSBR employs binary search in the probing phase for dropping localization, until the algorithm converges to a specific link. Since the binary search algorithm proceeds on each packet lost (possibly due to natural loss), in the presence of natural packet loss the algorithm either does not converge or incurs high false positives by framing benign links.

Packet modification attacks. A considerable number of fault localization protocols require each router to maintain certain traffic summaries for the received and sent packets. By periodically comparing the local traffic summaries with other routers' and checking flow conservation, such fault localization protocols can identify faulty links based on flow conservation. However, due to the challenges of efficiently authenticating packets and the traffic summaries, many of such fault localization protocols fail to authenticate packets using the traffic summaries, thus vulnerable to sophisticated packet modification attacks. For example, WATCHERS [43], AudIt [14] and Fatih [71] implement the traffic summaries using either counters or Bloom Filters [24] with no secret keys, thus remaining vulnerable to packet modification attacks. The recently proposed Network Confessional [15] also fails to prevent packet modification attacks due to the lack of efficient packet authentication.

Collusion attacks. Liu et al. propose enabling two-hop-away routers in the path to monitor each other [58] by using 2-hop acknowledgment packets. However, such a 2-hop-based detection scheme is vulnerable to colluding neighboring routers. Similarly, both Watchdog [66] and Catch [65] can identify and isolate malicious routers for wireless ad hoc networks, where a sender S verifies if the next-hop node f_i indeed forwards S's packets by *promiscuously* listening to f_i's transmission. Both Watchdog and Catch are vulnerable to collusion attacks, where a malicious node f_m drops the packets of a remote sender S (which is out of the promiscuous listening range of f_m) while the colluding neighbors in the promiscuous listening range of f_m intentionally do not report the packet dropping behavior of f_m.

8.3 Applicability and Practicality

Among the known secure proposals, the protocol due to Avramopoulos et al. [17] incurs high computational and communication overhead, because it requires acknowledgments from all routers in the path, and requires multiple digital signature generation and verification operations for *each* data packet.

Recently, Barak et al. proposed a set of fault localization protocols for the Internet [21]. How-

ever, their statistical FL protocol is mainly optimized to reduce communication overhead; and consequently achieves a rather poor *best case* detection rate on the order of 10^6 packets.

A recent proposal due to Wang et al. [89] for forwarding fault localization in sensor networks requires a special tree-like routing infrastructure where the communications take place only between a sensor node and the same trusted base station.

8.4 Trusted Computing for Network Security

Many efforts in trusted computing focus on efficient implementation of remote attestation, sealed storage, and secure boot for bootstrapping trust on commodity computers [75, 69]. A few proposals also consider utilizing trusted computing to address network security plagues [82, 40, 80]. However, BIND [82] focuses on routing security and cannot secure against raw user input and configurations. Not-a-Bot [40] leverages trusted computing and TPM to mitigate DDoS attacks but not to secure the network layer. Recently, Saroiu et al. [80] propose the design of TPM-based "trusted sensors" via remote attestation to secure a broad range of mobile applications.

Chapter 9

Conclusion

The rising demand for high-quality online services requires reliable packet delivery at the network layer. Data-plane fault localization is recognized as a promising means to this end, since it enables a source node to localize faulty links, find a fault-free path, and enforce contractual obligations among network providers. This dissertation designs, analyzes, implements, and evaluates secure and practical fault localization protocols. Instead of aiming to detect any single forwarding failure, we demonstrate that fault localization protocols can effectively limit the negative influence an adversary can inflict at the data plane, with a provable lower bound on the forwarding correctness. Based on the philosophy of limiting the adversarial activities, we develop a suite of probabilistic algorithms and leverage emerging hardware virtualization technologies, by which we dramatically reduce the protocol overhead without sacrificing security.

While we compare the efficiency of the proposed protocols early in Chapter 1 (Table 1.1), Table 9.1 further compares the effectiveness of fault localization between the proposed protocols. Clearly, PAAI incurs longer detection delay than the other protocols due to its use of packet sampling, where the fate of unsampled packets (e.g., $>$ 90% packets) cannot contribute to the monitoring process. Hence, PAAI requires more packet transmissions to achieve accurate fault localization. However, PAAI does not require any changes to the existing packet headers; thus it is most applicable to networks where the packet header cannot be changed.

ShortMAC represents a more efficient path-based protocol than PAAI, by acknowledging a set

of packets using counters in a single ACK. In addition, ShortMAC does not require loose time synchronization as PAAI and DynaFL do. By utilizing the efficient k-bit MAC authentication, ShortMAC dramatically reduces the communication overhead, and enables the use of state-efficient counters. As a path-based protocol, ShortMAC localizes data-plane faults to a specific link without the use of special hardware support (as TrueNet does), which cannot be achieved by DynaFL.

Protocol	Detection Delay	Forwarding Correctness	Precision	Global Sharing?
PAAI	3.5×10^4 pkts	95%	link	no
ShortMAC	2×10^3 pkts	95%	link	no
TrueNet	2×10^3 pkts	95%	link (software attack only)	yes
DynaFL	5×10^4 pkts	95%	1-hop neighborhood	yes

Table 9.1: Comparison of the fault localization effectiveness between the proposed protocols. The numeric values for the detection delay and guaranteed forwarding correctness are derived from simulations with the path length $d = 5$, allowed upper bound on false positive and negative rates $\delta = 0.01$, natural loss rate $\rho = 0.005$, and per-link detection threshold $T_{dr} = 0.01$.

However, 1-hop-based fault localization protocols have several fundamental advantages over path-based protocols. First, in both TrueNet and DynaFL, routers only maintain per-neighbor state, while as path-based protocols, both PAAI and ShortMAC require storing per-path state at routers. In addition, 1-hop-based fault localization protocols can support dynamic routing paths and traffic patterns. Finally, TrueNet enables secure global sharing of detection results, due to the use of trusted computing, and DynaFL achieves this property due to the involvement of a trusted centralized controller. However, these benefits of 1-hop-based protocols come at a cost. Specifically, TrueNet requires the use of trusted computing, thus potentially relying on special hardware support (such as TPM chips) and being vulnerable to hardware-based data-plane attacks. DynaFL does not rely on trusted computing, but localizes fault to a specific 1-hop neighborhood instead of a specific link.

Finally, we discuss the applicability of these protocols to different types of real networks, namely, 1) wireless sensor networks or mesh networks (or wireless multi-hop networks in general), 2) ISP networks, 3) enterprise and datacenter networks, and 4) the Internet.

Wireless multi-hop networks. First, since wireless multi-hop networks tend to have lower bandwidth resources than wired networks (and particularly, packet transmission is costly in sensor networks), DynaFL may be inapplicable due to its relatively high communication overhead for reporting the traffic summaries to the AC. In addition, since nodes in a wireless multi-hop network may be deployed at publically accessible locations (such as sensor networks), these nodes may be subject to physical compromise in which case the (physical) security of the trusted computing primitives required by TrueNet may no longer hold. Both ShortMAC and PAAI-1 can deal with physical compromise of nodes and are bandwidth-efficient. ShortMAC requires forwarding nodes to maintain per-path monitoring state, while the router state of PAAI-1 is bounded by the link bandwidth. Hence, to further decide whether ShortMAC or PAAI-1 is more state-efficient, a network administrator needs to take as inputs the network size, path distribution, and link capacity to calculate and compare the router storage cost for each protocol.

ISP networks. ISP networks are generally well managed and routers are physically protected. Hence, such a network can satisfy the security requirements of trusted computing (in other words, most attacks are through remote software exploits) and enjoy the high efficiency and small router state of TrueNet. In contrast, both PAAI and ShortMAC require per-sender key storage and path monitoring state, which may not scale in a large ISP network. Though DynaFL provides small router state and constant key storage, it only localizes data-plane faults to a 1-hop neighborhood, which requires further investigation within the suspicious neighborhood (thus incurring additional overhead).

Enterprise and datacenter networks. As already mentioned in DynaFL, modern enterprise and datacenter networks may employ fine-grained load balancing which results in both dynamic flows and paths. DynaFL is the only protocol among the proposed ones that does not require path-knowledge or path stability, and hence is most applicable to enterprise and datacenter networks.

The Internet. When performing fault localization at the scale of the Internet, we treat each Autonomous System (AS) as a single "node". Establishing "trust of code" via trusted computing across administrative domains would be troublesome; in addition, each AS physically controls its

own routers and thus can compromise the “trust” hardware to subvert the trusted computing primitives. Hence, TrueNet would be inapplicable to the Internet setting. In addition, DynaFL requires a centralized controller to manage all the nodes, which may be impractical either. PAAI requires time synchronization among different ASes, which may incur additional complexity and overhead at the scale of the Internet. Finally, ShortMAC can be deployed in the Internet, while the shared secret keys between ASes can be established via existing protocols such as Passport [60].

We anticipate that this dissertation demonstrates the possibility of achieving guaranteed forwarding correctness via secure and efficient fault localization. We hope the proposed fault localization protocols and probabilistic algorithms can serve as building blocks for constructing other secure network protocols such as secure routing and Denial-of-Service (DoS) defenses. In addition, we intend for TrueNet to be used as a case study to spark future research on leveraging trusted computing to solve other network security problems. As future work, we plan to derive theoretical performance bounds for fault localization protocols with a distribution of natural packet loss rate other than the uniform distribution. We plan to further investigate the incremental deployment and real deployment issues of fault localization protocols.

Bibliography

[1] Arbor networks: Infrastructure security survey 2010. http://www.arbornetworks.com/sp_security_report.php.

[2] Cisco security hole a whopper. http://www.wired.com/politics/security/news/2005/07/68328.

[3] Linux as IP router. http://freedomhec.pbworks.com/f/linux_ip_routers.pdf.

[4] Netperf benchmark. http://www.netperf.org/netperf/.

[5] The OpenMP API specification for parallel programming. http://openmp.org/wp/.

[6] Scalable simulation framework. http://www.ssfnet.org/.

[7] Symantec warns of router compromise. http://www.routersusa.com/symantec-warns-of-router-compromise-2.html.

[8] This project has benefited from the use of measurement data collected on the internet2 network as part of the internet2 observatory project. http://netflow.internet2.edu/.

[9] D. Achlioptas. Database-friendly random projections. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2001.

[10] Advanced Micro Devices. AMD 64 architecture programmer's manual: Volume 2: System programming, 2007.

[11] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. 1996.

[12] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proceedings of the ACM SIGCOMM*, 2008.

[13] X. Ao. Report on dimacs workshop on large-scale internet attacks. http://dimacs.rutgers.edu/Workshops/Attacks/internet-attack-9-03.pdf.

[14] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker. Loss and delay accountability

for the Internet. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2007.

[15] K. Argyraki, P. Maniatis, and A. Singla. Verifiable network-performance measurements. In *Proceedings of the ACM SIGCOMM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2010.

[16] B. Augustin, T. Friedman, and R. Teixeira. Measuring load-balanced paths in the Internet. In *Proceedings of the ACM International Measurement Conference (IMC)*, 2007.

[17] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Highly secure and efficient routing. In *Proceedings of the Conference of the IEEE Communications Society (Infocom*, 2004.

[18] I. Avramopoulos and J. Rexford. Stealth probing: Efficient data-plane security for IP routing. 2006.

[19] B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens. ODSBR: An on-demand secure Byzantine resilient routing protocol for wireless ad hoc networks. *ACM Transactions on Information and System Security*, 2008.

[20] J. Aweya. IP router architecture: An overview. *International Journal of Communication Systems*, June 2001.

[21] B. Barak, S. Goldberg, and D. Xiao. Protocols and lower bounds for failure localization in the Internet. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, 2008.

[22] M. Bellare, R. Guerin, and P. Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *CRYPTO '95*, 1995.

[23] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication(PMAC). In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, 2002.

[24] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communnications of the ACM*, 1970.

[25] K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson. Detecting disruptive routers: A distributed network monitoring approach. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1998.

[26] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Mckeown, and S. Shenker. ETHANE: Taking control of the enterprise. In *Proceedings of the ACM SIGCOMM*, 2007.

[27] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. Mckeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proceedings of the USENIX Security*, 2006.

[28] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. 2004.

[29] H. Chan and A. Perrig. Round-effcient broadcast authentication protocols for fixed topology classes. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[30] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 2004.

[31] I. Cunha, R. Teixeira, and C. Diot. Measuring and characterizing end-to-end route dynamics in the presence of load balancing. In *Proceedings of the PAM*, 2011.

[32] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 1979.

[33] J. Eriksson, M. Faloutsos, and S. V. Krishnamurthy. Routing amid colluding attackers. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2007.

[34] V. D. Gligor and P. Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes. In *Proceedings of the Fast Software Encryption*, 2001.

[35] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica. Pathlet routing. In *Proceedings of the ACM SIGCOMM*, 2009.

[36] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford. Path-quality monitoring in the presence of adversaries. In *Proceedings of SIGMETRICS*, 2008.

[37] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. 2005.

[38] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM*, 2009.

[39] T. C. Group. TPM specification version 1.2, 2009.

[40] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-bot: Improving service

availability in the face of botnet attacks. In *Proceedings of the Usenix NSDI*, 2009.

[41] K. J. Houle, G. M. Weaver, N. Long, and R. Thomas. Trends in denial of service attack technology. Technical report, CERT Coordination Center.

[42] Y.-C. Hu, A. Perrig, and M. Sirbu. SPV: Secure path vector routing for securing BGP. In *Proceedings of the ACM SIGCOMM*, 2004.

[43] J. R. Hughes, T. Aura, and M. Bishop. Using conservation of flow as a security mechanism in network protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2000.

[44] Intel Corportation. Intel trusted execution techonology – software development guide, 2008.

[45] Intel Mobility Group, Israel Development Center, Israel. Intel advanced encryption standard (AES) instructions set, 2010. http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/.

[46] J.Eidson and K. Lee. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In Sensors for Industry Conference, 2nd ISA/IEEE., 2002.

[47] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM*, 2009.

[48] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. van Wesep Thomas Anderson, and A. Krishnamurthy. Reverse traceroute. In *Proceedings of the USENIX NSDI*, 2010.

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

[50] S. Kent, C. Lynn, J. Mikkelson, and K. Seo. Secure border gateway protocol (S-BGP) — real world performance and deployment issues. In *Proceedings of the NDSS*, 2000.

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

[52] M. E. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham. Encrypting the Internet. In *Proceedings of the ACM SIGCOMM*, 2010.

[53] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb. 1997.

[54] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs. R-BGP: Staying connected in a connected world. In *Proceedings of the USENIX NSDI*, 2007.

[55] C. Labovitz, A. Ahuja, and M. Bailey. Shining light on dark address space. Technical report, Arbor Networks.

[56] P. Laskowski and J. Chuang. Network monitors and contracting systems: competition and innovation. In *Proceedings of the ACM SIGCOMM*, 2006.

[57] F. Le, G. G. Xie, and H. Zhang. Theory and new primitives for safely connecting routing protocol instances. In *Proceedings of the ACM SIGCOMM*, 2010.

[58] K. Liu, J. Deng, P. K. Varshney, and K. Balakrishnan. An acknowledgement-based approach for the detection of routing misbehavior in MANETs. *IEEE Transactions on Mobile Computing*, 2007.

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

[60] X. Liu, X. Yang, D. Wetherall, and T. Anderson. Efficient and secure source authentication with packet passports. In *Proceedings of the USENIX SRUTI*, 2006.

[61] X. Liu, X. Yang, and Y. Xia. NetFence: Preventing Internet Denial of Service from Inside Out. In *Proceedings of the ACM SIGCOMM*, 2010.

[62] Y. Lu, G. Shou, Y. Hu, and Z. Guo. The research and efficient FPGA implementation of Ghash core for GMAC. In *Proceedings of the International Conference on E-Business and Information System Security*, 2009.

[63] M. Luk, A. Perrig, and B. Whillock. Seven cardinal properties of sensor network broadcast authentication. In *Proceedings of the ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN 2006)*, 2006.

[64] A. Lysyanskaya, R. Tamassia, and N. Triandopoulos. Multicast authentication in fully adversarial networks. In *In Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[65] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Sustaining cooperation in multi-hop wireless networks. In *Proceedings of the Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[66] S. Marti, T. J. Giuli, K. Lai, and M. Baker. Mitigating routing misbehavior in mobile Ad Hoc networks. In *Proceedings of the ACM Mobicom*, 2000.

[67] M.Bellare, R.Canetti, and H.Krawczyk. Pseudorandom functions revisited: the cascade construction and its concrete security. In *Proceedings of the IEEE FOCS*, 1996.

[68] J. McCune, E. Shi, A. Perrig, and M. K. Reiter. Detection of denial-of-message attacks on sensor network broadcasts. In *Proceedings of IEEE Symposium on Security and Privacy*, 2005.

[69] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[70] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal TCB code execution (extended abstract). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[71] A. T. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage. Fatih: detecting and isolating malicious routers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[72] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path splicing. In *Proceedings of the ACM SIGCOMM*, 2008.

[73] S. L. Murphy and M. R. Badger. Digital signature protection of the OSPF routing protocol. In *Proceedings of the NDSS*, 1996.

[74] V. N. Padmanabhan and D. R. Simon. Secure traceroute to detect faulty or malicious routing. *SIGCOMM Computer Communication Review (CCR)*, 2003.

[75] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[76] R. Perlman. *Network Layer Protocol with Byzantine Agreement*. PhD thesis, 1988.

[77] A. Perrig, R. Canetti, D. Song, and D. Tygar. The TESLA broadcast authentication protocol. *RSA Cryptobytes*, 2002.

[78] H. Pucha, Y. Zhang, Z. M. Mao, and Y. C. Hu. Understanding network delay changes caused by routing events. In *Proceedings of the ACM SIGMETRICS*, 2007.

[79] B. Raghavan and A. C. Snoeren. A system for authenticated policy-compliant routing. In *Proceedings of the ACM SIGCOMM*, 2004.

[80] S. Saroiu and A. Wolman. I am a sensor, and i approve this message. In *Proceedings of the HotMobile*, 2010.

[81] A. Satoh, T. Sugawara, and T. Aoki. High-performance hardware architecture for Galois counter mode. *IEEE Transactions on Computers*, 2009.

[82] E. Shi, A. Perrig, and L. V. Doorn. BIND: A time-of-use attestation service for secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[83] J. Song, R. Poovendran, J. Lee, and T. Iwata. The AES-CMAC Algorithm. RFC 4493 (Informational), June 2006.

[84] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proceedings of the ACM SIGCOMM*, 2002.

[85] E. T. Krovetz. UMAC: Message Authentication Code using Universal Hashing. RFC 4418, 2006.

[86] H. Technology. AES based authentication cores. `http://www.heliontech.com/aes_auth.htm`.

[87] R. Thomas. ISP security BOF, nanog 28. `http://www.nanog.org/mtg-0306/pdf/thomas.pdf`.

[88] M. Thorup and Y. Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. *Proceedings of the SODA*, 2004.

[89] C. Wang, T. Feng, J. Kim, G. Wang, and W. Zhang. Catching packet droppers and modifiers in wireless sensor networks. In *Proceedings of the IEEE SECON*, 2009.

[90] D. Wendlandt, I. Avramopoulos, D. Andersen, and J. Rexford. Don't secure routing protocols, secure data delivery. In *Proceedings of the ACM Workshop on Hot Topics in Networks (Hotnets-V)*, 2006.

[91] W. Xu and J. Rexford. MIRO: Multi-path interdomain routing. In *Proceedings of the ACM SIGCOMM*, 2006.

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

[93] B. Yang, R. Karri, and D. A. McGrew. A high-speed hardware architecture for universal message authentication code. *IEEE Journal on Selected Areas in Communications*, 24(10):1831–1839, 2006.

[94] X. Yang, D. Clark, and A. W. Berger. Nira: a new inter-domain routing architecture. *IEEE/ACM Transactions on Networking*, 2007.

[95] X. Yang and D. Wetherall. Source selectable path diversity via routing deflections. In *Proceedings of the ACM SIGCOMM*, 2006.

[96] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen. SCION: Scalability, control, and isolation on next-generation networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[97] X. Zhang, A. Jain, and A. Perrig. Packet-dropping adversary identification for data plane security. In *Proceedings of the ACM SIGCOMM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2008.

[98] X. Zhang and A. Perrig. Correlation-resilient path selection in multi-path routing. In *Proceedings of the IEEE Global Communications Conference (Globecom'10)*, Dec. 2010.

[99] Y. Zhao, Y. Chen, and D. Bindel. Towards unbiased end-to-end network diagnosis. In *Proceedings of the ACM SIGCOMM*, 2006.

Appendix A

Proofs for PAAI

A.1 Proof of Theorem 7

In the following, we analyze the full-ack and PAAI-1 protocols together, and PAAI-2 separately. We prove for the general case where the adversary controls z malicious links in a forwarding path, and an upper bound of α in Definition 5 can be derived by setting $z = 1$.

Full-ack and PAAI-1. Since the onion report used in the full-ack and PAAI-1 schemes can be used to locate a specific link for each lost packet, *under converged condition* each malicious link can at most drop T_{dr} fraction of packets without being detected. This in turn implies our results in Theorem 7 for the full-ack and PAAI-1 schemes.

PAAI-2. First note that the score difference $\Delta_i = |s_{i+1} - s_i|$ is given by

$$\Delta_i = \eta_{i+1} \cdot \{1 - [\prod_{y=0}^{i}(1 - \rho_y^*)]^3\}, \tag{A.1}$$

where η_{i+1} is the number of times that f_{i+1} is selected, and ρ_i^* is the average drop rate of link l_i. Based on the values of Δ_i and η_i known to $\mathbb{S}$, $\mathbb{S}$ can compute the average drop rate ρ_i^* by (let $C = \prod_{y=0}^{k-1}(1 - \rho_y^*)^3$):

$$\rho_k^* = \begin{cases} 1 - (1 - \frac{\Delta_0}{\eta_1})^{\frac{1}{3}}, & k = 0 \\ \rho_k^* = 1 - [\frac{1}{C}(1 - \frac{\Delta_k}{\eta_{k+1}})]^{\frac{1}{3}} & , \ k \geq 1. \end{cases} \tag{A.2}$$

Now we establish an end-to-end drop rate threshold ψ_{th}. We must ensure that the actual end-to-end drop rate exceeds ψ_{th} only when at least one malicious link drops more than T_{dr} percentage of packets (where T_{dr} is the per-link drop rate threshold). Thus in the worst case (with the most natural packet drop loss, i.e., using T_{dr} as the per-link drop rate), we can compute

$$\psi_{th} = 1 - (1 - T_{dr})^{2d}. \tag{A.3}$$

If each link l_i has a drop rate $\rho_i^* < T_{dr}$, the end-to-end drop rate ψ_d is given by:

$$\psi_d = 1 - [\prod_{i=0}^{d-1}(1 - \rho_i^*)]^2 < 1 - (1 - T_{dr})^{2d} \Rightarrow \psi_d < \psi_{th}. \tag{A.4}$$

Thus when $\psi_d > \psi_{th}$, there must be at least one malicious link. Then $\mathbb{S}$ derives the individual drop rate ρ_i^* of each link l_i by using Equation A.2. By comparing each ρ_i^* with T_{dr}, $\mathbb{S}$ can identify the malicious links. Note that in the converged condition, there is no false negative, while the false positive is given by Theorem 10.

Now we compute the maximum end-to-end drop rate that an adversary can cause without being detected, i.e., without causing $\psi_d > \psi_{th}$. Suppose there are z malicious links with the drop rate $\rho_{\mathcal{M}_1}^*, \ldots, \rho_{\mathcal{M}_z}^* > T_{dr}$. Given the fixed threshold ψ_{th}, when the malicious links can cause maximum drop rate with $\psi_d < \psi_{th}$, we have:

$$1 - (1 - \rho)^{2(d-z)} \cdot [\prod_{k=1}^{z}(1 - \rho_{\mathcal{M}_k}^*)]^2 = 1 - (1 - T_{dr})^{2d}. \tag{A.5}$$

Therefore, z malicious links can drop at most

$$1 - [\prod_{k=1}^{z}(1 - \rho_{\mathcal{M}_k}^*)]^2 = 1 - \frac{(1 - T_{dr})^{2d}}{(1 - \rho)^{2(d-z)}}. \tag{A.6}$$

percentage of traffic without being detected.

A.2 Proof of Corollary 8

Suppose that a malicious link $l_{\mathcal{M}_k}$ $(k = 1, 2, \ldots, z)$ drops data, probe and ack packets at different rates, denoted by $\mu_{\mathcal{M}_k}$, $\nu_{\mathcal{M}_k}$ and $\omega_{\mathcal{M}_k}$ respectively. Given the fixed threshold ψ_{th}, the malicious

links can cause the maximum drop rate when $\psi_d < \psi_{th}$, yielding:

$$1-(1-\rho)^{2(d-z)}\cdot\prod_{i=1}^{z}(1-\mu_{\mathcal{M}_i})(1-\nu_{\mathcal{M}_i})(1-\omega_{\mathcal{M}_i})<\psi_{th}. \tag{A.7}$$

Therefore, the adversary can drop at most

$$1-\prod_{i=1}^{z}(1-\mu_{\mathcal{M}_i})(1-\nu_{\mathcal{M}_i})(1-\omega_{\mathcal{M}_i})=1-\frac{(1-T_{dr})^{2d}}{(1-\rho)^{2(d-z)}}. \tag{A.8}$$

fraction of packets without being detected. This yields the same result as Theorem 7.

A.3 Proof of Corollary 9

The results are straightforward for full-ack and PAAI-1. For PAAI-2, leveraging $(1-x)^n = 1-nx$ when $x \to 0$ and neglecting the second order ρ^2 term, we can transform the formula of ζ in Theorem 7 as:

$$\zeta \doteq 2d\epsilon+\rho\cdot\big(4d^2\epsilon+z(2-4d\epsilon)\big). \tag{A.9}$$

This proves that ζ increases proportionally to ρ. Next, we show that the optimal strategy of the adversary (to drop the maximum traffic with z compromised links) is to deploy only one malicious link for one path. We give a sketch of the proof by showing two extreme cases to illustrate the intuition:

In one extreme case where the adversary deploys all z compromised links on one path, the adversary can drop at most:

$$\zeta_1 = 2d\epsilon+\rho\cdot\big(4d^2\epsilon+z(2-4d\epsilon)\big). \tag{A.10}$$

fraction of packets as calculated above. In the other extreme case, where the adversary deploys one compromised link for one path (thus z paths contain a comprised link), the adversary can drop at most:

$$\zeta_2 = z\Big(2d\epsilon+\rho\cdot\big(4d^2\epsilon+1\cdot(2-4d\epsilon)\big)\Big) \tag{A.11}$$

fraction packets. Obviously we have

$$\zeta_1 \leq \zeta_2 \leq z \cdot \zeta_1. \tag{A.12}$$

A.4 Proof of Theorem 10

In the following proof, we first study how many packet transmissions are required to estimate the drop rate of a single link l_i within a certain *accuracy interval*. Suppose that the true value of drop rate of l_i is $\rho_i^{'}$, and the estimated drop rate of l_i is ρ_i^*. We compute the number of packets needed to achieve a $(\epsilon_{\rho_i^*}, \delta)$-accuracy for ρ_i^*:

$$Pr(|\rho_i^* - \rho_i^{'}| > \epsilon_{\rho_i^*}) < \delta, \tag{A.13}$$

i.e., with probability $1 - \delta$ the estimated ρ_i^* is within

$$(\rho_i^{'} - \epsilon_{\rho^*}, \rho_i^{'} + \epsilon_{\rho^*}). \tag{A.14}$$

Then we compute the total number of packets needed to achieve a $(\epsilon_{\rho_i^*}, \delta)$-accuracy for *every* link's ρ_i^*. In the following we analyze the full-ack scheme and the PAAI protocols in turn.

Full-ack. We first study a given link l_i. Let ρ_i^* be the estimated drop rate of link l_i, and p_i be the observed *conditional* probability that link l_i correctly forwards both a data packet and the returning ack *given that* a data packet reaches the upstream end of l_i, i.e., node f_i in Figure 2.1. Thus we have:

$$p_i = (1 - \rho_i^*)^2. \tag{A.15}$$

We define each time a data packet reaches node f_i as a random trial of l_i (or trial for l_i, in short). Then using *Maximum Likelihood Estimation* of p_i^* and *Hoeffding's inequality*, we have:

$$Pr(|p_i - p_i^*| > \epsilon_{p_i}) < 2e^{-2N_i\epsilon_{p_i}^2} \;\; \Rightarrow \;\; N_i = \frac{\ln(\frac{2}{\delta})}{2\epsilon_{p_i}^2}. \tag{A.16}$$

Note that we are not interested in ϵ_{p_i}, but $\epsilon_{\rho_i^*}$ instead. However, ρ_i^* cannot be directly estimated,

but must be derived from Equation A.15, i.e.:

$$\rho_i^* = 1 - p_i^{\frac{1}{2}}. \tag{A.17}$$

Given the error ϵ_{p_i} of p_i, we can further derive the error $\epsilon_{\rho_i^*}$ of ρ_i^* using the *Uncertainty Propagation Rule*:

$$\epsilon_{\rho_i^*} = |\frac{\partial \rho_i^*}{\partial p_i}\epsilon_{p_i}| = \frac{1}{2}p_i^{-\frac{1}{2}} \cdot \epsilon_{p_i} \;\;\Rightarrow\;\; \epsilon_{p_i} = 2\epsilon_{\rho_i^*} \cdot p_i^{\frac{1}{2}}. \tag{A.18}$$

Combining Equations A.15, B.13 and A.18, and given $\epsilon_{\rho_i^*} \leq \epsilon$ we have:

$$N_i = \frac{\ln(\frac{2}{\delta})}{2\left(2\epsilon_{\rho_i^*} \cdot p_i^{\frac{1}{2}}\right)} = \frac{\ln(\frac{2}{\delta})}{8\epsilon_{\rho_i^*}^2 \cdot (1-\rho)^2} \geq \frac{\ln(\frac{2}{\delta})}{8\epsilon^2 \cdot (1-\rho)^2}. \tag{A.19}$$

Now we compute the number of packets needed to give an estimate with (ϵ, δ)-accuracy for every link in a given path. When each packet transmitted by the source can reach node f_{d-1}, it provides a trial for every link l_i. Therefore, transmitting N_i packets to f_{d-1} also suffices to give other links enough trials, which requires

$$N_{d-1}\frac{1}{(1-T_{dr})^d} = \frac{\ln(\frac{2}{\delta})}{8\epsilon^2 \cdot (1-\rho)^{2+d}} \tag{A.20}$$

total packets transmitted from the source.

PAAI protocols The result for PAAI-1 is straightforward, therefore we now focus on PAAI-2. Let ρ_k^* be the estimated drop rate of link l_k. Let the event that node f_{k+1} is *selected* be a random trial for l_k (or trial for l_k for short). Let p_k be the observed *conditional* probability that node f_{k+1} fails to ack when f_{k+1} is *selected*, and p_k^* be the true value of p_k. Similar to the proof of full-ack scheme above, using Hoeffding's inequality we can have:

$$Pr(|p_k - p_k^*| > \epsilon_{p_k}) < 2e^{-2N_k\epsilon_{p_k}^2} \;\;\Rightarrow\;\; N_k = \frac{\ln(\frac{2}{\delta})}{2\epsilon_{p_k}^2}. \tag{A.21}$$

Again, since we are not interested in ϵ_{p_k}, but instead $\epsilon_{\rho_k^*}$, we derive $\epsilon_{\rho_k^*}$ from ϵ_{p_k} in the following. Since $p_k = \frac{\Delta_k}{\eta_{k+1}}$, leveraging Equation A.2 and simplifying it using $(1-x)^n = 1 - nx$ when $x \to 0$

and neglecting high order (≥ 2) terms of x, we can have:

$$\rho_k^* = \frac{1}{3}p_k - \sum_{y=0}^{k-1}\rho_y^*. \tag{A.22}$$

Using the *Uncertainty Propagation Rule* further yields:

$$\epsilon_{\rho_k^*}^2 = \sum_{y=0}^{k-1}\big(\frac{\partial\rho_k^*}{\partial\rho_y^*}\cdot\epsilon_{\rho_y^*}\big)^2 + \big(\frac{\partial\rho_k^*}{\partial p_k}\cdot\epsilon_{p_k}\big)^2 = \sum_{y=0}^{k-1}\epsilon_{\rho_y^*}^2 + \frac{1}{9}\epsilon_{p_k}^2. \tag{A.23}$$

Solving the above recursion and leveraging Equation A.21, we further have:

$$N \doteq 2^d\frac{\ln(\frac{2}{\delta})}{18\epsilon^2}. \tag{A.24}$$

Now we compute the total number of packets needed to give a (ϵ, δ)-accuracy estimate for every link's drop rate in a given path. If we abstract a random trial for link l_k as *coupon* k, then a path with length $\mathbb{D}$ has $\mathbb{D}$ different coupons. The problem is to compute the expected wait time (number of trials) to gather N copies for each coupon k. When $N = 1$ the problem reduces to the classic *Coupon Collector* problem, which has an expected wait time $O(d \cdot \log(d))$. With $N \neq 1$, the wait time has a simple upper bound:

$$O(N \cdot d \cdot \log(d)) = O(2^d\frac{\ln(\frac{2}{\delta})}{18\epsilon^2} \cdot d \cdot \log(d)). \tag{A.25}$$

This proves the theorem.

A.5 Proof of Corollary 11

Since the proof of this corollary is straightforward, we only give a proof sketch here. Given that the natural loss rate $\rho << 1$ in practice, we can approximate $(1-\rho)^{2+d} \approx 1$ for calculating N_1 and N_2 in Theorem 10. We can then study the influence of each parameter by taking a partial derivative on each parameter from N_1 and N_2.

Appendix B

Proofs for ShortMAC

B.1 Proof of Lemma 13

Recall from Section 5.3 that in ShortMAC, the source finds the first C_i^{bad} such that $C_i^{bad} > T_{in}$, and identifies link l_i as malicious. In this proof, we first derive the upper bound β of malicious packet injection (which is based on T_{in}) according to the upper bound δ of false negative rate. Then we calculate the injection threshold T_{in} given the false positive upper bound δ.

With k-bit MACs, when f_{i-1} receives a fake packet, the probability that C_{i-1}^{bad} will be increased is $q = \frac{2^k-1}{2^k}$, since the adversary can only randomly generate a k-bit string for the fake packet without knowledge of the secret keys of other (benign) routers. The probability that C_i^{bad} will be increased is $q(1-q)$.

Malicious Injection Bound. WLOG, suppose f_m is a malicious router and f_{m+1} is benign (there can be other malicious routers between the source and f_m). Suppose the malicious routers between the source and f_m (including f_m) inject y packets on link l_{m+1}. Then whether l_{m+1} will be detected depends on the value of C_{m+1}^{bad}, and the false negative rate $\mathbb{P}_{fn}$ is given by:

$$\begin{aligned}\mathbb{P}_{fn} &= \mathbb{P}(C_{m+1}^{bad} < T_{in}) \\ &= \mathbb{P}\big((q-\epsilon)y < T_{in}\big) \qquad \text{(B.1)} \\ &\leq 2e^{-2y\left(q-\frac{T_{in}}{y}\right)^2} \text{(Hoeffding's inequality)},\end{aligned}$$

where ϵ is the deviation and $0 \leq \epsilon \leq q$. To achieve the desired upper bound $\mathbb{P}_{fn} \leq \delta$, we set the threshold β such that

$$2e^{-2\beta(q-\frac{T_{in}}{\beta})^2} = \delta. \tag{B.2}$$

Solving for β gives:

$$\beta = \frac{T_{in}}{q} + \frac{\sqrt{\left(\ln\frac{2}{\delta}\right)^2 + 8qT_{in}\ln\frac{2}{\delta}} + \ln\frac{2}{\delta}}{4q^2}. \tag{B.3}$$

(B.3) implies that if the adversary injects more than β packets on a single link l_{m+1}, C^{bad}_{m+1} will exceed T_{in} and l_{m+1} will be detected with a high probability $\geq 1-\delta$ (or a false negative rate lower than δ).

Injection Detection Threshold. WLOG, suppose f_m is a malicious router and f_{m+1} is benign (there can be other malicious routers between the source and f_m). Suppose the malicious routers between the source and f_m (including f_m) inject y packets on link l_{m+1}. False positives occur when $C^{bad}_{m+1} < T_{in}$ but $C^{bad}_i \geq T_{in}$ (where $i \geq m+2$). (WLOG, suppose f_{i-1} and f_i are honest.) Hence, a benign link l_i is falsely accused, and the false positive rate $\mathbb{P}_{fp}$ is:

$$\begin{aligned} \mathbb{P}_{fp} &:= \sum_{i=m+2}^{d} \mathbb{P}(C^{bad}_{m+1} < T_{in}, C^{bad}_i \geq T_{in} | l_i \text{ benign}) \\ &\leq d \cdot \mathbb{P}(C^{bad}_{m+1} < C^{bad}_{m+2}). \end{aligned} \tag{B.4}$$

The actual C^{bad}_{m+1} and C^{bad}_{m+2} values can be represented by:

$$\begin{aligned} C^{bad}_{m+1} &= (q-\epsilon_1)\cdot y \\ C^{bad}_{m+2} &= \big(q(1-q)+\epsilon_2\big)\cdot y. \end{aligned} \tag{B.5}$$

If we can bound

$$\epsilon_1 = \epsilon_2 = \epsilon \leq \frac{p^2}{2}, \tag{B.6}$$

then we can guarantee that $C^{bad}_{m+1} > C^{bad}_{m+2}$. Therefore, we have:

$$\begin{aligned}\mathbb{P}_{fp} &\leq 1 - \mathbb{P}(\epsilon \leq \frac{q^2}{2}) \\ &= \mathbb{P}(\epsilon > \frac{q^2}{2}) \\ &\leq 2e^{-2y(\frac{q^2}{2})^2}.\end{aligned} \tag{B.7}$$

Note that in (B.7), we leverage Hoeffding's inequality and the fact $y \geq T_{in}$ in the false positive cases.

To achieve the desired upper bound $\mathbb{P}_{fp} \leq \delta$, we set the threshold T_{in} such that

$$2e^{-2T_{in}(\frac{q^2}{2})^2} = \delta. \tag{B.8}$$

Solving for T_{in} gives

$$T_{in} = \frac{2\ln\frac{2d}{\delta}}{q^4}. \tag{B.9}$$

B.2 Proof of Lemma 14

Drop Detection Threshold and Detection Space. False positives arise when the *observed* drop rate of a benign link l_i, denoted by ρ^*_i, exceeds the drop detection threshold T_{dr}. To bound the total false positive rate below δ, it is sufficient to ensure that each ρ^*_i may exceed T_{dr} with a probability $\delta_i = \frac{\delta}{d}$ (since we need to ensure the overall false positive rate $\sum_i \delta_i \leq \delta$), i.e., $\mathbb{P}(\rho^*_i > T_{dr}) < \frac{\delta}{d}$, which is equivalent to:

$$\mathbb{P}(\rho^*_i - \rho > T_{dr} - \rho) < \frac{\delta}{d}. \tag{B.10}$$

By using Hoeffding's inequality, we have:

$$\begin{aligned}&\mathbb{P}\left(\rho^*_i - \rho > T_{dr} - \rho\right) < 2e^{-2C^{good}_{i-1}(T_{dr}-\rho)^2} \\ &\Rightarrow \; C^{good}_{i-1} \geq \frac{\ln(\frac{2d}{\delta})}{2(T_{dr}-\rho)^2}.\end{aligned} \tag{B.11}$$

Recall that the check-dropping procedure will detect the malicious link with excessive drop rate closest to the source, denoted by l_m. So we need to guarantee $C_i^{good} \geq \frac{\ln(\frac{2d}{\delta})}{2(T_{dr}-\rho)^2}$ for any $i < m$. Since we also have

$$C_i^{good} \geq N(1-T_{dr})^i \text{ for } i < m, \tag{B.12}$$

we get:

$$N = \frac{\ln(\frac{2d}{\delta})}{2(T_{dr}-\rho)^2(1-T_{dr})^d}. \tag{B.13}$$

Analogously, we can also calculate the false negative rate, which yields the same result.

Malicious Dropping Bound. Suppose a malicious node f_m closest to the source receives C_m^{recv} data packets, but claims that it receives C_m^{good} data packets, and drops x fraction of the received C_m^{good} data packets on l_{m+1}. We first have the following facts:

$$\begin{aligned} C_m^{recv} &\leq C_{m-1}^{good} \\ C_{m+1}^{good} &= (1-x)C_m^{recv} + \beta. \end{aligned} \tag{B.14}$$

To make neither of its incident links undetected, f_m must manage to satisfy:

$$\begin{aligned} \frac{C_m^{good}}{C_{m-1}^{good}} &\geq 1 - T_{dr} \\ \frac{C_{m+1}^{good}}{C_m^{good}} &\geq 1 - T_{dr}, \end{aligned} \tag{B.15}$$

which yields

$$\begin{aligned} C_{m-1}^{good} &\geq (1-T_{dr})^{m-1}N \\ &\geq (1-T_{dr})^d N. \end{aligned} \tag{B.16}$$

Solving (B.14), (B.15) and (B.16), we have

$$\begin{aligned} x &\leq 1 - (1-T_{dr})^2 + \frac{\beta}{N(1-T_{dr})^d} \\ &= \alpha. \end{aligned} \tag{B.17}$$

B.3 Proof of Theorem 15

$(\alpha, \beta)_\delta-$Statistical Security can directly follow Lemma 13 and Lemma 14. In the following, we will prove $(\Omega, \theta)-$Guaranteed Forwarding Correctness.

Given N and δ, we can set the drop detection threshold T_{dr} from Lemma 14 and the injection bound β from Lemma 13. Let η^{fake} denote the fake data packets the destination has received but not detected yet, and η^{leg} denote the legitimate data packets the destination has received out of N data packets from the source. Then we have:

$$\begin{aligned} \theta &= \frac{\eta^{leg}}{N} \\ &= \frac{C_{d+1}^{good} - \eta^{fake}}{N}. \end{aligned} \tag{B.18}$$

When no fault is detected in the identify stage, it satisfies:

$$\begin{aligned} &C_{d+1}^{good} \geq (1 - T_{dr})^d N \\ &\eta^{fake} \leq \beta. \end{aligned} \tag{B.19}$$

By (B.18) and (B.19), we have

$$\theta = (1 - T_{dr})^d - \frac{\beta}{N}. \tag{B.20}$$

Finally, we can integrate ShortMAC with routing as follows. The control plane first provides a routing path p for the source S, and then avoids faulty links using feedback (fault localization results) from the data plane. In this way, ShortMAC enables the source to identify the malicious links that reside in previously explored paths. In a network with Ω malicious links, the source can bypass *at least one* of the malicious links after each epoch until a working path is found, resulting in an exploration of at most Ω epochs to find a working path.

Appendix C

Proof for DynaFL

C.1 Proof of Property 2 for Sketch

A sketch function $\mathcal{F}$ over a set of elements $\mathbb{S} = \{p_1, p_2, \ldots, p_n\}$ can be implemented in a "streaming" mode using a hash function h [36], where:

$$h(p_i) \rightarrow \vec{v}_i \tag{C.1}$$

and $\vec{v}_i$ denotes a vector. More specifically:

$$\mathcal{F}(\mathbb{S}) = \mathcal{F}(\{p_1, p_2, \ldots, p_n\}) = h(p_1) + h(p_2) + \ldots + h(p_n) \tag{C.2}$$

Hence, given two packet streams $\mathbb{S} = \{p_1, p_2, \ldots, p_n\}$ and $\mathbb{S}^{'} = \{p_1^{'}, p_2^{'}, \ldots, p_{n'}^{'}\}$, we have:

$$\begin{aligned} \mathcal{F}(\mathbb{S} \cup \mathbb{S}^{'}) &= \mathcal{F}(\{p_1, \ldots, p_n, p_1^{'}, \ldots, p_{n'}^{'}\}) \\ &= h(p_1) + \ldots + h(p_n) + h(p_1^{'}) + \ldots + h(p_{n'}^{'}) \end{aligned} \tag{C.3}$$

and:

$$\begin{aligned} \mathcal{F}(\mathbb{S}) + \mathcal{F}(\mathbb{S}^{'}) &= \mathcal{F}(\{p_1, \ldots, p_n\}) + \mathcal{F}(\{p_1^{'}, \ldots, p_{n'}^{'}\}) \\ &= h(p_1) + \ldots + h(p_n) + h(p_1^{'}) + \ldots + h(p_{n'}^{'}) \end{aligned} \tag{C.4}$$

From Equations C.3 and C.4 we can see that: when $\mathcal{F}(\mathbb{S}) \cup \mathcal{F}(\mathbb{S}^{'})$ is defined as $\mathcal{F}(\mathbb{S}) + \mathcal{F}(\mathbb{S}^{'})$, we have $\mathcal{F}(\mathbb{S} \cup \mathbb{S}^{'}) = \mathcal{F}(\mathbb{S}) \cup \mathcal{F}(\mathbb{S}^{'})$, thus proving Property 2 for Sketch.