A Cryptographic Study of Secure Fault Detection in the Internet *

Sharon Goldberg, David Xiao, Boaz Barak, and Jennifer Rexford
Princeton University, Princeton, NJ 08544

March 5, 2007. Last Update, Oct. 1, 2007.

Abstract

Mechanisms for measuring data-path quality and identifying locations where packets were dropped are crucial for informing routing decisions. If such mechanisms are to be reliable, they must be designed to prevent ASes from 'gaming' measurements to their advantage (*e.g.* by hiding packet loss). This paper is a rigorous cryptographic study of *secure fault detection*, an end-to-end technique that allows a sender to detect whether or not her traffic arrived unaltered at a receiver, even when some of the nodes on the the data path maliciously attempt to bias measurements. We explore mechanisms for accurately detecting packet loss events on a data path in the presence of both benign loss (congestion, link failure) and active adversaries (ASes motivated by malice or greed). We do not advocate a specific network architecture. Instead, we use rigorous techniques from theoretical cryptography to present new protocols and negative results that can guide the placement of measurement and security mechanisms in future networks.

Our major contributions are: (1) Negative results that prove that any detection mechanism requires secret keys, cryptography and storage at every participating node, and (2) *Pepper Probing* and *Salt Probing*: two efficient protocols for accurate end-to-end detection of packet loss on a path, even in the presence of adversaries. In Pepper Probing, Alice and Bob sample packets in a coordinated manner with a cryptographic hash function. To reduce the key-management overhead, Salt Probing uses a timing strategy to extend Pepper Probing to the public-key setting without compromising the efficiency of the protocol.

*This is a Princeton University Department of Computer Science Technical Report. It was previously part of "A Cryptographic Study of Secure Internet Measurement", Princeton University Department of Computer Science Tech Report, May, 2007.

Contents

1 Introduction **3**

2 Our setting **4**

3 Definitions and Notations **5**
3.1 Security Definition 6
3.1.1 Per-Packet Definition 7
3.1.2 Statistical Definition 7
3.2 Properties of Security Definition 8

4 Negative Results **9**
4.1 Keys are Necessary for Fault Detection 9
4.2 Cryptography is Necessary for Fault Detection 10
4.3 Storage is Necessary for Fault Detection 11

5 Per-Packet-Ack: A Simple FD Protocol **12**

6 Why use Statistical Fault Detection based on Sampling? **14**

7 Pepper Probing: Symmetric-Key Statistical FD **15**
7.1 Description of Pepper Probing 16
7.2 Security of Pepper Probing 17
7.3 Efficiency of Pepper Probing 19

8 Salt Probing: Public-Key Statistical FD **19**
8.1 Description of Salt Probing 19
8.2 Timing for Salt Probing 22
8.3 Security of Salt Probing 23
8.4 Efficiency of Salt Probing 24

9 Comparison of Fault Detection Protocols **25**

10 Conclusions and Implications **26**

1 Introduction

The Internet's best-effort service model provides neither *a priori* guarantees of packet delivery nor *post hoc* information about the fate of dropped packets. However, tools for measuring data-path quality and identifying locations where packets were dropped are crucial for informing routing decisions at the edge of the network (*e.g.* in source routing, intelligent route control, and multipath routing). Moreover, [14] recently argued that the Internet industry's resistance to evolution *cannot* be overcome without an accountability framework that measures path quality and then holds ISPs responsible for poor path performance. The combination of an accountability framework and tools for intelligent routing could counter the growing trend of commoditization on the Internet by giving ISPs an economic incentive to upgrade their networks in order to attract customers [6, 14, 1]. In this paper we explore methods that empower the edge of the network to measure data path quality. Because the network layer is typically responsible for making routing decisions, we focus specifically on tools that source edge networks (*i.e.* stub ASes or edge routers) can use to perform *fault detection (FD)* to detect faults on a data path. We say that a fault occurs when a packet sent by a source edge network fails to arrive unaltered at its destination edge network within a reasonable amount of time.

The presence of adversaries: The design of robust measurement schemes is complicated by the presence of parties on the network that have a *strong incentive to bias path quality measurements*. Consider these natural scenarios:

- ISPs may provide poor service to selected classes of traffic in order to cut costs. ISPs have an economic incentive to prevent their customers from taking their business elsewhere or from demanding accountability for service level agreement (SLA) violations. As such, greedy ISPs may attempt to bias measurements to prevent edge networks from detecting degraded path quality.

- A malicious router or AS, corrupted by a remote attacker or disgruntled network operator, may advertise routes through himself so that he can selectively block or tamper with certain flows. For example, a remote attacker may tamper with packets containing information about a corporation's stock price, sent by a website like CNN.com. The hacked router has a strong incentive to bias measurements so that his malicious behavior can continue undetected.

- A router may 'benignly' drop packets due to malfunction, misconfiguration or excessive congestion. For example, an MTU (Maximum Transmission Unit) mismatch along the path might cause a router to drop large packets, while continuing to forward small packets (*e.g.* from ping and traceroute). Rather than making assumptions about the nature of packet loss events, we can instead assume that packet loss occurs in the "worst possible way", *i.e.* according to a pattern that introduces bias in our measurements.

As such, we avoid making *ad hoc* assumptions about the good behaviour of ASes or routers on the data path. Instead, *we assume that a source edge-network can trust only the destination edge-network at the end of the data path she is measuring, while the intermediate nodes on the path may adversarially affect her measurements*. We focus on finding efficient protocols and minimal resource requirements for fault detection; to avoid burdening our schemes with extra machinery, our model does not consider traditional security issues like preventing faults, providing confidentiality, or protecting against traffic analysis or denial-of-service attacks.

Accountability, Security and Measurement: Our work lies at the intersection of the literature on network accountability, data-plane security, and path-quality measurement. The literature on accountability [1, 14] focuses on ASes' economic motivations for making decisions, but it does not explicitly consider mechanisms to prevent ASes from 'gaming' an accountability framework to their economic advantage. On the other hand, much of the work [19, 2, 4, 11, 18, 16] on data-plane security implicitly assumes that faults caused by adversarial nodes should be detected on a *per-packet* basis (*i.e.* when just one packet is dropped [19,2,4,11]). Here we consider benign faults (*i.e.* congestion, link or node failures) alongside adversarial behaviour, which makes our models simultaneously more realistic and more difficult to analyze. Moreover, in addition to considering the per-packet approach, we also explore *statistical*

approaches for estimating *average fault rate* (*i.e.* the fraction of packets that experienced faults) on the data path. Finally, our approach can be seen as a secure analogue of the statistical path-quality measurement techniques designed by the Internet measurement community [7,8,15,12,21] for the adversarial setting. Our work is most closely related to that of Stealth Probing [3], a statistical fault detection protocol designed for a similar adversarial setting.

Techniques from theoretical cryptography: In contrast to previous work, *in this paper we use a rigorous theoretical framework to explore a wider space of solutions for the adversarial setting; we consider both per-packet and statistical approaches for performing fault detection (FD).* In fact, by working within the framework of theoretical cryptography to precisely define security for our adversarial setting, we have exposed security vulnerabilities of other FD protocols [16, 22] (we discuss these in the body and footnotes of the paper). Furthermore, our rigorous approach has allowed us to *prove* the security of our FD protocols, and to use the techniques of [13] to present a set of *negative results* that determine the minimum resources necessary to build any FD protocol that is robust to adversaries. Though our security definitions consider active adversaries with extensive powers (Section 2 and 3.1), many of our negative results hold even in a weaker adversarial setting (see Section 4). We begin by introducing the reader to Alice, a source edge network, (edge router or stub AS), who sends data packets to Bob, a destination edge network. Eve, the adversary, occupies some subset of intermediate nodes on the path. *In this paper, a node can be either an AS or an individual router.*

Results: This paper is a rigorous theoretical exploration of solution space for FD; rather than advocating a specific network architecture, we present new protocols and negative results that can be used to guide the placement of measurement and security functions in future networks (Section 10). We present negative results (Section 4) that prove that *any* secure per-packet or statistical FD protocol requires (1) a key infrastructure, (2) cryptographic operations at Alice/Bob, and (3) dedicated storage at Alice. Our results imply that any scheme that does not make use of these resources *cannot* be secure in the adversarial setting. Next, we present a simple secure per-packet FD protocol, *Per-Packet Ack*, and two novel secure statistical FD protocols, **Pepper Probing** and **Salt Probing**. In Per-Packet Ack, Bob acknowledges each packet that Alice sends with an authenticated ack packet (Section 5). Pepper Probing and Salt Probing are efficient protocols that allow Alice to estimate the average fault rate, up to an arbitrary level of accuracy, without requiring any modifications to Alice's traffic. In Pepper Probing (Section 7), Alice and Bob sample packets in a coordinated manner with a cryptographic hash function, similar to the approach of Trajectory Sampling [8]. Salt Probing (Section 8) extends Pepper Probing to the public-key setting via a timing strategy similar to that of TESLA [20].

2 Our setting

Goals and non-goals: When considering path quality we focus on both *availability* (path delivers packets to the correct destination) and *integrity* (packets arrive unaltered). Therefore, we say that a fault occurs when a packet sent by Alice fails to arrive unmodified at Bob within a reasonable amount of time. Our techniques can be extended to obtain finer measurements, such as calculating the average round-trip time (RTT); however, to simplify the presentation we do not discuss RTT measurements here. In this paper, we do *not* study techniques to *prevent* an adversary from adding packets to the data path or from (selectively or fully) causing faults, nor do we consider techniques that guarantee confidentiality or protect against traffic-analysis attacks. Moreover, our protocols are *not* designed to *diagnose* the cause of a fault. In the adversarial setting it is extremely difficult to accurately distinguish between benign and adversarial behaviour because an adversary can always drop packets in a way that 'looks like congestion'. As such, we do not distinguish between faults caused by benign causes (*e.g.* congestion, node failure), malicious behaviour, or even increased congestion on a link during a denial-of-service (DoS) attack. Finally, for lack of space, we do *not* explicitly address the problem of *protecting* our FD protocols themselves from DoS attacks (*e.g.* an adversary crashes an FD monitor by flooding it with messages that require verification of a digital signature); however, standard rate-limiting techniques can mitigate these attacks.

Adversary model: We consider both benign faults (due to congestion or node failure) and malicious faults (caused by an active adversary). We assume that Alice cannot trust any intermediate node along the data path, except for Bob. One (or more) nodes on the data path may be controlled by an active adversary, Eve, with extensive powers: she may add, drop, or modify traffic on the data path; she may observe the traffic on the data path and use any observations, including timing information, to learn how to bias Alice's measurements; she may attempt to blame her own malicious behaviour on innocent nodes, *e.g.* by simulating the behavior of the system during past instances of congestion-related loss.

Per-packet vs. statistical approaches: In per-packet schemes, Alice learns if a fault occurred for *each packet* that she sends to Bob. To reduce the overhead of per-packet schemes, we also consider statistical schemes, where Alice instead estimates the average fault rate on the data path. We believe that statistical schemes are sufficient for most situations; because it is natural that some packets are dropped due to congestion, we argue that the network layer neither cares nor expects to know the fate of every single packet sent. On the other hand, if the network layer wants to detect small transient problems which selectively harm specific end-hosts (*i.e.* drops of one or two TCP SYN packets), then per-packet approaches are more appropriate.

Resources and Evaluation Criteria: We shall evaluate our protocols based on how efficiently they use the following five resources: We prefer protocols that minimize (1) *communication overhead*, (2) *computation* of cryptographic operations, and (3) use of dedicated *storage* in the router. Furthermore, the difficulty of building up and maintaining a (4) *key infrastructure* for the Internet is well known. As such, we prefer schemes that require few keys; public-key schemes, where each node has only a single key, are preferred over schemes where each pair of nodes share a symmetric secret key. Finally, we prefer protocols that (5) *do not affect the router's internal packet-processing path.* In fact, all our protocols do not modify any packets sent by the source edge-network, so that they can implemented in a monitor located off the critical path in the router. This approach has the additional advantage of minimizing latency in the router, not increasing packet size, and allowing Alice to turn measurements on and off without having to coordinate with Bob.

Organization: We start by defining security for FD, present a set of negative results that shows the resources required for any secure FD protocol, and then present a simple secure per-packet FD protocol. We then present two novel statistical FD protocols, Pepper Probing and Salt Probing.

3 Definitions and Notations

A symmetric fault detection scheme consists of five algorithms, $\mathsf{Init}, R_A, R_B, \mathsf{Decode}, \mathsf{FaultFunction}$, which together tell Alice and Bob how to decide transmit data packets, how to recover the data from the protocol, and how to judge the faultiness of the link.

The function Init is used to create some secret auxiliary information, which Alice and Bob know but not the adversary. That is, $(\mathsf{aux}_A, \mathsf{aux}_B) = \mathsf{Init}(x)$ for a uniformly random x, and aux_A is given to Alice and aux_B is given to Bob. We do not discuss how Alice and Bob may agree upon the auxiliary in secret; the interested reader may refer to *e.g.* [9, 10] for ideas. Intuitively, this auxiliary information will provide Alice and Bob with some keys, *e.g.* a secret symmetric key or a public/private key pair.

The algorithms R_A, R_B are used by Alice and Bob respectively to encode and transmit their data. They may be interactive, *i.e.* R_A, R_B engage in several rounds of communication in order to transmit each data packet d, or they may be non-interactive, *i.e.* R_A consists of taking the data and generating a single packet, and R_B consists of taking the received packet and generating a single response message.

We call the communication associated with transmitting a single packet d the *exchange* associated with packet d. Let $\langle R_A, R_B\rangle(d)$ denote the transcript between Alice and Bob generated when for the exchange transmitting d.

The algorithm Decode is used to retrieve the message from the exchange transcript, *i.e.* $d = \mathsf{Decode}(\mathsf{aux}_B, W_d^B, \langle R_A, R_B\rangle(d))$ where aux_B is the auxiliary information of Bob and W_d^B is the randomness Bob used to perform exchange d with Alice. Naturally we will require our systems to retrieve the data correctly

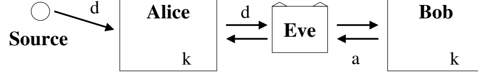

Figure 1: FD security game.

when no adversary is present, *i.e.* $d' = d$ when $\langle R_A, R_B\rangle(d)$ is transcript where no adversary was present.

The algorithm FaultFunction is used to decide when an adversary was active during the security game. The FaultFunction function takes as input the auxiliary information of Alice aux_A, the history of the game H and Alice's private randomness W, and outputs $\mathsf{FaultFunction}(\mathsf{aux}_A, H, W)$, which will have a different format depending on whether we care about per-packet security or statistical security. (The format is specified in each of the definitions below.)

The normal execution of the protocol involves using Init to set up the auxiliary information of Alice and Bob, using R_A, R_B transmit the data from Alice to Bob and Bob using Decode to retrieve the data from the transcript, and finally Alice using FaultFunction to decide how faulty the game was.

We let $\Delta(X, Y)$ denote the *statistical distance* between two distributions X and Y [9].

3.1 Security Definition

We will define the behavior we want from the algorithms we described in the previous section. First we require that Bob can always recover the data correctly from the transcript.

Definition 3.1 (Non-triviality)**.** A per-packet fault detection scheme is *non-trivial* if for any aux_B, W_d^B, $\langle R_A, R_B\rangle(d)$ (generated according to W_d^B), we have that

$$\mathsf{Decode}(\mathsf{aux}_B, W_d^B, \langle R_A, R_B\rangle(d)) = d$$

In theoretical cryptography, game-based definitions are used to obtain precise guarantees of security [9]. We now present a game-based definition for secure per-packet FD, consisting of a description of the game setting, followed by correctness and security conditions.

Definition 3.2. The *fault detection game* for a fault detection scheme (Init, R_A, R_B, Decode, FaultFunction), versus an adversary (i.e. an algorithm) Adv is defined as the following. Adv is given access to three oracles, a Source oracle and oracles computing R_A, R_B, which contain their respective parts of $(\mathsf{aux}_A, \mathsf{aux}_B) = \mathsf{Init}(x)$ for a random x. Each algorithm and each oracle keeps local clocks that may be used for computing time-outs or timing the arrival of packets. We assume that the oracles simulate a constant latency delay between each link.[1]

We use the 'Source' entity in Figure 1 to model the end-hosts that generate the data packets d that Alice sends Bob. When Adv queries Source a data packet d is generated and the exchange for d between R_A, R_B is initiated. The method of generating d will be depend on whether we wish to consider strong or weak security (see Definition 3.4 and Definition 3.3).

Adv is allowed to make queries to R_A and R_B oracles at will. During the execution of the game we keep track of two lists: $\mathcal{L}_{\mathsf{Sent}}$ which is a list of all the d that were generated by Source, while $\mathcal{L}_{\mathsf{Received}}$ is a list of all the d that were successfully recovered by Bob using the procedure Decode.

We'll use two different Source oracles for the fault detection game in Definition 3.2 to give weak and strong definitions. The weak version will require that the adversary win against any pre-determined message distribution, while the strong version will allow the adversary to choose the messages.

[1] We do not assume any relationship between the time it takes to perform computation and the time it takes to transmit packets. This is because an adversary may be willing to invest enormous resources into his computational power and so may be able to compute faster than we expect; or, contrapositively, we cannot base our security on the assumption that the adversary is much slower than the communication links.

p:	Fraction of packets acknowledged by Bob.
α:	False alarm threshold. Alice avoids raising an alarm when the fault rate is below α.
β:	Detection threshold. Alice raises an alarm when the fault rate exceed β.

Table 1: Parameters for statistical FD.

Definition 3.3. By *weak fault detection game* we mean the fault detection game of Definition 3.2 where the Source oracle generates messages d sampled independently each time according to some fixed distribution $\mathcal{D}$, subject to the condition that the collision probability of $\mathcal{D}$ is negligible (*i.e.* the probability that Source will generate identical packets is negligible). [2]

Definition 3.4. By *strong fault detection game* we mean the fault detection game of Definition 3.2 where the Source oracle generates messages given by the adversary, *i.e.* Adv specifies to Source the message d to be sent in an exchange, subject to the restriction that all messages d are unique.

We will use the weak definition in our negative results and the strong definition in our protocol constructions. Doing this gives us the strongest possible statements; the protocols we present are secure against even strong adversaries, while our negative results apply to a broad class of protocols (even those where the adversary has no control of the data sent by end hosts).

3.1.1 Per-Packet Definition

We begin with the per-packet definition, since it is similar to the definition of classical cryptographic primitives such as message authentication codes and authenticated login systems.

In per-packet security, the output of $\mathsf{FaultFunction}(\cdot)$ is a vector of T entries, one for each exchange, where each entry takes value in $\{\mathsf{Fault}, \mathsf{NoFault}\}$.

Definition 3.5. We say a fault detection scheme is weak (resp. strong) per-packet secure if no efficient adversary Eve playing the weak (resp. strong) fault detection game can, with non-negligible probability, cause there to exist a faulty exchange that Alice does not detect. That is, the protocol satisfies the following two conditions. The **correctness condition** says that, for every exchange d in which Eve does not drop or tamper with any messages, we have that $\mathsf{FaultFunction}(\mathsf{aux}_A, H, W)_d = \mathsf{NoFault}$. The **security condition** says that no efficient Eve can cause there to exist an exchange such that data does not arrive unaltered at Bob, and yet Alice does not detect that the data was tampered with or dropped. Formally, for all efficient Eve playing the weak (resp. strong) fault detection game, we have that,

$$\Pr[\exists d \in \mathcal{L}_{\mathsf{Sent}} \setminus \mathcal{L}_{\mathsf{Received}} \ s.t.\ \mathsf{FaultFunction}(\mathsf{aux}_A, H, W)_d = \mathsf{NoFault}] \leq \varepsilon$$

for $\varepsilon = \varepsilon(n)$ a negligible function of the security parameter.

3.1.2 Statistical Definition

We argue that for many applications, Alice is more interested in determining if the average fault rate (*i.e.* the fractions of packets sent by Alice for which faults occurred) than determining the fate of each individual packet. As such, we can use statistical methods, similar to those used by the Internet

[2] We make this assumption to prevent Eve from trivially winning the FD game using a *replay attack*; suppose Source sends the same data packet d twice. The first time, Eve forwards d to Bob and gets valid ack a. The second time she *drops* d but responds to Alice with the valid ack a. In practice, we can prevent replay attacks by timestamping Bob's acks with an expiry time, such that no repeated packets are sent, (*e.g.* due to natural entropy in packet contents, TCP sequence numbers, and IP ID fields) for the duration of the time interval for which Bob's acks are valid.

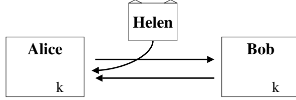

Figure 2: Helen.

measurement community, to reduce the overhead required in per-packet FD protocols. In statistical security Alice again will output Fault or NoFault for a group of many exchanges, which we call an *interval* of exchanges, rather than for each single exchange individually.

In statistical security, the FaultFunction$(\cdot)$ function outputs a vector whose length is the number of intervals occuring in the game. For each interval j, FaultFunction$(\cdot)_j \in \{\text{Fault}, \text{NoFault}\}$.

Definition 3.6 ((α, β, δ)-Statistical FD Security)**.** A fault detection scheme is weak (resp. strong) $(\alpha, \beta, \delta, T_0)$-statistically secure (where $\alpha < \beta$ and T_0 is a function of α, β, δ), if for any adversary playing the weak (resp. strong) fault detection game, both the *correctness* and *security* conditions (defined below) hold. In the statistical setting, we divide the exchanges into intervals of T_0 consecutive exchanges. Let $\kappa(u)$ be the fault rate for the u'th interval. At the end of the interval, Alice either outputs Fault(if she decides the fault rate exceeds β) or NoFault(if she decides that the fault rate did not exceed α.)
Correctness condition: Intuitively, an FD protocol is correct if Alice outputs NoFault at the end of an interval where less than an $\kappa(u)$-fraction of data sent by Alice did not arrive at Bob for $\kappa(u) < \alpha$, and no other messages, (*e.g.* data packets, acks) were tampered with or dropped for the remaining $1 - \kappa(u)$ exchanges.

More formally, consider an interval u where Bob fails to recover the data sent during each faulty exchange. In other words, consider only adversaries that cause fault by preventing data packets from reaching Bob; we exclude adversaries that cause faults by letting data through to Bob, but tamper with other messages sent during an exchange, *e.g.* dropping the acks sent back to Alice. We say that an FD protocol is correct if, for any such interval u and conditioned on all previous intervals, the probability that the fault rate is $\kappa(u) \leq \alpha$ and Alice outputs Fault is bounded by $\delta + \varepsilon(n)$.

Security condition: The security condition is satisfied if, for any efficient adversary Eve playing the weak (resp. strong) fault detection game, for any interval j, conditioned on all previous intervals, the probability that the fault rate $\kappa_j > \beta$ and Alice outputs NoFault is bounded by $\delta + \varepsilon(n)$.

3.2 Properties of Security Definition

We now discuss some properties of our security definition.

Duration of an interval: Notice that our definition explicitly puts a lower bound on T, number of exchanges for that the game must be played for before Alice decides whether or not a fault occured. This bound on T is *essential* because a statistical scheme only measures behaviour on average; Alice can only have high confidence in her measurements when her sample size is sufficiently large.

Monotonicity: Our security definition is *monotone* in that it protects not only against adversaries like Eve in Figure 1, but also against adversaries like Helen in Figure Figure 2 (a node *off the data path monitored by Alice* who attempts to trick Alice into detecting faults on the path to Bob so that Alice will switch her traffic to a path through Helen). We assume that Helen can only *add* packets to the data path between Alice and Bob. Observe that Helen could potentially trick Alice into detecting a fault (when no fault occurred) by sending Alice invalid ack packets spoofed to look like they came from Bob. However, our definition protects against attacks by Helen because the definition is *monotone*: as long as Alice receives a valid ack for every packet she sends, she will never declare a fault, *even if she receives additional nonsense packets.*

Congestion: Our definition does not explicitly model congestion. FD is an end-to-end measurement; as such, it is sufficient to detect the total fault rate over an entire path, without distinguishing between faults caused by normal congestion or adversarial behaviour.

Faulty forward or reverse path? Collecting one-way measurements of a path is notoriously difficult, since a sender cannot easily differentiate between faults on the forward path (from Alice to Bob) and faults on the reverse path (from Bob to Alice). As such, we define our FD protocols so that Alice always obtains a conservative estimate, *i.e.* a lower bound, on the fault rate on her forward path. This definition works best in the setting of *symmetric paths*, where the forward and reverse paths are identical, since here Eve has no incentive to drop the acks on the reverse path. (If she does, she simply makes the path look worse). In contrast, in the setting of *asymmetric paths*, it is useful to distinguish between faults that occur on the forward path and faults that occur on the reverse path. In this setting, Eve occupying only the reverse path may have an incentive to drop acks, perhaps to confuse Alice into thinking that the forward path is faulty.

We argue that any FD protocol that distinguishes between faults on the forward vs the reverse path must also include a coordinated path-switching mechanism between Alice and Bob, even in the benign setting. To see why, observe that Alice cannot distinguish between (a) the forward path failing and (b) the reverse path failing. In both situations, Alice cannot communicate with Bob. However, to learn why communication failed, in case (a) the forward path must be switched, while in case (b) the reverse path must be switched. Because path-switching mechanisms are outside our scope, in this paper we do *not* construct FD protocols that explicitly distinguish between faults on the forward vs the reverse path. While we leave this interesting problem to future work, we note that such FD protocols are still subject to our negative results, and they could be designed using our simpler (Per-Packet Ack, Pepper and Salt Probing) FD protocols as building blocks.

4 Negative Results

Here we show that *any* FD scheme that is secure according to the per-packet definition in Sections 3.1 (1) requires shared keys between Alice and Bob, (2) requires Alice/Bob to perform some cryptographic operations,[3] and (3) requires Alice to modify her memory for each packet that she monitors. Furthermore, while our results about the necessity of keys and cryptography rely on the assumption that Eve actively forges acks, our result about the necessity of storage holds even when Eve's adversarial powers are limited to dropping packets. Our negative results imply that the resource overhead of our Per-Packet-Ack, Pepper and Salt Probing protocols (Sections 5, 7 and 8) are unavoidable in the sense that we could not have designed them without keys, cryptography and storage.

4.1 Keys are Necessary for Fault Detection

Here we show that Alice and Bob need some secret *correlated* information or else schemes cannot be secure.

Proposition 4.1. *For any fault detection scheme, if* aux_A *and* aux_B *are independent, or if it is easy to guess* aux_B*, then the scheme is not secure (in any sense!).*

Proof. We prove this in the contrapositive. Assume that $\mathsf{aux}_A, \mathsf{aux}_B$ are independent so that Alice and Bob have no shared secret key. It follows that Eve knows everything that Bob knows, so that Eve can simply sample aux_B for herself and drop all the data that Alice sends to Bob, but respond in the way that Bob would respond. Therefore, since the sampled aux_B is distributed identically to the aux_B of honest Bob, Eve convinces Alice that nothing is wrong, and the FD scheme cannot be secure.

Furthermore, if it is easy to guess aux_B, then clearly Eve can win again whenever she guesses aux_B correctly by simulating Bob's responses to Alice and while dropping all the packets that Alice sends to Bob. ∎

[3] Listen [22] is an FD protocol that does not require Alice or Bob to perform cryptographic operations. Our results imply that Listen is insecure in our adversarial setting.

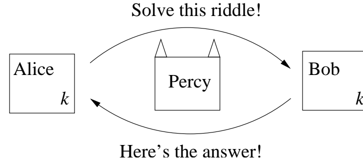

Figure 3: KIS security game.

Thus, for the remainder of the paper we will assume that there is indeed shared secret information between Alice and Bob. In particular, we will assume that they either have shared secret keys or a public/private key pair.

4.2 Cryptography is Necessary for Fault Detection

We now prove that the keys shared by Alice and Bob must be used in a 'cryptographically-strong' manner. We now show that any secure per-packet FD is *at least as complex* as a a secure keyed identification scheme (KIS). KIS is well-studied cryptographic primitive that is know to be at least as complex as symmetric-key encryption. (In Appendix **??** provide an equivalent argument by reducing secure FD directly to one-way functions.) In a **secure keyed identification scheme (KIS)** [9] Alice and Bob share a secret key, and Alice ascertains Bob's identity by asking him to respond to a challenge (*e.g.* solve a riddle) that can only be responded to by someone who knows the secret key. (To illustrate this, Figure 3 we show a sample two-message KIS protocol. In practice a KIS can have a more arbitrary structure, *e.g.* a four-message protocol, etc.) A KIS is secure if an impersonator who does not know the key can't respond to the challenge with better success than by just guessing a random answer. Here, we are concerned with KIS schemes that are secure against a passive adversary, Percy (the impersonator), who eavesdrops on the interactions between Alice and Bob and then tries to impersonate Bob by responding to Alice's challenge on his own

Our proof implies that secure identification is an unavoidable part of secure FD. That is, any secure FD protocol must ensure that Alice can distinguish between acks sent by Bob, and acks sent by an adversary impersonating Bob. Furthermore because KIS is known to be at least as complex as symmetric-key encryption [13], our proof implies that *one cannot expect to design secure FD protocols that run blazingly faster than symmetric-key encryption protocols.*[4]

We start with the per-packet setting.

Theorem 4.2. *Any weakly-secure per-packet fault detection scheme is at least as computationally complex as a KIS that is secure against a passive adversary.*

Proof. To prove that any secure FD scheme is at least as complex as a secure KIS, we show that an arbitrary secure FD scheme can be used to construct a new KIS. The construction is simple: Alice's key in our new KIS is aux_A (from the FD scheme). Bob's key in our new KIS is aux_B (from the FD scheme). The challenge in our new KIS is Alice's algorithm R_A (run on one exchange d of the FD scheme). The correct response to the challenge in our new KIS is Bob's algorithm R_B (run on one exchange of the FD scheme). Alice accepts Bob's identification in this KIS iff Alice outputs $\mathsf{NoFault}$ at the end of the exchange (of the FD scheme).

To complete the proof, we must show that if the FD scheme used in the above construction is secure, then our new KIS construction is also secure. We do this in contrapostive, by showing that if there existed an efficient adversary Percy that breaks the security of this KIS construction, then Percy can be

[4]Our reduction from per-packet secure FD to secure KIS is essentially "tight" in the sense that one exchange of the FD protocol becomes one authentication session of the KIS protocol. Showing that secure KIS implies symmetric-key encryption incurs a much larger loss of efficiency: the most straight-forward proof requires going through one-way functions. It is an open problem whether one could construct a secure symmetric-key encryption from secure KIS in a more efficient manner.

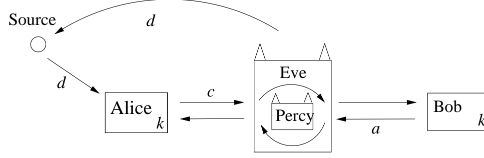

Figure 4: Reduction from FD to KIS.

used to construct an adversary Eve that breaks the security of the FD protocol. As in Figure 4, Eve uses Percy to break the security of the FD protocol as follows: At first, when Percy wants to eavesdrop on an interaction between Alice and Bob, Eve asks the Source to generate a random packet d. Eve then lets Percy see the exchange between Alice and Bob related to packet d. Note that Percy gets to see messages that are exactly what he expects to see in the KIS security game. Next, when Percy is ready to impersonate Bob, Eve again asks the Source for a packet d, but now instead of forwarding Alice's messages pertaining to d to Bob she *drops* all communication (for this exchange) between Alice and Bob, while responding to Alice with Percy's message. The proof follows from the fact that Eve successfully tricks Alice into thinking the Bob received Alice's messages (and therefore breaks the security of the FD protocol) whenever Percy successfully responds the the challenge in the KIS (and therefore breaks the security of KIS by impersonating Bob). ■

We now extend this proof to the statistical setting.

Theorem 4.3. *Any weakly-secure (α, β, δ) fault detection scheme is at least as computationally complex as a KIS that is secure against a passive adversary.*

Proof. This (contrapostive) proof is entirely identical to the proof of the per-packet case except that now, we define one interaction of the KIS as a entire *interval* (*i.e.* T_0 exchanges) of the FD protocol (instead of defining one interaction of the KIS as a single *exchange* of the FD protocol.) That is, in the per-packet case the KIS was a two-message protocol; in the statistical case, the KIS is a $2T_0$-message protocol.

Whereas before we showed that Eve can break security by dropping a packet and using Percy to impersonate Bob acks during a single exchange, Eve now breaks security by dropping packets during an entire *interval* and using Percy to impersonate Bob during the entire interval. During the impersonated interval, since Percy impersonates Bob's behaviour an interval where the the fault rate does not exceed α, Alice will not raise an alarm (during the impersonated interval) even though the true false rate exceeded β. As such, Eve again breaks the security of the FD protocol, completing the contrapositive proof. ■

4.3 Storage is Necessary for Fault Detection

In this section, we prove that in secure FD Alice must modify storage for (almost) each packet that she samples. We start with a proof for the per-packet FD, and then extend the proof to the statistical setting.

Theorem 4.4. *In a per-packet weakly secure FD protocol, then Alice must modify storage at least once during each exchange.*

Proof. We prove this in the contrapositive. Assume that Alice had stored no information (*i.e.* does not modify her memory) for some exchange related to a packet d. It follows that Eve can simply block all communication from Alice to Bob, and Alice won't notice (because Alice immediately forgets about the packet d after she sends it), and the FD scheme cannot be secure. ■

We now extend the proof to the statistical setting, for *FD protocols based on sampling.* In an FD protocol based on sampling, Alice decides if each packet it sends requires acknowledgement from Bob or not. We say that a packet that Alice requires Bob to acknowledge is *sampled.*

Theorem 4.5. *In any (α, β, δ)-weakly secure FD protocol based on sampling, Alice must modify storage for at least $\Omega((\frac{\beta}{\beta-\alpha})^2 \ln \frac{1}{1/2+\delta})$ exchanges.*[5]

Proof. We prove this in the contrapositive. Suppose that Alice modifies storage for for less than $\Omega((\frac{\beta}{\beta-\alpha})^2 \ln \frac{1}{\frac{1}{2}+\delta})$ exchanges. Each exchange where Alice does not modify storage cannot be sampled since Alice does not even remember seeing that interval. Let $T < \Omega((\frac{\beta}{\beta-\alpha})^2 \ln \frac{1}{\frac{1}{2}+\delta})$ be the actual number of exchanges that Alice samples.

Now consider the adversary that drops packets at random with probability β. With probability 1/2 it will drop at least a β fraction of the exchanges.

Now Alice samples only T exchanges, and let $X_1, \ldots, X_T$ be whether or not the i'th exchange she samples is dropped by Eve. Notice that $\mathbb{E}[X_i] = \beta$. Now Alice's estimate of the fault rate will be at most $\frac{1}{T}\sum_{i=1}^{T} X_i$. Letting $\mu = \frac{\alpha+\beta}{2}$, this is incorrectly undetected with probability

$$\Pr[\tfrac{1}{T} X_i < \mu] > \binom{T}{\mu T} \beta^{\mu T}(1-\beta)^{\mu T}$$

which, using Stirling's approximation (or see the alternative derivation in [?]) is at least

$$\geq \tfrac{1}{T+1} 2^{-D(\mu \| \beta)T}$$

Here $D(\cdot\|\cdot)$ is the relative entropy function. Looking at the Taylor expansion of $D(\cdot\|\cdot)$ with respect to μ, β, we see that for our range of parameters $D(\frac{\alpha+\beta}{2}\|\beta) \leq \frac{16(\beta-\alpha)^2}{\beta}$, and so the above is at least

$$\geq \tfrac{1}{T+1} 2^{-16\frac{(\beta-\alpha)^2}{\beta}T}$$

For our choice of T, this gives us that the probability is at least

$$> 1/2 + \delta$$

So Eve's overall probability of breaking security is:

$$\Pr[\kappa > \beta \text{ and } \tfrac{1}{T} X_i < \tfrac{\alpha+\beta}{2}] > 1 - (1/2 + 1/2 - \delta) > \delta$$

■

5 Per-Packet-Ack: A Simple FD Protocol

Per-Packet Ack is a simple secure per-packet FD protocol, where Bob securely acknowledges receipt of every packet he sees. Despite the simplicity of the protocol, shown in Figure 5, we explain it here in some detail in order to introduce the reader to some of the cryptography we use in later parts of the paper.

[5]Note that this bound is essentially independent of δ. One can obtain better dependence on δ at the cost of worse dependence on $\beta-\alpha$, namely something like $\Omega((\frac{\beta}{3\beta-\alpha})^2 \ln \frac{1}{\delta})$. This is done by a simple modification of our proof to require Eve to drop at a rate of 3β instead of β and then showing that the probability that the fault rate of the game is below β is exponentially small. However, since the dependence on δ is logarithmic anyway we prefer to focus on the stated version of the theorem.

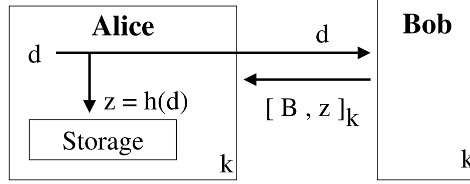

Figure 5: Per-Packet Ack protocol.

The protocol is built from two cryptographic primitives: A **collision-resistant hash function (CRH)** [9], which we denote by h, is a hash function for which it is computationally infeasible to find two inputs $x \neq x'$ that map to the same output $h(x) = h(x')$. In practice h can be implemented with a function such as SHA-2 [17].

Definition 5.1 (CRH)**.** A function family $\mathcal{H} = \{h\}_k$ mapping $h_k : \{0,1\}^* \rightarrow \{0,1\}^{n'}$ for $n' < n$ is a family of collision-resistant hash functions if for all efficient algorithms A, the probability over a random $h \overset{R}{\leftarrow} \mathcal{H}$ that A can find m, m' such that $h(m) = h(m')$ is negligible, or formally

$$\Pr_{h \overset{R}{\leftarrow} \mathcal{H}} [A(h, 1^n) = (m, m') \ s.t. \ h(m) = h(m')] \leq \varepsilon_{\mathsf{crh}}$$

where $\varepsilon_{\mathsf{crh}}$ is the *CRH-advantage* and is negligible in n.

A secure deterministic **message authentication code** (MAC) [9] protects a message's integrity and authenticity by allowing a receiver, who shares a secret key k with the sender, to detect any changes made to the sender's message. A MAC is secure if no efficient adversary can forge a valid signature on any arbitrary message (with non-negligible probability). We use the notation $[m]_k$ to denote message m MAC'd with key k.

Definition 5.2. Formally, a message authentication code consists of two algorithms $(\mathsf{Sign}, \mathsf{Ver})$ where Sign is used to MAC the message, and Ver is used to verify the validity of the MAC on a message. A MAC is secure if for all efficient algorithms Fred, where Fred gets access to oracles for Sign and Ver[6], and Fred outputs a message and signature (m, σ), subject to the condition that m was not queried to Fred's Sign oracle, then

$$\Pr_k[\mathsf{Fred}^{\mathsf{Sign}_k(\cdot), \mathsf{Ver}_k(\cdot)}(1^n) = (m, \sigma) \ s.t. \ \mathsf{Ver}_k(m, \sigma) = 1] \leq \epsilon_{\mathsf{mac}}(n)$$

for a negligible function $\epsilon_{\mathsf{mac}}(n)$. We call $\epsilon_{\mathsf{mac}}(n)$ the *MAC-advantage*.

The Per-Packet-Ack protocol is a simple protocol where each packet is acknowledged with a MAC'd ack. This protocol satisfies strong per-packet security.

Definition 5.3. Init generates a truly random secret key k to be shared by Alice and Bob. We assume there is some publicly known CRH h. Alice sends each data packet d completely in the clear to Bob, and stores a packet digest[7] $z = h(d)$ along with a time-out (we say that a fault occurs if the time-out expires before Alice receives an ack for that packet). Then Bob receives the packet d, he computes the packet digest $z = h(d)$ and sends an ack of the form $a = [B, z]_k$ where B is a public unique identifier for Bob's identity. Alice removes a packet digest z from storage when she receives an acknowledgment a containing a matching packet digest z and a valid MAC. Periodically, Alice checks her storage to see if any packets awaiting acknowledgment have timed out; if so she raises an alarm and declares that a fault occurred.

Theorem 5.4. *Per-Packet-Ack is strongly per-packet secure.*

[6]But Fred does not know the key used by the Sign and Ver oracles

[7]Packet digests must be computed only on *immutable* fields of the packet, that are unchanged as the packet traverses the data path. See [8] for immutable fields in an IP packet.

Proof. Notice that in this protocol, acks are *unambiguous*: they specifically depend on the contents of the packet d that Bob receives. It follows that if Eve wants to create a valid ack to a packet d that was dropped before it reached Bob, she either has (i) to find a collision d' in the CRH h, so that $h(d) = h(d')$, and ask Bob to generate an ack for d', or (ii) she has to forge the MAC signature on an ack $(B, h(d))$ without Bob's help. By the definition of CRH, the probability (i) that the adversary finds d, d' such that $h(d) = h(d')$ is at most $\varepsilon_{\mathsf{crh}}$.

Conditioned on event (i) not happening, we can also show that (ii) occurs with probability at most $\varepsilon_{\mathsf{mac}}$ by reducing the Per-Packet-Ack Protocol to MAC. To do this, observe that if we had an Eve breaking the security of the Per-Packet-ACK protocol, we could construct an adversary, Fred, that forges signatures. (Fred can simulate the FD game to Eve as follows: He simulates the Source and Alice by generating random, unique messages d and passing them on to Eve, and then simulates Bob by computing the packet digest $z = h(d)$, getting his Sign oracle to generate the ack $a = [B, z]_k$ by querying the oracle on (B, z), and then passing a along to Eve. Then Fred can use Eve to win the MAC game by outputting the ack a the Eve generated during the exchange where Eve forged the MAC on some packet Eve dropped. During the exchange where Eve drops a packet, Fred need not query his Sign oracle on the packet, so that if Eve wins the FD game, then Fred will have a MAC on a unique message that was never queried to the Sign oracle.)

Thus the overall advantage of any efficient adversary is at most $\varepsilon_{\mathsf{crh}} + \varepsilon_{\mathsf{mac}}$ which is negligible, so Per-Packet-Ack is secure. ∎

Remark 5.5 (Ambiguousness)**.** A protocol is unambiguous if distinct data packets have distinct acknowledgements. Consider for example a scheme is one in which Bob sends "Got a probe" each time he receives a probe. Intuitively, this scheme should be insecure, because an adversary can drop all of Alice's messages to Bob and convince Alice to accept the route by sending Alice "Got a probe" for every packet he drops. This a similar type of attack is possible if the protocol simply sends back a protocol in which Bob sends back a count of the number of packets or bits received (*e.g.* as suggested in one version of Fatih [16]); namely, the adversary can drop all of Alice's packets, and simply send Bob a bunch of nonsense packets such the the count of the number of packets that Bob receives is identical to the number of packets or bits sent by Alice.

One way to improve the communication overhead of Per-Packet Ack protocol is to "batch together" many acknowledgments into a single super-ack packet. We note however that the batching operation should preserve the unambiguousness property of the Per-Packet-ACK protocol. The obvious way to do this is to send many acks $a_1, a_2, .., a_{50}$ in a single 'super-ack' packet.

Moreover, because packets can arrive at Bob out of order, any batching operation must either reconstruct the exact same order of packets as sent Alice or must unaffected the order of the packets. The former option is very heavy-weight since it means that some ordering information must be added to the data, while the latter option means essentially that Bob has to include a separate packet digest for each individual packet he receives in in the super-ack anyway, so that he does not save much in the way computation. Furthermore, batching acks together may increase the storage requirements at Bob and Alice, since now Alice has to store her data for a longer of period of time (RTT plus time it takes to create a batch).

6 Why use Statistical Fault Detection based on Sampling?

The per-packet FD protocol of Section 5 required acks and memory modifications every packet sent. To reduce the high overhead required for per-packet protocols, we now study techniques that require only an accurate measurement of the average fault rate. We will focus on schemes that use sampling.

Sampling: Instead of performing FD on every single packet, as in the Per-Packet Ack protocol that we present in Section 5, in sampling schemes we randomly select a p fraction of the packets to monitor in order to obtain an estimate of average behavior. Our estimate becomes accurate if we obtain a sufficient number of samples. We let *probing schemes* denote statistical FD schemes that use sampling,

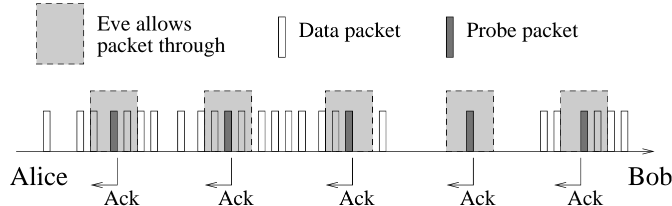

Figure 6: Attack on active measurement.

and *probes* denote packets that require acks, and $p < 1$ denote the fraction of packets sent by Alice that are acknowledged by Bob. Security in the statistical setting guarantees that *Eve cannot bias Alice's estimate of the average fault rate.*

Probe indistinguishability and the trouble with active measurement: Any probing scheme that is secure in the adversarial setting requires a property called probe indistinguishability [3, 1]. That is, to prevent Eve from biasing Alice's measurements by treating probe packets preferentially (while, say, dropping all other packets), *Eve must not be able to distinguish probe packets from non-probe packets.* In active probing, Alice injects specially crafted probe packets into her data traffic stream according to some arrival process, and performs FD on probe packets only (*e.g.* ping, traceroute, and the approaches of [15, 21, 12]). Active probing is undesirable for a number of reasons; for example, injecting additional special probe packets into the system interfers with measurement [5] and increases traffic on the network. Furthermore, to prevent trivial attacks on probe indistinguishability, active probes must also be designed to 'blend in' with data traffic (*e.g.* by padding and randomizing probe packet contents and length, or by encrypting the entire traffic + probe scheme, as in Stealth Probing [3].) Furthermore, we now show how use timing information can be used to break probe indistinguishability, even when the entire traffic + probe scheme is encrypted.

To illustrate these timing attacks, consider an extreme example where probes are injected according to a periodic arrival process and Bob sends an acknowledgement for each probe packet as in Figure 6. Eve can easily learn how to distinguish probes from existing traffic by observing when Bob sends ack packets. Eve can then drop all packets sent outside a small time window around the beginning of the probe period. Notice that Eve can easily discover the period of the probe arrival process begins by observing when Bob sends his acknowledgement packets, thus learning how to distinguish probes from non-probes. Then, Eve can attack by dropping all the packets that are sent outside a small time window around the beginning of the probing period. [8] Furthermore, even when active probes arrive according to a randomized arrival process (*e.g.* a Poisson process), the statistics of the probe arrival process are likely to be different from those of the data traffic arrival process, and this information can be used by a clever adversary to distinguish probe from data traffic. As such, in the remainder of this paper we avoid schemes based on active measurement, and instead focus on *passive sampling* protocols that sample from existing traffic.

7 Pepper Probing: Symmetric-Key Statistical FD

Pepper Probing extends the Per-Packet Ack protocol of Section 5 to the statistical setting by allowing Alice and Bob to sample packets in a coordinated way by computing a 'cryptographic hash' over packet contents (similar to Trajectory Sampling [8]'s approach).

Psuedo-Random Functions (PRFs): The protocol relies on the properties of pseudorandom functions. A PRF [9] is a keyed function that cannot be distinguished by any computationally-efficient algorithm from a *truly random function.* (A random function $F : \{0,1\}^n \to \{0,1\}^n$ can be thought of as the following process: for each one of its possible 2^n inputs x, choose a random n-bit string to be $F(x)$.

[8]This attack can be avoided if Bob sends out acknowledgements at fixed time intervals that are independent of the probe arrival process. However, by doing this Alice loses the ability to measure packet delay through the network.

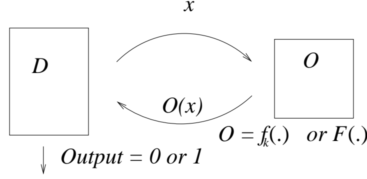

Figure 7: PRF distinguishing game.

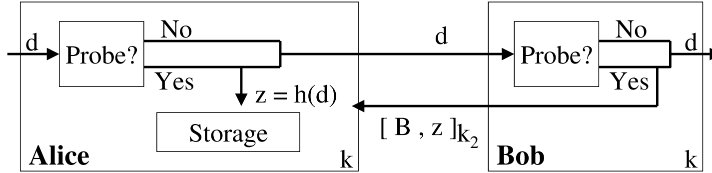

Figure 8: Pepper Probing.

This means that we need $2^n \cdot n$ coins to choose a random function.) A pseudorandom function collection is a function that can be described with only n bits but is indistinguishable from a random function. We say that the collection is efficiently computable if the mapping $(k, x) \mapsto f_k(x)$ is computable in polynomial time.

Definition 7.1. A pseudorandom function collection $\mathcal{F} = \{f_k\}$ is a collection of efficiently computable functions that are computationally indistinguishable from a random function F. Formally, we say that $\mathcal{F}$ is a PRF family if for all efficient oracle algorithms D with oracle access to $\mathcal{O}$ containing either $f_k(\cdot)$ for a random $k \in \{0,1\}^n$ (as shown in Figure 7) or $F(\cdot)$ for a truly random function F[9], it holds that

$$\left| \Pr_k[D^{f_k(\cdot)} = 1] - \Pr_F[D^{F(\cdot)} = 1] \right| \leq \epsilon_{\mathsf{prf}}(n)$$

for negligible function $\epsilon_{\mathsf{prf}}(n)$. We call ϵ_{prf} the *PRF-advantage*.

In practice, the high-throughput PRF we require for Pepper Probing can be efficiently and securely implemented in hardware with pipelined AES in CBC-mode or with a cryptographic hash function [17]. However, the oft-suggested CRC function with a secret modulus $f_k(x) = x \mod k$ should not be used because it is not a PRF! To see why, observe that the CRC is a linear function that can easily be distinguished from a random function. Consider a distinguisher D interacting with it's oracle $\mathcal{O}$ as in Figure 7. D first asks its oracle for $\mathcal{O}(x)$, and then asks for $\mathcal{O}(x+1)$. To distinguish between the CRC and the random function, D checks if $\mathcal{O}(x+1) = \mathcal{O}(x)+1$. If it is, D decides that $\mathcal{O} = f_k$, and otherwise D decides that $\mathcal{O} = F$. ■

7.1 Description of Pepper Probing

We are now ready to describe the Pepper Probing protocol, shown in Figure 8.

Definition 7.2 (Pepper Probing)**.** As shown in Figure 8, Alice and Bob share a secret $k = (k_1, k_2)$. For each packet sent, they use k_1 to compute a function Probe that determines whether or not a packet d is a probe and should therefore be digested $z = h(d)$, stored, and acknowledged. To acknowledge a probe, Bob sends Alice an ack that is MAC'd using k_2 to key a secure MAC, as in Per-Packet Ack (Section 5). The Probe function is implemented using a PRF. which we can think of as a keyed hash function (keyed here with k_1) that takes in an input (here, the data packet d), and outputs a number

[9] D does not know if $\mathcal{O}$ contains $f_k(\cdot)$ or $F(\cdot)$.

$f_{k_1}(d) \in \{1, \ldots, 2^n\}$. We let

$$\begin{array}{ll} \mathsf{Probe}_k(x) = \text{Yes} & \text{if } \frac{f_{k_1}(d)}{2^n} < p \\ \mathsf{Probe}_k(x) = \text{No} & \text{otherwise} \end{array} \tag{7.1}$$

For each interval u, Alice stores each probe packet (*i.e.* each packet d such that $\mathsf{Probe}_{k_1}(d) = \text{Yes}$) in a table $\mathcal{L}_{\mathsf{ack}}(u)$. Upon receiving ack $a = [z]_{k_{\mathsf{mac}}}$: Alice verifies that a is correctly MAC'd, and if so, removes z from her table $\mathcal{L}_{\mathsf{ack}}(u)$. To obtain an (α, β, δ)-secure protocol, the $\mathsf{FaultFunction}(\cdot)$ function outputs Fault for the u'th interval when $\frac{|\mathcal{L}_{\mathsf{ack}}(u)|}{pT} > \frac{\alpha+\beta}{2}$, while it outputs $\mathsf{NoFault}$ when $\frac{|\mathcal{L}_{\mathsf{ack}}(u)|}{pT} < \frac{\alpha+\beta}{2}$.

7.2 Security of Pepper Probing

Theorem 7.3. *Pepper Probing is a strongly $(\alpha, \beta, \delta, T)$-statistically secure fault detection scheme for any $0 < \alpha < \beta < 1$ and $0 < \delta < 1$ where probing intervals last for $T_0 = O(\frac{\beta}{p(\beta-\alpha)^2} \log \frac{1}{\delta})$ exchanges.*

Proof. For the purpose of this proof, we refer to game G, as the strongly-secure FD game using a statistical winning condition, played for the the Pepper Probing protocol.

Next, we divide the probability that Eve manages to bias Alice's measurement into two parts: either (i) Eve produces an ack to a dropped packet by finding a collision in the CRH forging a MAC (ii) otherwise (*e.g.* she violated probe indistinguishability and then made sure to drop only non-probe packets). Let E denote the event that (i) is true. Thus, we want to show that for any efficient Eve, the probability Eve winning the strong fault detection game in the (α, β, δ)- statistical sense, is

$$\begin{aligned} \Pr[\mathsf{Eve} \text{ wins game}] = & \Pr[\mathsf{Eve} \text{ wins game } G \text{ and } \mathsf{Eve} \text{ forges an ack to a dropped packet}] \\ & + \Pr[\mathsf{Eve} \text{ wins game } G \text{ and } \mathsf{Eve} \text{ fails to forges an ack for any dropped packet}] \\ & = \Pr[\mathsf{Eve} \text{ wins game } G \cap E] + \Pr[\mathsf{Eve} \text{ wins game } G \cap \overline{E}] \end{aligned} \tag{7.2}$$

which is bounded by a negligible function of the security parameter n.

We can show that

$$\Pr[\mathsf{Eve} \text{ wins game } G \cap E] < \epsilon_{\mathsf{crh}} + \epsilon_{\mathsf{mac}} \tag{7.3}$$

using an argument identical to the one we used in the proof of security for Per-Packet-Ack.

We continue the proof by showing that if the Probe function uses a PRF, then Pepper Probing is strongly $(\alpha, \beta, \delta, T_0)$-statistically secure. We do this using a contrapositive argument. We start by assuming that Pepper Probing is insecure, so that for some α, β, δ and $T_0 = O(\frac{\beta}{p(\beta-\alpha)^2} \log \frac{1}{\delta})$

$$\Pr[\mathsf{Eve} \text{ wins game } G \cap \overline{E}] \geq \delta + \tilde{\varepsilon} \tag{7.4}$$

where $\tilde{\varepsilon}$ is a *non-negligible* function of n. We shall show that that this implies that the PRF is insecure (*i.e.* that we can construct a distinguisher D that can distinguish between an PRF and a Random Function, see Definition 7.1).

Recall that we denote by game G the strongly secure FD game using Pepper Probing protocol, which makes use of a PRF in the Probe function. Now, consider game G', which is identical to game G, except that now, the strongly secure FD game is played for a FD protocol that is just like Pepper Probing *except that the PRF in the Probe function is replaced with a random function.*

Now in game G', because Alice and Bob use a truly random function decide which packets become probes, it follows from the definition of the Probe function (Equation 7.1) any packet is probe with truly random probability p. Furthermore, since was assume that all packets sent by the Source oracle are unique (thus preventing Eve from violating probe indistinguishability by observing which packets where non-probes in the past, and then dropping those packets when reappear), it follows that Eve cannot predict which packet is a probe with probability better than p. In other words, in Game G' probes are truly indistinguishable from non-probes. Finally, note that the uniqueness of packets and the properties of random functions imply that in game G' each packet is *independently* sampled (*i.e.* is designated as

a probe) with probability p. Letting $\kappa(u)$ denote the true fault rate of interval u and $H_{1,\dots,u-1}$ denote the history of intervals 1 through $u-1$, it follows that, for *every interval* u,

$$\Pr[\mathsf{Eve}\text{ wins game } G' \cap \overline{E}] \leq \Pr[\tfrac{|\mathcal{L}_{\mathsf{ack}}(u)|}{pT} \leq \tfrac{\alpha+\beta}{2} \mid (\kappa(u) > \beta), H_{1,\dots,u-1}, \overline{E}]$$

where we used the fact that $\Pr[\mathsf{Eve}\text{ fails to forges an ack to any dropped packet}] \leq 1$. Now, because a truly random function gives us truly independent samples, and since Eve did not forge any acks, it follows that $|\mathcal{L}_{\mathsf{ack}}(u)|$ is distributed as a binomial random variable $X \sim B(\kappa(u)T_0, p)$ (*i.e.* a binomial random variable sampling with probability p out of T elements), so that

$$\leq \Pr[\tfrac{X}{pT} \leq \tfrac{\alpha+\beta}{2} \mid \kappa(u) \leq \alpha]$$

Using the fact that $\mathbb{E}[X] = \kappa(u)T_0$ and $\kappa(u) > \beta$, we can derive that the above is bounded by

$$\leq \Pr[\tfrac{X}{T} - \beta p < (\alpha-\beta)p/2]$$

Using a Chernoff bound [9] (which tells us the estimated fault rate should be close to the true fault rate if the number of exchanges sampled $T_0 = O(\frac{\beta}{(\beta-\alpha)^2}\ln\frac{1}{\delta})$ is sufficiently large), we conclude that

$$\Pr[\mathsf{Eve}\text{ wins game } G' \cap \overline{E}] < e^{-\frac{(\beta-\alpha)^2 p}{12\beta}T_0} < \delta$$

Now using Inequality 7.4 and the inequality above, we have that

$$\Pr[\mathsf{Eve}\text{ wins game } G \cap \overline{E}] - \Pr[\mathsf{Eve}\text{ wins game } G' \cap \overline{E}] > (\delta + \tilde{\varepsilon}) - \delta = \tilde{\varepsilon} \tag{7.5}$$

We now claim that the Eve in Equation 7.5 can be used to construct an efficient distinguisher D that breaks the security of the PRF as follows: D simply simulates to Eve the entire strong fault detection game using Pepper Probing, replacing calls of the Probe function to the PRF or random function with calls to the oracle $\mathcal{O}$ in the PRF distinguishing game. At the end of Eve's execution, F checks if Eve wins and if event $\overline{E}$ happens (*i.e.* that Eve does not forge any acks). Then, the distinguisher D outputs 1 iff Eve wins game $G \cap \overline{E}$. From Equation 7.5 it follows that

$$\Pr_k[D^{f_k(\cdot)} = 1] - \Pr_F[D^{F(\cdot)} = 1] > \tilde{\varepsilon} > \epsilon_{\mathsf{prf}}(n)$$

since we assume $\tilde{\varepsilon}$ is non-negligible, which breaks the security of the PRF, completing our contrapositive argument.

From Equation 7.2, Inequality 7.3, and the contrapositive argument above, we now know that

$$\Pr[\mathsf{Eve}\text{ wins game } G] < \delta + \epsilon_{\mathsf{mac}} + \epsilon_{\mathsf{crh}} + \epsilon_{\mathsf{prf}}$$

and since $\epsilon_{\mathsf{mac}}, \epsilon_{\mathsf{crh}}, \epsilon_{\mathsf{prf}}$ are each negligible, it follows that Pepper Probing is $(\alpha, \beta, \delta, T_0)$-secure as long as each interval lasts for $T_0 = O(\frac{\beta}{p(\beta-\alpha)^2}\log\frac{1}{\delta})$ exchanges. ■

A similar Chernoff bound can be used to show the Pepper Probing is correct.

Theorem 7.4. *Pepper Probing is correct for* $(\alpha, \beta, \delta, T)$ *for any* $0 < \alpha < \beta < 1$ *and* $0 < \delta < 1$ *where probing intervals last for* $T_0 = O(\frac{\alpha}{p(\beta-\alpha)^2}\log\frac{1}{\delta})$ *exchanges.*

Now since $\alpha < \beta$, Theorem 7.3 and Theorem 7.3 imply that Pepper Probing is both correct and strongly $(\alpha, \beta, \delta, T)$-statistically secure when probing intervals last for at least $T_0 = O(\frac{\beta}{p(\beta-\alpha)^2}\log\frac{1}{\delta})$ exchanges.

Remark 7.5. Security fails without pseudo-randomness: Suppose that the Probe function of Equation 7.1 was implemented using a CRC keyed with a secret modulus, as in [8], instead of a PRF. Model the CRC function as $f_k(x) = x \mod k$, and consider the following attack: Eve starts by observing

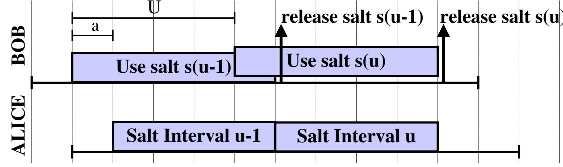

Figure 9: Timing for Salt Probing.

the interaction between Alice and Bob, and recording a list of packets that were not acknowledged by Bob. Then, whenever she sees a new packet that is within a small additive distance of old packet that was not acknowledged, she drops the packet. Thus, Eve can drop non-probe packets with high probability, and she can bias Alice's estimate F below the true fault rate. This attack is possible because the CRC does not use its 'secret key' in 'cryptographically-strong' manner (*c.f.,* our result on the necessity of cryptography, Section 4).

7.3 Efficiency of Pepper Probing

For a reasonable set of parameters, say overhead $p = 0.02$, false-alarm threshold $\alpha = 0.005$, and detection threshold $\beta = 0.01$, and $\delta = .01$ it follows from the statement of Theorem 7.3 (or see Table 2) that Pepper Probing is (α, β, δ)-secure when the protocol runs for at least $T > 1.3 \times 10^6$ packets. If each packet digest is 128 bits long, it follows that Alice needs about $pT \times 128 = 3.3$ Mbits of storage. (While Alice requires this much storage for each Bob network, she can choose to run Pepper Probing with only a small number of Bob networks at a time by ignoring all the acks sent by all other Bob networks.) Pepper Probing incurs only a small communication overhead of p (due only to to acks; Alice's traffic is unmodified). Consider now the Stealth Probing protocol of [3], another secure statistical FD protocol which also uses passive sampling and authenticated acks for a p-fraction of packets. Whereas Stealth Probing requires Alice to authenticate and encrypt all of her traffic, Pepper Probing does not require any modifications to Alice's traffic.

8 Salt Probing: Public-Key Statistical FD

While Pepper Probing requires pairwise symmetric keys between each Alice-Bob pair, Salt Probing requires fewer keys by operating in the public-key setting. While known public-key cryptographic primitives are too computationally expensive to execute at line rate, we bypass this problem by using techniques reminiscent of TESLA[10] [20], so that public-key operations are used very infrequently. Hence, the computational overhead of Salt Probing is almost identical to that of Pepper Probing. Like TESLA, Salt Probing requires that Alice and Bob coarsely synchronize their clocks. Time is divided into *salt intervals*. In each interval Bob uses the *same salt value* to key each probing session with each Alice network.

8.1 Description of Salt Probing

Setup: Alice has some local secret key k_A, which is not shared with anyone; she uses this key to verify data that she sends to Bob and then is sent back to her. Bob has a public key PK_B that is known to Alice, and he keeps secret the corresponding secret key SK_B. Alice and Bob also know a fixed time constant a, which approximately equals 1.5 round trip times (RTT) (*e.g.* $a = 150$ ms). Before the protocol begins, *Alice synchronizes her clock so that it lags Bob's clock by at most a seconds* as follows: At time t_A (on Alice's clock) Alice sends Bob a message $[A, t_A]_{k_A}$ MAC'd using her local secret key. Bob receives this message at time t_B (on Bob's clock) and responds with digitally signed message $[B, t_B, [A, t_A]_{k_A}]_{SK_B}$. (Note that a **digital signature** performs the same function as a MAC, but in

[10] TESLA uses only symmetric-key operations (except if a PKI is used for key exchange). Salt Probing could also be adapted to TESLA's symmetric key setting by adopting TESLA's use of one-way chains [20].

packet digest	ack	Probe
$z_1 = h(d_1)$		Yes
$z_2 = h(d_2)$	$[B, z_2, u,]_{\mathsf{s}_2(u)}$	Yes
$z_3 = h(d_3)$		No
$z_4 = h(d_4)$		No
$z_5 = h(d_5)$		No
$z_6 = h(d_6)$		No
$z_7 = h(d_7)$		No
$z_8 = h(d_8)$	$[B, z_8, u]_{\mathsf{s}_2(u)}$	Yes
$z_9 = h(d_9)$		No
$z_{10} = h(d_{10})$		No
$z_{11} = h(d_{11})$	$[B, z_{11}, u]_{\mathsf{s}_2(u)}$	Yes
$z_{12} = h(d_{12})$		Yes
$z_{13} = h(d_{13})$		No

Figure 10: Alice's table after at the end of salt interval u. Here Alice observes faults during exchanges $1, 12$.

the public-key setting; here the key SK used to sign a message is secret while the key PK used for verification is publicly known.) Alice will accept Bob's time t_B as long as his message is validly signed, contains a valid copy of Alice's message $[A, t_A]_{k_A}$, and arrives within a seconds of time t_A (on Alice's clock). Then, at time $t_A + a$ Alice synchronizes her clock to Bob's time t_B. If, after many attempts, Alice fails to receive a valid response to her synchronization message, then she decides the data path is faulty and raises an alarm. Synchronization happens infrequently (say, once a day).

Bob's algorithm: At the beginning of each salt interval u, Bob randomly chooses a pair of *salt* values $(\mathsf{s}_1(u), \mathsf{s}_2(u))$ that he keeps secret for the duration of the salt interval. Then, Bob runs a slightly modified version of Pepper Probing, replacing the symmetric key $k = (k_1, k_2)$ in Figure 8 and Equation 7.1 with the salt values $(\mathsf{s}_1(u), \mathsf{s}_2(u))$. That is, during salt interval u, Bob runs the Probe function of Equation 7.1 on packet digests $z = h(d)$ using salt $\mathsf{s}_1(u)$ as a key, and for each packet that is a probe, he constructs and sends to Alice an ack of the form $[B, z, u]_{\mathsf{s}_2(u)}$, which contains a packet digest z and a salt interval number u. As shown in Figure 11, Bob reveals the salt $(\mathsf{s}_1(u), \mathsf{s}_2(u))$ to Alice (and to all other source networks connected to him) a seconds after salt interval u ends by sending a (public-key) signed *salt release message* $[B, u, \mathsf{s}_1(u), \mathsf{s}_2(u)]_{SK_B}$.

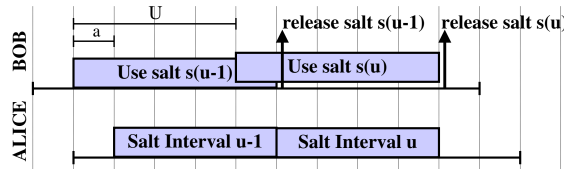

Figure 11: Overview of Timing for Salt Probing.

Alice's algorithm: Because Alice does not know the salt for the duration of the salt interval, she is unable to run Probe 'in real time' as she sends each packet (as she did in Pepper Probing). Instead, Alice will store a packet digest for *each* of the packets that she sends to Bob as shown in Figure 10. Whenever Alice receives an ack $[B, z, u]_{\mathsf{s}_2(u)}$ from Bob, she stores the ack in her table only if the ack is tagged with the current salt interval u and a packet digest z that matches a packet digest that she has stored in her table. While on the surface it may seem that in Salt Probing Alice needs to store information about each packet she sends to Bob for the duration of a salt interval, storage requirements can be reduced without compromising security if Alice *independently subsamples* packets with (truly random) probability q (and then stores only the subsampled packets in memory).

If Alice fails to receive a validly signed salt release message $[B, u, \mathsf{s}_1(u), \mathsf{s}_2(u)]_{SK_B}$ after salt interval u

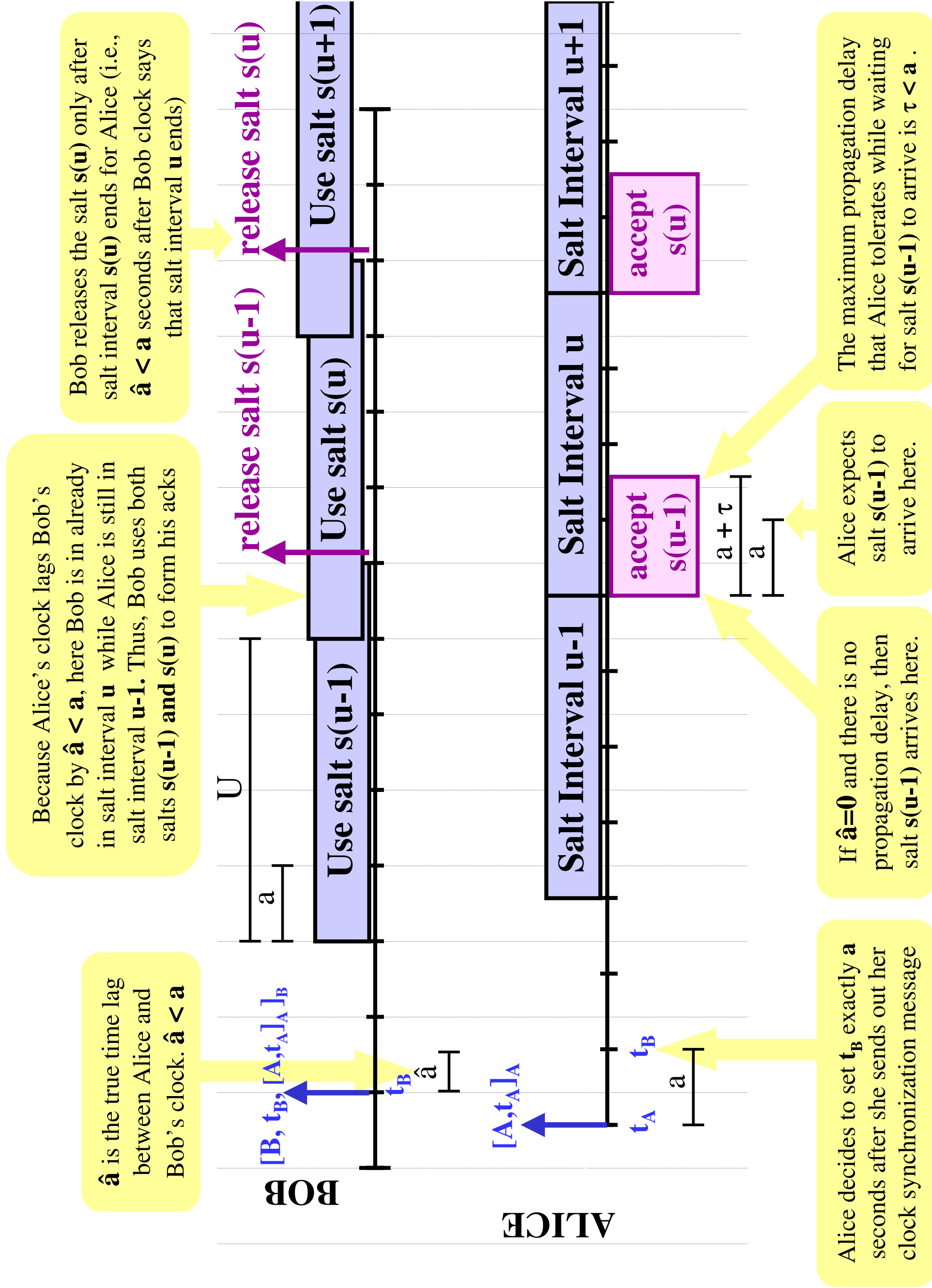

Figure 12: Detailed Overview of Timing for Salt Probing.

ends, she concludes that her data path to Bob is faulty and raises an alarm. Otherwise, Alice uses salt $\mathsf{s}_1(u)$ (from the salt release message) to run the Probe function on the packet digests in her table, and salt $\mathsf{s}_2(u)$ to verify the acks in her table. Then, to compute an estimate F of the fault rate that occurred during the salt interval, Alice counts the fraction of exchanges for which $\mathsf{Probe}(z)$ = Yes and no valid ack is stored in her table. Finally, Alice raises an alarm if $F > \frac{\alpha+\beta}{2}$ and decides that everything was normal if $F < \frac{\alpha+\beta}{2}$.

8.2 Timing for Salt Probing

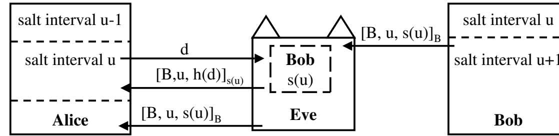

Figure 13: Attack on salt probing without global clocks.

Need for global clock synchronization: In Salt Probing, it is extremely important that Bob releases salt $\mathsf{s}(u)$ only after Alice stops accepting acknowledgements signed by the salt $\mathsf{s}(u)$. To do this, Alice and Bob require some (coarsely) globally synchronized clocks. To see why, consider the (timing-related) attack shown in Figure 13 that could be performed when Alice and Bob's clocks are not synchronized: Suppose Bob releases the salt $\mathsf{s}(u)$ at the end of his salt interval u, and enters salt interval $u+1$. Eve will now trick Alice into thinking that Bob is still in salt interval u as follows: Eve stores the salt message instead of forwarding it Alice. Then, when Eve gets packets from Alice, instead of forwarding these packets to Bob, she uses $\mathsf{s}(u)$ to (simulate Bob by) running the Probe? function and acknowledging the appropriate packets. Eve does this for the a period of time equal to the length of a salt interval, and then forwards the salt message containing $\mathsf{s}(u)$ to Alice. Notice Alice does not realize that her packet failed to reach Bob, because she has received valid acknowledgements and salt message for salt interval u.

Clock Synchronization: As shown in Figure 12, our synchronization protocol allows Alice to synchronize her clock to Bob's clock to within exactly $\hat{a}$ seconds, where $\hat{a} \leq a$.

Releasing salt: Bob releases the salt for interval u only after he is certain that Alice is no longer accepting acks generated using $s(u)$. Since Alice's clock lags Bob's clock by at most a seconds, Bob releases the salt at least a seconds after his clock indicates that salt interval u is over. See Figure 12.

Double salting: Note however from Figure 12, that because Alice's clock lags Bob's clock by at most a seconds, it follows that there will be period of time of length $< a$ where Alice is operating in salt interval $u-1$ while Bob has already moved into salt interval u. To remedy this, during the first a seconds of each salt interval, Bob uses *both* the salt of the current interval $\mathsf{s}(u)$ and the salt from the *previous* interval $\mathsf{s}(u-1)$ in order to create his acks.

Accepting salt: As shown in Figure 12, Alice accepts the salt $s(u)$ only after salt interval u ends (on her clock). Furthermore, since Alice knows that Bob releases the salt a seconds after the salt interval ends (on Bob's clock), Alice will somewhat 'naively' expect the salt $s(u)$ to arrive a seconds after the salt interval ends. Finally, we let τ be the maximum length of time Alice is willing to wait for the salt to arrive (before purging her storage of the table (as in Figure 10) related to salt interval u. We expect that τ is on the order of $\frac{1}{2}$RTT. Therefore, the length of time that Alice should accept the Salt is $a + \tau$ which is bounded by $2a$.

Length of the salt interval: The salt interval lasts for U seconds. To understand how to choose U, first observe that because the length of time required for 'double-salting' is a seconds, it follows that (to avoid frequent 'double-salting' or even 'triple-salting', etc.) we should have $U > a$. Futhermore, the longer U is (relative to a), the less frequently 'double-salting' is required. On the other hand, the longer the salt interval lasts, the more storage is required (see Figure 10).

Next, observe that a must be larger than 1 RTT, (because a is the length of time Alice is willing to wait before receiving a response from Bob to her synchronization message, what on average takes on 1 RTT to arrive). We add a safety factor of 50% and assume that $a \approx 1.5RTT$.

Also, from Figure 12, we observe that $\hat{a}$, the true time lag between Alice and Bob's clock is equal to a minus the one-way propagation delay that Alice's synchronization message $[A, z]_{k_A}$ experienced before it reached Bob. It follows that on average, $\hat{a} = a - \frac{1}{2}\text{RTT}$.

8.3 Security of Salt Probing

Proving the security of Salt Probing is very similar to proof of security for Pepper Probing.

Theorem 8.1. *Salt Probing is a strongly (α, β, δ)-statistically secure fault detection scheme for $T_0 = O(\frac{\beta}{pq(\beta-\alpha)^2} \log \frac{1}{\delta})$ sufficiently large.*

Proof. The security of Salt Probing relies on (i) Eve's inability to skew Alice's synchronization to Bob's clock beyond a seconds, (ii) Eve's inability to forge a salt release message, and (iii) Eve's inability to bias Alice's estimate of the fault rate during a salt interval.

We start by arguing that (i) Eve cannot skew Alice's synchronization to Bob's clock beyond a seconds. To see why, first observe that Alice will reject the synchronization reply $[B, t_B, [A, t_A]_{k_A}]_{SK_B}$ if Eve delays it for more than a seconds. Next, because the sync reply includes Alice's sync request $[A, t_A]_{k_A}$, which is timestamped relative to Alice's clock, it follows that the Eve cannot skew Alice's synchronization by replaying an old synchronization reply from Bob. It follows that the only way Eve can skew Alice's synchronization to Bob is to either (a) forge Alice's MAC on a synchronization request $[A, t_A]_{k_A}$ (and then ask Bob to produce a synchronization reply to the forged synchronization request, that Eve gives to Alice after some time lag of greater that a seconds) or (b) to forge Bob's digital signature on his synchronization reply. By the security of the MAC and digital signature schemes used in Salt Probing we have that

$$\Pr[\mathsf{Eve} \text{ wins by skewing synchronization}] < \epsilon_{\mathsf{mac}} + \varepsilon_{\mathsf{sig}}$$

For the duration of salt interval u on Alice's clock, Alice will store in table (see Figure 10) packets and acks stamped with salt interval u, which she then verifies using the salt she obtains after the salt interval ends. We argue that (ii) Eve cannot trick Alice into using an incorrect salt value to verify the acks she receives unless she Eve forges the digital signature on the salt message $[B, u, \mathsf{s}_1(u), \mathsf{s}_2(u)]_{SK_B}$. To see why, observe that the salt release message is stamped with the salt interval u. Since by assumption Bob is honest, the salt release message stamped with interval u will always contain the salt used in interval u[11] It follows that if Eve managed to produce a salt release message stamped with interval u but containing a salt value $s'(u)$ that is different from the salt value $s(u)$ actually used by Bob during salt interval u, then she must have forged Bob's signature. (The formal reduction from Eve that breaks the security of FD game by forging a salt release message to Fred to breaks the security of a digital signature game procceeds exactly along these lines.) As such, it follows that

$$\Pr[\mathsf{Eve} \text{ wins by forging salt release}] < \varepsilon_{\mathsf{sig}}$$

Now, conditioned on Eve not skewing synchronization and Eve not forging salt release, observe that the salt $s(u)$ used to construct and verify acks during salt interval u kept secret for the duration of salt interval u on Alice's clock. (See Section 8.2 and Figure 12 for more details). Furthermore, at the end of the salt interval, Alice knows the correct salt value $s(u)$ that is used to verify her messages. It follows that we can use a similar argument that we used in the Theorem 7.3 to prove that (iii) Eve cannot bias Alice's measurement of the fault rate during the salt interval.

The only wrinkle in the argument here is that now Alice is subsampling packets with probability q (*i.e.* Alice is only storing packets in her table with probability q. For the remaining $1 - q$ packets she sends,

[11] TESLA [20] does not assume that 'Bob' is honest. Instead, Bob commits to the salt (using a hiding and binding commitment scheme [9]) at the beginning of the salt interval, and a the end of the salt interval Bob opens his commitment.

she simply 'forgets' about them after she sends them.) As in Theorem 7.3, we denote by game G the strongly secure FD game using Salt Probing protocol, which makes use of a PRF in the Probe function, and Game G' as the strongly secure FD game is played for a FD protocol that is just like Salt Probing *except that the PRF in the* Probe *function is replaced with a random function.*

Now in game G', because Alice and Bob use a truly random function decide which packets become probes, it follows from the definition of the Probe function (Equation 7.1) any packet is probe with truly random probability p. From the properties of random functions imply that in game G' each packet is *independently* sampled by Alice (*i.e.* is designated as a probe by Alice and Bob AND is subsampled by Alice) with truly probability pq. (Recall that by definition Alice subsamples truly randomly). Using the identical argument used in Theorem 7.3, we can show that in Game G', Alice's count of the number of packets in her table (see Figure 10) that are not correctly acknowledge is is distributed as a binomial random variable $X \sim B(\kappa(u)T, pq)$ (*i.e.* a binomial random variable sampling with probability pq out of $\kappa(u)T$ elements).

Applying now applying logic similar to that used in Theorem 7.3 (replacing p with pq), we have that

$$\Pr[\mathsf{Eve}\text{ skews Alice's fault rate estimate in game } G] < e^{-\frac{(\beta-\alpha)^2 pq}{12\beta}T_0} + \epsilon_{\mathsf{mac}} + \epsilon_{\mathsf{crh}} + \epsilon_{\mathsf{prf}}$$

which is bounded by $\delta + \epsilon_{\mathsf{mac}} + \epsilon_{\mathsf{crh}} + \epsilon_{\mathsf{prf}}$ if we set $T_0 = O(\frac{\beta}{pq(\beta-\alpha)^2} \ln \frac{1}{\delta})$. It follows that Eve cannot skew Alice's fault rate estimate during an interval lasting for at least T_0 exchanges.

It follows that

$$\begin{aligned}\Pr[\mathsf{Eve}\text{ wins}] \leq & \Pr[\mathsf{Eve}\text{ skews synchronization}] + \Pr[\mathsf{Eve}\text{ forges salt release}] \\ & + \Pr[\mathsf{Eve}\text{ skews Alice's fault rate estimate}] \\ \leq & (\epsilon_{\mathsf{mac}} + \epsilon_{\mathsf{sig}}) + \epsilon_{\mathsf{sig}} + (\delta + \epsilon_{\mathsf{mac}} + \epsilon_{\mathsf{crh}} + \epsilon_{\mathsf{prf}})\end{aligned}$$

which completes the proof of security. ■

Again, a similar Chernoff bound can be used to show that Salt Probing is correct.

Theorem 8.2. *Salt Probing is correct for* $(\alpha, \beta, \delta, T)$ *for any* $0 < \alpha < \beta < 1$ *and* $0 < \delta < 1$ *where probing intervals last for* $T_0 = O(\frac{\alpha}{p(\beta-\alpha)^2} \log \frac{1}{\delta})$ *exchanges.*

Again since $\alpha < \beta$, Theorem 8.1 and Theorem 8.1 imply that Salt Probing is both correct and strongly $(\alpha, \beta, \delta, T)$-statistically secure when probing intervals last for at least $T_0 = O(\frac{\beta}{p(\beta-\alpha)^2} \log \frac{1}{\delta})$ exchanges.

8.4 Efficiency of Salt Probing

While on the surface it may seem that in Salt Probing Alice needs to store information about each packet she sends to Bob for the duration of a salt interval, recall that storage requirements can be reduced without compromising security if Alice *independently subsamples* packets with (truly random) probability q (doing this of course lengthens the duration of the probing session by a factor of $\frac{1}{q}$, see Theorem 8.1).

For reasonable parameters: $p = 0.02$, false-alarm threshold $\alpha = 0.005$, and detection threshold $\beta = 0.01$, it follows from Theorem 8.1 that Alice must store at least $qT = 1.3 \times 10^6$ entries in her table (Figure 10). If each packet digest $z = h(d)$ is 128 bits long, Alice must store on average about $128(1 + p) + 1 = 131$ bits for each row of her table, she requires about $qT \times 131 \approx 170$ Mbits of storage. Now, assuming that average packet is 3000 bits long, RTTs are on the order of 100 ms, and line rates are 5 Gbps, then about 10^7 packets are sent during one RTT. Then, if a salt interval lasts for about 5 RTTs, it follows that it is sufficient for Alice to subsample packets at rate $q = \frac{1.3 \times 10^6}{5 \times 10^7} \approx 2.6\%$ in order to obtain an unbiased measurement during a salt interval.

	Per-Packet Ack	**Pepper**	**Salt**	**Stealth** [3]
communication overhead (due to acks)	100%	p	p	p
key structure	symmetric	symmetric	public	symmetric
clock synch between Alice and Bob?	No	No	Coarse	No
modifications to Alice's traffic?	No	No	No	Yes
minimum duration of probing session, in packets	1	$\frac{64\beta}{p(\beta-\alpha)^2}$	$\frac{1}{q}\frac{64\beta}{p(\beta-\alpha)^2}$	$\frac{64\beta}{p(\beta-\alpha)^2}$
number of packet digests stored at Alice	all	p-fraction	q-fraction	p-fraction

Table 2: Comparison of (α, β, δ)-secure FD protocols for $\delta = 0.01$.

9 Comparison of Fault Detection Protocols

We compare Salt Probing, Pepper Probing, Per-Packet Ack and Stealth Probing [3] in Table 2.

While the Per-Packet Ack provides the strongest security guarantee, enabling Alice to detect if even a single one of her packets is dropped or modified, the scheme suffers from 100% communication overhead (even though, in practice acks can be batched together and sent in a single packet, the requirement for unambiguousness implies that the batched together ack is likely to be large because it must include information about each individual packet sent.) The Per-Packet Ack protocol is probably best suited for applications that require a high level of availability and integrity.

Stealth Probing [3] is a secure passive probing protocol, in which Alice designates a p-fraction of data packets as probes. To ensure data integrity and to prevent Eve from distinguishing probes from non-probes, the entire data stream is encrypted and MAC'd. While the Stealth Probing reduces communication overhead to p, the protocol still requires Alice to modify all the traffic she sends. Thus, Stealth Probing must be built into the router's packet processing path.

Like Stealth Probing, Pepper Probing protocol has limited communication overhead. Pepper Probing does not modify the packets sent by Alice, and thus can be implemented as a monitor off of the routers critical packet-processing path. In Pepper Probing the only cryptography that must be computed on every single packet is a PRF (keyed cryptographic hash) operation that can be efficiently and cheaply implemented in hardware. However, because Pepper Probing is designed for the symmetric key setting, it requires pair-wise security associations between every Alice-Bob pair. Furthermore, a Bob network engaging in probing with many Alice source networks will need to look up the symmetric key required to construct the appropriate ack packets for each packet he receives.

Finally, Salt Probing inherits the limited cryptographic and communication overhead of Pepper Probing (with the addition of a few, infrequent, public-key operations) while also scaling to setting of many Alice-Bob pairs. In contrast to Pepper Probing, in Salt Probing, Bob does not need to perform a key lookup for each packet he receives, since the same salt is used for each of Bob's probing sessions with each different Alice source network. Furthermore, because the protocol works in the public key setting, for a network of M edge-networks we require only M keys (rather than M^2 keys as in the symmetric setting). However, the protocol requires (coarsely) globally synchronized clocks; we presented a simple protocol for clock synchronization in Section 8.

10 Conclusions and Implications

Because FD protocols require only pairwise participation from nodes, deployment of FD can proceed in an incremental fashion that is compatible with incentives for informing routing decisions at the network edge. However, when we consider the placement and selection of FD protocols, natural questions arise about the division of labour between the end host and the edge router. We argue that the placement of FD protocols depends on the parties responsible for providing confidentiality and driving routing decisions. Consider, for example, the following three scenarios:

- Firstly, if the end host both provides confidentiality and makes routing decisions, then host-based cryptography, *e.g.* SSL, is appropriate for detecting faults on a per-packet basis. This may be the case in a new routing architecture, or a special-purpose secure network.
- On the other hand, if the edge router both provides confidentiality and makes routing decisions, then Stealth Probing [3]'s secure statistical FD protocol, that also performs edge-to-edge encryption of traffic, is most appropriate. This particularly appropriate when two stub ASes belong to the same institution.
- Finally, when routers make routing decisions but confidentiality is optionally provided at the end hosts, our Pepper and Salt Probing protocols (designed to avoid modifying and re-encrypting traffic at the edge router) are most appropriate. We believe that this is the case for the majority of scenarios at the edge of the Internet (*e.g.* a residential broadband provider who runs FD on traffic that users optionally protect with SSL), as well as in the *core* of Internet.

In fact, our Pepper and Salt Probing may even be efficient enough to be deployed in the core of Internet, as part of an architecture where core routers inform their routing decisions by running FD to destination networks.

References

[1] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing packet obituaries. *ACM HotNets-III*, 2004.

[2] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Highly secure and efficient routing. In *IEEE INFOCOM*, 2004.

[3] I. Avramopoulos and J. Rexford. Stealth probing: Efficient data-plane security for ip routing. *USENIX*, 2006.

[4] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens. An on-demand secure routing protocol resilient to byzantine failures. In *ACM WiSE*, 2002.

[5] F. Baccelliand, S. Machiraju, D. Veitch, and J. Bolot. The role of PASTA in network measurement. In *ACM SIGCOMM*, 2006.

[6] D. D. Clark, J. Wroclawski, K. R. Sollins, and R. Braden. Tussle in cyberspace: defining tomorrow's Internet. *IEEE/ACM Trans. Netw.*, 13(3), 2005.

[7] M. Crovella and B. Krishnamurthy. *Internet Measurement: Infrastructure, Traffic and Applications*. Wiley, 2006.

[8] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *ACM SIGCOMM*, 2000.

[9] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2007.

[10] D. Harkins and D. Carrel. Rfc2409: The internet key exchange (ike).

[11] A. Herzberg and S. Kutten. Early detection of message forwarding faults. *SIAM J. Comput.*, 30(4):1169–96, 2001.

[12] IETF. Working Group on IP Performance Metrics (ippm). *http://www.ietf.org/html.charters/ippm-charter.html*.

[13] R. Impagliazzo and M. Luby. One-way functions are essential for complexity based cryptography. *FOCS*, 1989.

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

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

[16] A. T. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage. Fatih: detecting and isolating malicious routers. In *IEEE Internat. Conf. Dependable Systems and Networks*, 2005.

[17] NIST. Hash function workshop. *http://csrc.nist.gov/pki/HashWorkshop/*.

[18] V. Padmanabhan and D. Simon. Secure traceroute to detect faulty or malicious routing. *HotNets-I*, 2002.

[19] R. Perlman. *Network Layer Protocols with Byzantine Robustness*. PhD thesis, MIT, 1988.

[20] A. Perrig, R. Canetti, D. Song, and J. D. Tygar. Efficient authentication and signing of multicast streams over lossy channels. In *IEEE Security and Privacy Symposium*, 2000.

[21] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving accuracy in end-to-end packet loss measurement. In *ACM SIGCOMM*, 2005.

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