Distillation Codes
and Applications to DoS Resistant Multicast Authentication

Chris Karlof, UC Berkeley
Naveen Sastry, UC Berkeley
Yaping Li, UC Berkeley
Adrian Perrig, CMU
J. D. Tygar, UC Berkeley

Abstract

We introduce distillation codes, a method for streaming and storing data. Like erasure codes, distillation codes allow information to be decoded from a sufficiently large quorum of symbols. In contrast to erasure codes, distillation codes are robust against pollution attacks, a powerful class of denial of service (DoS) attacks in which adversaries inject invalid symbols during the decoding process.

We examine applications of distillation codes to multicast authentication. Previous applications of erasure codes to multicast authentication are vulnerable to low bandwidth pollution attacks. We demonstrate pollution attacks against previous approaches which prevent receivers from verifying any authentic packets. To resist pollution attacks, we introduce Pollution Resistant Authenticated Block Streams*, which have low overhead and can tolerate arbitrary patterns of packet loss within a block up to a predetermined number of packets. In the face of 40Mb/s of attack traffic, PRABS receivers successfully authenticate the stream and consume only 10% of their CPU.*

1. Introduction

Single-source multicast enables a sender to efficiently disseminate digital media to a large audience, but to defend against adversaries who inject bogus packets, receivers must verify the authenticity of packets. One approach to multicast authentication is *signature amortization*. Signature amortization schemes divide the multicast stream into *blocks* of sequential packets and authenticate all the packets in a block with a single signature. Signature amortization is a compelling approach to multicast authentication because it distributes the communication and computation overhead of a digital signature over many packets.

One challenge in signature amortization schemes is robustness to packet loss. Receivers need the digital signature to verify the authenticity of the packets in the block, but the best way to reliably transmit the signature requires consideration. Including the signature in every packet is robust to packet loss, but incurs high overhead. Including a few bytes of the signature in each packet is space efficient, but is not robust to loss. Signature amortization schemes differ mainly in their solution to this problem.

Three previous approaches are hash graphs [11, 18, 26, 33], the Wong-Lam scheme [35], and erasure codes [21, 22]. All are vulnerable to denial of service attacks. Hash graph protocols construct a directed graph over the packets where each node in the graph contains the hash values of the neighbors on its incoming edges. The hash graph terminates with a signature packet, which authenticates a handful of the nodes in the graph. If the signature packet is not lost and there exists a path from a particular packet to the signature, the receiver authenticates the packet by traversing the hash path to the digital signature and verifying the signature.

The Wong-Lam [35] scheme constructs a Merkle hash tree over the packets in the block and signs the root of the tree. Each packet contains the signature and the nodes in tree necessary to reconstruct the root. By including the signature in every packet, each packet is individually verifiable. Receivers authenticate each packet by reconstructing the root value of the tree and verifying the signature.

Hash graph protocols and the Wong-Lam scheme are vulnerable to signature flooding attacks. An adversary flooding the stream with invalid signatures will overwhelm the computational resources of receivers attempting to verify the signatures. Additionally, in hash graph protocols, adversarial loss patterns can cause denial of service. For example, if an adversary causes the loss of all signature packets, nothing is verifiable.

Several researchers advocate the use of erasure codes [15, 16, 28, 29] for signature amortization [21, 22]. Erasure codes are a mechanism that allow receivers to decode messages from a sufficiently large quorum of encoding symbols. Erasure codes are robust to arbitrary patterns of loss among the encoding symbols as long as the decoder receives a sufficiently large subset of them. Multicast authentication protocols using erasure codes are robust to packet loss and have low overhead. However, erasure codes are designed to handle only a specific threat model: packet loss. Erasure codes assume that symbols are sometimes lost but not corrupted in transit; this is the *erasure channel* model.

Unfortunately, the assumptions that underlie erasure codes are unrealistic in hostile environments. Adversaries can *pollute* the message stream by injecting invalid symbols. We call this a *pollution attack*. If an erasure code uses an invalid symbol as input to its decoding algorithm, it will reconstruct invalid data. The communication model that incorporates this more realistic threat is the *polluted erasure channel*, in which valid symbols can be lost, and an adversary can inject additional invalid symbols claiming to be valid. Polluted erasure channels more accurately model multicast environments: malicious end hosts and routers can observe, inject, modify, delay, and drop messages in an erasure encoded multicast stream.

This paper introduces and gives efficient constructions of *distillation codes*, which are robust against pollution attacks, signature flooding, and adversarial loss patterns. We make the following contributions:

- We introduce the notion of *pollution* to erasure channels, which allows us to more accurately model the threats of multicast data dissemination. We also introduce *pollution attacks* and demonstrate how low bandwidth pollution attacks can cause denial of service for erasure codes.

- We introduce distillation codes; we show that distillation codes function well in the polluted erasure channel model, and prove that they are resistant to pollution attacks.

- We use distillation codes to construct a new multicast authentication protocol: Pollution Resistant Authenticated Block Streams (PRABS). PRABS can tolerate arbitrary patterns of packet loss within a block up to a predetermined number of packets and are resistant to pollution attacks on receivers. Figure 1 compares PRABS to existing multicast authentication protocols.

- We present measurements of an implementation of distillation codes and PRABS. These measurements demonstrate the effectiveness of distillation codes against pollution attacks.

Scheme	Overhead (bytes)	Denial of service vulnerabilities
Hash graphs [11, 18, 26, 33]	≈ 40–50	Signature flooding & adversarial loss
Wong-Lam [35]	188	Signature flooding
SAIDA[23]	22	Pollution attacks
Pannetrat-Molva[21]	12	Pollution attacks
PRABS	65	—

Table 1. Comparison of PRABS to existing multicast authentication protocols.

The overhead was computed assuming 80 bit cryptographic hashes, 128 byte RSA signatures, and 128 packet blocks. For SAIDA and PRABS, we assume up to 64 packet losses per block. For PRABS, we use the optimization described in Section 4.6. Our scheme, PRABS, is resistant to pollution attacks, signature flooding, and adversarial loss patterns.

2 Preliminaries

2.1. Broadcast and multicast authentication

Disseminating information from a server in a broadcast setting to multiple receivers demands a mechanism for guaranteeing the authenticity of the data stream. We need a space efficient and computationally lightweight authentication mechanism, especially if the receivers are embedded devices. In unicast settings, symmetric key cryptography can provide an efficient solution to the authentication problem [5]. Naively extending such schemes to a multicast setting by distributing the secret key to all the receivers is not secure: any receiver can forge messages using the secret key.

The goals and requirements of broadcast/multicast authentication are as follows:

- *Packet authenticity.* Each receiver can verify that packets originated from the sender and were unmodified in transit. Receivers must be able to distinguish traffic injected by other parties, including any of the receivers.

- *Packet loss robustness.* Receivers can authenticate each packet despite the loss of a fixed fraction of the total packets.

- *Loss model independent.* In addition to packet loss robustness, receivers can verify packets even when the loss is bursty, correlated, or in any other pattern.

- *Denial of service (DoS) resistant.* Receivers can resist denial of service attacks against their resources.

2.2. Erasure codes

An erasure code [15, 16, 28, 29] is an *encoder* and *decoder* that use forward error correction to tolerate loss. The encoder redundantly encodes information into a set of symbols. If the decoder receives sufficiently many symbols, it can reconstruct the original information. An

(n, t) erasure encoder generates a set S of n symbols $\{s_1, s_2, \ldots, s_n\}$ from the input. The decoder can tolerate a loss of up to t packets, i.e., it can reconstruct the original data given any $n - t$ symbols from S.

Reed-Solomon, Tornado, and LT codes are examples of erasure codes. Reed-Solomon [29] codes typically require $O(n^2)$ time to encode and decode; Tornado and LT codes [15, 16] require $O(n)$ time. Although Reed-Solomon codes are slower, they have the advantage that reconstructing the original data is guaranteed to be successful if the decoder has at least $n - t$ encoding symbols. With Tornado and LT codes, reconstructing with at least $(n - t) \cdot (1 + \epsilon)$ encoding symbols is successful with high probability for $\epsilon \approx 0.05$.

2.3. Terminology and assumptions

We assume there is a single party authorized to encode and send messages. We refer to this party as the legitimate encoder. If the legitimate encoder encodes and sends a message D over the channel to the decoder, D is said to be *valid*. If a message D was never encoded and sent by the legitimate encoder over the channel, D is said to be *invalid*.

Let S be a set of n symbols generated by an erasure encoder with input D. We assume n and the loss parameter t are fixed and known to the encoder and the decoder. Each symbol is an ordered pair (s_i, i), $1 \leq i \leq n$, so each symbol contains its index value. The symbols in S are *valid symbols of* D if this encoding process is executed by the legitimate encoder; other ordered pairs are *invalid symbols*. We are concerned about invalid symbols injected by an adversary. Given a set of symbols which includes valid symbols and possibly invalid symbols, the decoder produces a *candidate reconstruction* R. The reconstruction R is *valid* when $R = D$ for some valid D and *invalid* otherwise. We assume erasure decoding with at least $n - t$ valid symbols of D and no other symbols will result in a valid reconstruction of D.

We assume the encoder and decoder have access to $\text{TAG}(\cdot)$ and $\text{VALIDATE}(\cdot)$ algorithms, respectively. The $\text{TAG}(\cdot)$ algorithm augments its input with some additional information that enables the $\text{VALIDATE}(\cdot)$ algorithm to verify its authenticity. For correctness, we require $\text{VALIDATE}(\text{TAG}(R))$ = **true** for all R. To guarantee authenticity, we assume it is difficult for an adversary to forge R such that $\text{VALIDATE}(R)$ = **true**. We also assume the existence of an algorithm $\text{STRIP}(\cdot)$ that strips off the authentication information added by $\text{TAG}(\cdot)$. One possible instantiation of ($\text{TAG}(\cdot)$,$\text{VALIDATE}(\cdot)$) is public key signature generation and verification. See Appendix A for a formal treatment of our authenticity requirements for ($\text{TAG}(\cdot)$,$\text{VALIDATE}(\cdot)$).

To enable decoders to determine whether a candidate reconstruction is valid, we will erasure encode $\text{TAG}(D)$ rather than D. Then given a candidate reconstruction R, a decoder determines its authenticity by checking whether $\text{VALIDATE}(R)$ = **true**. We refer to the process of applying $\text{VALIDATE}(\cdot)$ to a candidate reconstruction as *reconstruction validation*.

2.4. Pollution attacks: DoS against erasure codes

Adversaries can disrupt the decoding process by introducing invalid symbols. If the decoder uses an invalid symbol, it will generate an invalid reconstruction, causing denial of service. We call this a *pollution attack*, and refer to an erasure channel with pollution attacks as a *polluted erasure channel*.

Decoders can easily recover from pollution attacks with only a small number of invalid symbols and no lost valid symbols. Since both valid and invalid symbols contain an index, the decoder simply looks for duplicate indices and drops both symbols. If at least $n - t$ symbols remain after dropping duplicates, the decoder will recover the valid reconstruction.

Recovery becomes more difficult as the number of invalid symbols increases. For example, suppose the decoder receives only the first $n - t$ valid symbols and an adversary injects one invalid symbol in each of those positions. The decoder cannot simply drop the duplicates since no symbols will remain. Alternatively, the decoder could select one symbol from each position, execute the decoding algorithm, and apply $\text{VALIDATE}(\cdot)$ to verify the authenticity of the candidate reconstruction. This approach is ill-fated: the decoder is successful only if it is lucky enough to select the valid symbol in every position. This event has probability $\frac{1}{2^{n-t}}$, and in the worst case, the decoder will produce 2^{n-t} candidate reconstructions before the valid one is found.

2.5. Threat model

We assume that an adversary can observe, inject, modify, delay, and drop traffic in the channel between the sender and receiver. An adversary could be a compromised router on the path between the sender and receiver, for example.

Denial of service attacks DoS attacks take many forms, depending on the resource they are trying to exhaust. An adversary can attack the sender, the network infrastructure between sender and receiver, and the receiver. In the broadcast setting, the sender does not accept any data from the network, so we will assume that the sender is not susceptible to DoS attacks. We also do not consider bandwidth exhaustion attacks, as they are outside of the scope of this paper. A receiver has little recourse if its last hop router drops all its traffic or thousands of zombie machines flood and overload its last hop link. To recover from these attacks, receivers must rely on help from the infrastructure to detect the problem and take appropriate action. Recent research results address these challenges [2, 3, 10]. However, we must consider DoS attacks against the receiver's computation and storage resources. An attacker should not be able to exhaust these resources to cause DoS.

The *attack factor* is the ratio of the bandwidth of injected invalid traffic to the bandwidth of valid traffic. For example, an attack factor of five implies that for every 1000 bytes of legitimate transmitted data, an adversary injects 5000 bytes of invalid data. We are primarily interested in medium bandwidth pollution attacks, e.g., up to an attack factor of ten. We assume that beyond these values, the adversary will saturate the channel and cause large packet loss within the network.

2.6. Cryptographic primitives

Universal one-way hash functions We assume the existence of families of universal one-way hash functions (UOWHFs) [19]. UOWHFs satisfy a property known as target collision-resistance (TCR) [7]. U is called a family of UOWHFs if for all polynomial-time adversaries A, A has low probability in winning the following game: A first chooses a message M, and then A is given a random $h(\cdot) \in U$. To win, A must output $M' \neq M$ such that $h(M') = h(M)$. This differs from any collision-resistance (ACR), in which the adversary has the freedom to choose both M and M' after she is given $h(\cdot)$. TCR has two advantages over ACR: (1) Since TCR is a weaker notion, it is believed to be easier to achieve in concrete instantiations. (2) Since M is specified before the hash function $h(\cdot)$, birthday paradox attacks to find collisions do not directly apply, and the hash output can be half the size of an ACR hash function.

In the multicast setting, adopting TCR allows adversaries to have complete control over the underlying data in a stream, but only before transmission starts. If this assumption does not hold, we must replace most applications of TCR hash functions with ACR hash functions, and that would increase our overhead by a factor less than two. For the remainder of this paper, we assume all collision-resistant hash functions are TCR.

Merkle hash trees Merkle hash trees [17] are a mechanism for computing a single cryptographically secure hash digest over a set of data items. Merkle hash trees are constructed in the following manner. Let $h(\cdot)$ be a collision-resistant hash function and let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of data items. For the sake of simplicity, suppose that $n = 2^{\ell-1}$ for $\ell > 1$. Then, we construct an ℓ-level complete binary tree using the hashes of the data items, $h_i = h(s_i)$, as leaves. Each internal node of the tree is the hash of the concatenation of its two children, as in Figure 1.

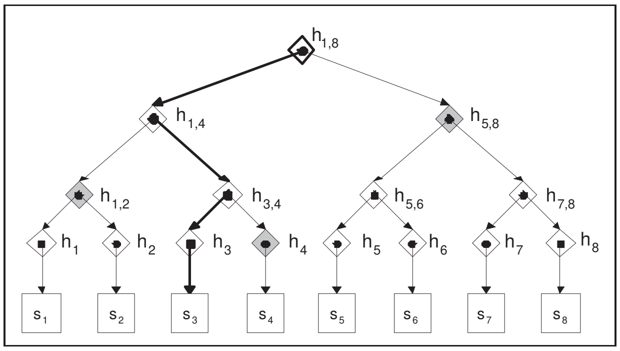


Figure 1. Merkle Hash Trees.
Each leaf node h_i is calculated by taking the hash of the corresponding data item s_i, and each internal node is computed by taking the hash of the concatenation of its two children. The shaded nodes h_4, $h_{1,2}$, and $h_{5,8}$ form a verification sequence of s_3. Given the leaf element s_3 and its verification sequence, one can reconstruct and verify the root value $h_{1,8}$ by computing $h(h(h_{1,2}, h(h(s_3), h_4)), h_{5,8})$.

Merkle hash trees have several nice properties. Each internal node $h_{i,j}$ can be viewed as a hash digest of the data items $s_i, s_{i+1}, \ldots, s_j$, and the root of the tree can be viewed as the hash of the whole set S. If the verifier can verify the authenticity of the root value, for example with a signature, and has all of the data items over which the tree was constructed (and the corresponding positions), she can verify the authenticity of every data item by reconstructing the hash tree and comparing the computed root value with the authenticated root value.

However, the data items are not individually verifiable; to recalculate the root value, the entire set S is needed. To make each s_i individually verifiable, it must be augmented with additional verification information. Given an item s, a verifier can recalculate the root of the tree if it also has the "sibling" nodes on the path from $h(s)$ to the root of the tree. We refer to this sequence of nodes as the *verification sequence* of s. For example, in Figure 1, given element s_3 and its verification sequence $(h_4, h_{1,2}, h_{5,8})$ (the shaded nodes), one can reconstruct and verify the root value $h_{1,8}$ by computing $h(h(h_{1,2}, h(h(s_3), h_4)), h_{5,8})$. In general, each verification sequence requires $\theta(\log(n))$ space, and the associated root value can be reconstructed with $\theta(\log(n))$ hash operations.

3. Distillation Codes

We need a new coding scheme to address polluted erasure channels. We define

> DISTILLATION CODING:
>
> An (n, t) distillation code encodes a message D into a set of n symbols $S = \{s_1, s_2, \ldots, s_n\}$ and transmits them over a polluted erasure channel. The code should satisfy the following properties:
>
> **Authenticity** The distillation decoder should never output an invalid reconstruction.
>
> **Correctness** Suppose for some valid D, T contains at least $n - t$ valid symbols of D. Then execution of the distillation decoder on T will output a valid reconstruction.

We first present and analyze three naive distillation codes.

3.1. Three strawman schemes

Decode all possibilities One simple distillation coding scheme is to modify an erasure decoder to try all possible combinations of $n - t$ symbols and apply VALIDATE$(\cdot)$ to each reconstruction to identify a valid one. If the decoder receives at least $n - t$ valid symbols of some valid D, then eventually it will use a combination containing only valid symbols of D and output a valid reconstruction. This approach has a serious problem: an exponential number of executions of the decoding algorithm are required in the worst case before a valid reconstruction is found.

Digitally sign every symbol A second approach is to use a conventional erasure code and digitally sign each symbol. The decoder authenticates each received symbol and uses only valid symbols in the decoding process. However, all known signature schemes have at least one of following problems: (1) In most signature schemes, generating signatures is expensive. Digitally signing every symbol will overwhelm the computational resources of the encoder for even modest values of n. (2) Signature verification can also be expensive. Since, every injected invalid symbol requires an additional signature verification by the decoder, this creates a potential DoS attack. (3) Some digital signatures are large (e.g., 128 bytes for RSA-1024). When the symbol size is relatively small, including a large signature with every symbol is undesirable. (4) Some one-time signature schemes have relatively small signatures and feature fast signature generation and verification. However, to enable multiple signatures, the fastest variants require large public keys which are impractical to distribute [24, 30]. Some signature schemes may adequately address one or two of these problems, but until there is a signature scheme with short signatures that are fast to generate and verify with a short public key, digitally signing every symbol is not an option.

Error correcting codes A third approach is to use error correcting codes (ECC). An (n, t) ECC encodes D into n symbols such that the decoder can recover D in the presence of a altered and e erased symbols if $2a + e \leq t$. ECC views invalid symbols simply as errors. This approach has several problems: ECC encoding produces longer symbols and ECC decoding is slower than in pure erasure codes. More seriously, ECC is vulnerable to pollution attacks. Consider an attack similar to the one presented in Section 2.4, where the decoder has multiple choices for the symbol to use at a particular position of the input to the decoding algorithm. If the number of positions with multiple choices is $\geq \frac{t}{2}$, then ECC decoding will require exponential time as well.

3.2. Our approach

In this section, we introduce an efficient construction of distillation codes. Before presenting the details of our construction, we review why the strawman schemes are impractical and motivate how we address their shortcomings.

Since adversaries can pollute the channel, decoders must verify the authenticity of reconstructions. The second straw man approach guarantees the decoder uses only valid symbols in the decoding process, but under attack, verifying every symbol overwhelms the receiver. To reduce the number of signature verifications required to obtain a valid reconstruction, we authenticate the reconstructions rather than the symbols. We partition the symbols in a way that distills the valid symbols from the invalid ones, and then decode each of the partitions and authenticate the resulting reconstructions. Since we only consider partitions with at least $n - t$ symbols, adversaries must inject at least $n - t$ symbols to cause an additional decoding and verification operation. If the decoder receives m symbols, then it executes at most $\lfloor \frac{m}{n-t} \rfloor$ decoding and verification operations to recover the valid reconstruction.

Partitioning the symbols Suppose, given a set T containing both valid and invalid symbols, the decoder can run an algorithm PARTITION SYMBOLS that partitions the symbols into $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_k\}$ satisfying the following property:

Definition 1. [Distillation Property] *Let T be a set containing invalid and valid symbols. A set of partitions $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_k\}$ of T satisfies the Distillation*

Property if the following holds: if $\mathcal{D} = \{D : D$ is valid and $\exists t \in T$ such that t is a valid symbol of $D\}$, then for all $D \in \mathcal{D}$, one partition contains exactly all the valid symbols of D.

The distillation decoder can then erasure decode each Q_i to obtain a set of candidate reconstructions. Assuming that for some valid D, at least $n - t$ valid symbols of D were received by the decoder, at least one of these candidates will be valid and can be found by running VALIDATE$(\cdot)$.

The complete specification for efficient distillation decoding is shown in Figure 3. What remains are: (1) an encoding algorithm which enables the decoder to partition the symbols, and (2) an efficient construction of PARTITION SYMBOLS. We describe both in the next section.

3.3. Distillation encoding using one-way accumulators

In this section, we present our implementations of distillation encoding and the algorithm PARTITION SYMBOLS. In both constructions we make use of one-way accumulators.

3.3.1 One-way accumulators

Our construction of PARTITION SYMBOLS relies on a secure *set membership* operation. We have a set $T = \{t_1, t_2, \ldots, t_m\}$ of received symbols and want to partition T into $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_k\}$ which satisfies the Distillation Property. Note that to do this we do not need to determine if a given symbol is valid. Instead, given a valid symbol t of some $D \in \mathcal{D}$ and a set of symbols Q, we would like to determine that $t \in Q$ if Q is a set of valid symbols of D and $t \notin Q$ otherwise. If t is an invalid symbol and Q is a set of valid symbols, we would like to determine $t \notin Q$.

We build a secure set membership operation by using *one-way accumulators* [4, 8, 9, 12, 20, 32]. One-way accumulators combine a set of inputs into a single value called an *accumulator*. Using auxiliary *witness* information, one can authenticate an element as a member of the set. One-way accumulator schemes typically include three functions:[1]

$$\begin{aligned} \mathsf{Accumulate}(S) &\rightarrow a \\ \mathsf{Witness}(s, S) &\rightarrow w \\ \mathsf{Verify}(s, w, a) &\rightarrow b \end{aligned}$$

$\mathsf{Accumulate}(\cdot)$ takes a set S of values as input and outputs its accumulator a. $\mathsf{Witness}(\cdot, \cdot)$ takes an $s \in S$ and the set S and produces a witness w for s. $\mathsf{Verify}(\cdot, \cdot, \cdot)$ takes as input a conjectured element s of S, its witness w, and an accumulator a of S, and outputs $b \in \{\mathbf{true}, \mathbf{false}\}$. If $b = \mathbf{true}$, we determine $s \in S$. Otherwise $s \notin S$.

It must be hard to forge elements of S. That is, it must be hard to find an $s' \notin S$ and w' such that $\mathsf{Verify}(s', w', \mathsf{Accumulate}(S)) = \mathbf{true}$, even if the attacker has seen other valid (s, w) pairs and a.

In many accumulator schemes, one can recover the accumulator $a = \mathsf{Accumulate}(S)$ of a set S given an element $s \in S$ and its witness w. Let this process be represented by the function

$$\mathsf{Recover}(s, w) \rightarrow a.$$

When $\mathsf{Recover}(\cdot, \cdot)$ exists for an accumulator scheme, $\mathsf{Verify}(\cdot, \cdot, \cdot)$ is typically implemented by verifying that $\mathsf{Recover}(s, w) = a$. In our instantiation of PARTITION SYMBOLS, we rely on the $\mathsf{Recover}(\cdot, \cdot)$ function and use it in a special way. In particular, with $\mathsf{Recover}(\cdot, \cdot)$ a verifier does not need to know the accumulator a to determine if two elements s_i and s_j belong to the same set. It only needs to verify that $\mathsf{Recover}(s_i, w_i) = \mathsf{Recover}(s_j, w_j)$. For the sake of brevity, we say that s and w has accumulator value a if $\mathsf{Recover}(s, w) = a$.

3.3.2 Implementing DISTILLATION ENCODE and PARTITION SYMBOLS

To resist pollution attacks, the distillation encoder must enable the decoder to distill the valid symbols of an erasure encoding from a larger set of invalid ones. Our encoding algorithm accomplishes this by accumulating the set of valid symbols and then augmenting each symbol with its witness. The full description of DISTILLATION ENCODE is given in Figure 2.

We can now use the $\mathsf{Recover}(\cdot, \cdot)$ algorithm of the one-way accumulator to implement PARTITION SYMBOLS. $\mathsf{Recover}(\cdot, \cdot)$ is evaluated for each received symbol/witness pair, and symbols with the same accumulator value are put in the same partition. The full specification of PARTITION SYMBOLS is given in Figure 4. For an adversary to cause an invalid symbol to be placed in the same partition as the valid symbols implies that she is able to break the one-way accumulator scheme, i.e., she is able to forge an element of the set protected by the accumulator.

3.3.3 Merkle hash trees as a one-way accumulator

Merkle hash trees [17] are attractive one-way accumulators for distillation codes.[2] When Merkle hash trees serve

[1] For a more rigorous treatment of one-way accumulators, refer to Benaloh and Mare [8] or Baric and Pfitzmann [4].

[2] There are several one-way accumulator schemes [4, 8, 9, 12, 32] based on exponentiation modulo an RSA modulus and the (strong) RSA

DISTILLATION ENCODE:

Input: A message D.
Output: An (n, t) distillation encoding of D, represented as a set $S = \{s_1, s_2, \ldots, s_n\}$.

1. Let $D' = \text{TAG}(D)$.
2. Construct an (n, t) erasure encoding (ERASURE ENCODE$(\cdot)$) of D'. Let $S' = \{s'_1, s'_2, \ldots, s'_n\}$ be the resulting symbols.
3. Construct an augmented set of symbols $S = \{s_1, s_2, \ldots, s_n\}$ where $s_i = (s'_i, w_i)$ and $w_i = \mathsf{Witness}(s'_i, S')$. Output S.

Figure 2. Our algorithm for distillation encoding.

DISTILLATION DECODE:

Input: A set $T = \{t_1, t_2, \ldots, t_m\}$ containing valid and invalid symbols.
Output: A valid reconstruction or ERROR.

1. Invoke PARTITION SYMBOLS on T, resulting in partitions $Q_1, Q_2, \ldots, Q_k$.
2. Throw away all partitions containing less than $n - t$ symbols. Let $Q_1, Q_2, \ldots, Q_\ell$ be the remaining partitions.
3. (a) For all Q_i, replace each $s = (s', w)$ in Q_i with s' (i.e., strip off witness information).
   (b) Execute the erasure decoding algorithm (ERASURE DECODE$(\cdot)$) on each Q_i, resulting in candidate reconstructions $R_1, R_2, \ldots, R_\ell$.
4. Run VALIDATE on each of $R_1, R_2, \ldots, R_\ell$. Let $\mathcal{V} = \{R_i : \text{VALIDATE}(R_i) = \mathbf{true}\}$. If $\mathcal{V} = \emptyset$, output ERROR. Otherwise, randomly select an R_i from $\mathcal{V}$ and output $\text{STRIP}(R_i)$.

Figure 3. Our algorithm for distillation decoding.

PARTITION SYMBOLS:

Input: A set $T = \{t_1, t_2, \ldots, t_m\}$ containing valid and invalid symbols, each augmented with witnesses. Valid symbols are from an (n, t) distillation encoding of D.
Output: A set of partitions $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_k\}$ of T satisfying the Distillation Property.

1. Initialize $\mathcal{Q}$ to the empty list. Let A be a list of accumulator values, initialized to be empty.
2. For $i = 1$ to m do
   (a) For each $t_i = (s_i, w_i)$, calculate $a = \mathsf{Recover}(s_i, w_i)$.
   (b) If $a \notin A$, add a to the end of A and add $\{(s_i, w_i)\}$ to the end of $\mathcal{Q}$. Otherwise, there exists an accumulator a_j in A such that $a = a_j$. Add (s_i, w_i) to Q_j.
3. Output $\mathcal{Q}$.

Figure 4. Implementation of PARTITION SYMBOLS, using one-way accumulators.

as one-way accumulators [12, 17, 32], the size of witnesses grows logarithmically with the size of the accumulated set. This is not a serious problem since Merkle hash trees rely only on cryptographic hash functions, and the accumulator and witness generation and recovery algorithms are fast and efficient.

Given a set $S = \{s_1, s_2, \ldots, s_n\}$, we implement the one-way accumulator operations as follows:

$$\begin{aligned} \mathsf{Accumulate}(S) &\rightarrow h_{1,n} \\ \mathsf{Witness}(s, S) &\rightarrow v \\ \mathsf{Recover}(s, v) &\rightarrow h'_{1,n} \end{aligned}$$

The accumulator value $h_{1,n}$ is the root value of a Merkle hash tree constructed over S as described in Section 2.6. The witness of an element s is the verification sequence v of s in the same hash tree. $\mathsf{Recover}(s, v)$ is implemented by reconstructing the candidate root $h'_{1,n}$ of the hash tree using s and its verification sequence v. Given an authenticated accumulator value $h_{1,n}$, $\mathsf{Verify}(s, v, h_{1,n})$ is implemented by verifying $h'_{1,n} = h_{1,n}$. Using Merkle hash trees, $\mathsf{Accumulate}(\cdot)$ has running time $\theta(n)$, and the other operations have running time $\theta(\log(n))$.

Even without an authenticated root, given the corresponding verification sequences v_i and v_j, we can verify that two elements s_i and s_j are elements of the same set by checking that $\mathsf{Recover}(s_i, v_i) = \mathsf{Recover}(s_j, v_j)$. This is exactly the property needed to implement Step 2 in PARTITION SYMBOLS. Although the decoder cannot determine if an accumulator value is authentic until DISTILLATION DECODE has completed, $\mathsf{Recover}(\cdot, \cdot)$ allows PARTITION SYMBOLS to create a partitions of valid symbols which contain no invalid ones.

3.4. Security analysis: Pollution resistance of distillation codes

We prove three security properties of distillation codes:

Authenticity If (TAG($\cdot$),VALIDATE($\cdot$)) guarantee authenticity, then distillation codes also guarantee authenticity. This means that DISTILLATION DECODE will never output invalid reconstructions.

Correctness Suppose VALIDATE(D) = **true** for all valid D, and (TAG($\cdot$),VALIDATE($\cdot$)) guarantee authenticity. If for some valid D, T contains at least $n-t$ valid symbols of D, then the execution of DISTILLATION DECODE on T will output a valid reconstruction.

DoS-resistance Distillation codes efficiently satisfy the above properties in the presence of medium bandwidth pollution attacks (up to an attack factor of ten).

3.4.1 Authenticity

The authenticity property is that if (TAG($\cdot$),VALIDATE($\cdot$)) guarantee message authenticity, then DISTILLATION DECODE will never output invalid reconstructions. This implies that if DISTILLATION DECODE outputs R, then R was encoded and sent by the legitimate encoder. We prove this property in Appendix A.

3.4.2 Correctness

To prove the correctness property, we must show that if for some valid D, T contains at least $n - t$ valid symbols of D, then the execution of DISTILLATION DECODE on T will output a valid reconstruction.

Theorem 1. *Assume (*TAG($\cdot$),VALIDATE($\cdot$)*) guarantees authenticity of reconstructions,* VALIDATE(D) = **true** *for all valid D, and the underlying one-way accumulator in* DISTILLATION DECODE *resists element forgery. Suppose T contains at least $n-t$ valid symbols of D for some valid D. Then the execution of* DISTILLATION DECODE *on T will output a valid reconstruction.*

Proof. Let $\mathcal{Q} = \{Q_1, \ldots, Q_k\}$ be the set of partitioned symbols resulting from Step 1 of DISTILLATION DECODE. Recall that the decoder partitions the symbols such that all the symbols in Q_i share the same accumulator value (i.e., there exists a such that for all (s', w) in Q_i, $\mathsf{Recover}(s', w) = a$). In particular, if S' is a set of valid erasure code symbols created in Step 2 of DISTILLATION ENCODE applied to D, then for every valid distillation code symbol (s', w) resulting from Step 3 of DISTILLATION ENCODE, $\mathsf{Recover}(s', w) = \mathsf{Accumulate}(S')$. Thus one partition, say Q_v, contains all the valid distillation code symbols of D.

Now we show Q_v contains no invalid symbols. Suppose, by contradiction, that Q_v contains an invalid symbol $(\bar{s}', \bar{w})$. If $a = \mathsf{Accumulate}(S')$, then $\mathsf{Recover}(\bar{s}', \bar{w}) = a$, implying $\mathsf{Verify}(\bar{s}', \bar{w}, a) = \mathbf{true}$. However, since $\bar{s}' \notin S'$, this violates the security condition for element forgery in one-way accumulators (Section 3.3.1). Thus, Q_v contains no invalid symbols and all the received valid symbols of D.

Since, by assumption $|Q_v| \geq n - t$, then $D' = $ ERASURE DECODE(Q_v) is successful. Since VALIDATE(D') = **true**, then $\mathcal{V} \neq \emptyset$ (from Step 4 of DISTILLATION DECODE), and the authenticity property

assumption. However, the size of the accumulator and witness are on the order of the RSA modulus, and the computation required to generate witnesses and verify elements is roughly equivalent to signature generation and verification. We have already argued that this level of overhead is infeasible.

Nyberg [20] proposed a one-way accumulator scheme using only hashing and pseudorandom number generation based on Bloom filters. The main drawback of Nyberg's scheme is that the accumulator value must be on the order of several thousand bytes.

of distillation codes implies $\mathcal{V}$ contains no invalid reconstruction. Therefore, DISTILLATION DECODE outputs a valid reconstruction. □

3.4.3 DoS-resistance

In this section, we show distillation codes can efficiently satisfy the authenticity and correctness properties in the presence of medium bandwidth pollution attacks. This means an adversary cannot cause resource exhaustion denial of service attacks against the receivers.

Computational DoS-resistance We first prove an upper bound on the extra computation an adversary can cause with a pollution attack with attack factor f. Consider the three expensive operations in distillation decoding: hash function applications, erasure decodings, and VALIDATE$(\cdot)$ executions.

Theorem 2. *In* DISTILLATION DECODE*, the most computation an adversary can cause with a pollution attack with attack factor f is $(f+1)\cdot n\cdot(\log(n)+1)$ hash function applications and $\lfloor\frac{f\cdot n}{n-t}\rfloor+1$ erasure decodings and* VALIDATE$(\cdot)$ *executions.*

Proof. To prove this upper bound, we calculate separate upper bounds on the number of hash applications and the number of erasure decodings and VALIDATE$(\cdot)$ executions.

(1) Hash function applications: Every received symbol triggers the execution of the accumulator operation Recover$(\cdot,\cdot)$. With our Merkle hash tree implementation of one-way accumulators, this requires $\log(n)+1$ hash function applications per symbol. Under attack factor f, this results in $(f+1)\cdot n\cdot(\log(n)+1)$ total hash function applications.

(2) Erasure decodings and VALIDATE$(\cdot)$ executions: In the DISTILLATION DECODE algorithm, an erasure decoding is executed if and only if VALIDATE$(\cdot)$ is executed. To trigger an additional erasure decoding and VALIDATE$(\cdot)$ execution, the adversary must cause DISTILLATION DECODE to create an additional partition containing at least $n-t$ symbols. Since symbols are only put into a single partition, creating an additional partition with at least $n-t$ symbols requires the adversary to inject at least $n-t$ symbols. This holds regardless of the adversary's attack method. Thus, with attack factor f, an adversary can create at most $\lfloor\frac{f\cdot n}{n-t}\rfloor$ additional partitions, and DISTILLATION DECODE executes at most $\lfloor\frac{f\cdot n}{n-t}\rfloor+1$ erasure decodings and VALIDATE$(\cdot)$ calls.

Thus, the most computation an adversary can cause with a pollution attack with attack factor f is $(f+1)\cdot n\cdot(\log(n)+1)$ hash function applications and $\lfloor\frac{f\cdot n}{n-t}\rfloor+1$ erasure decodings and VALIDATE$(\cdot)$ executions. □

This analysis demonstrates a nice property of distillation codes: the computational workload of DISTILLATION DECODE scales linearly with the bandwidth of the attack and is independent of the attack traffic pattern. In Section 4, we show why this property of distillation codes is useful for constructing DoS-resistant multicast authentication protocols.

To demonstrate what this upper bound means in concrete terms, consider the case of a medium bandwidth attack ($f\leq 10$) where $n=128$ and $t=64$. Suppose (TAG$(\cdot)$,VALIDATE$(\cdot)$) are RSA-1024 signature generation and verification, symbols are roughly the size of a network packet (1024 bytes), and one message is sent per second. This corresponds to 128 encoding symbols per second of valid traffic, or 1Mb per second. For each valid message sent by the encoder, the decoder will execute at most 11,264 hash function applications and 21 erasure decodings and signature verifications. This is relatively insignificant: with these parameters, a 2.4GHz Pentium 4 machine running Linux can compute on average 70,000 1024-byte SHA1 hashes per second, 1700 RSA-1024 signature verifications per second, and 300 (128,64) Reed-Solomon decoding operations per second. We confirm this analysis experientially in Section 5 with an implementation of distillation codes.

Strong pollution attacks We now demonstrate a pollution attack which achieves this upper bound. To cause PARTITION SYMBOLS to create a partition with $n-t$ symbols, the adversary must generate at least $n-t$ symbol/witness pairs that recover to the same accumulator value. To do this, an adversary generates a set of random symbols and runs Step 2 of DISTILLATION ENCODE to augment the symbols with witness values. The adversary then injects the invalid symbol/witness pairs and repeats this process a total of $\lfloor\frac{f\cdot n}{n-t}\rfloor$ times.

State-holding DoS-resistance We defer analysis of state holding attacks until Section 4.5.3, where we analyze an application of distillation codes to multicast authentication.

3.4.4 Message reordering and replay

The authenticity and correctness properties of distillation codes by themselves do not prevent replay and reordering attacks. The correctness property guarantees that if the legitimate encoder encodes and sends D over the channel and the decoder receives at least $n - t$ valid symbols of D, then DISTILLATION DECODE will output *some* valid reconstruction. D is not guaranteed to be the output because an adversary can replay valid symbols from previous messages into the decoding process. In the Step 4 of DISTILLATION DECODE, $\mathcal{V}$ might contain multiple valid reconstructions, and one will be selected randomly as the output.

A more desirable correctness property is the following: if T contains at least $n - t$ valid symbols of D, then DISTILLATION DECODE will output D. To achieve this, we must add replay protection to (TAG($\cdot$),VALIDATE($\cdot$)). Most any replay protection mechanism is applicable. For example, the TAG($\cdot$) algorithm can append a monotonically increasing counter to D before authenticating it. VALIDATE(D) first verifies the authenticity of D and then verifies the counter value is fresh. To handle adversaries that delay messages (i.e., deliver symbols from multiple fresh valid messages in a single execution of DISTILLATION DECODE), we can extend DISTILLATION DECODE to output multiple valid reconstructions.

4. Pollution Resistant Authenticated Block Streams

Pollution Resistant Authenticated Block Streams (PRABS) use distillation codes to construct authenticated multicast streams. PRABS builds on SAIDA (Signature Amortization using the Information Dispersal Algorithm) [22], a multicast authentication protocol proposed by Park, Chong and Siegel which uses erasure codes. Pannetrat and Molva [21] present a protocol similar to SAIDA which has less overhead, but is slightly more complex. Applying distillation codes to the Pannetrat-Molva construction results in a protocol with about 10 bytes less of overhead per packet, but for the sake of simplicity we focus on SAIDA.

SAIDA is a *signature amortization* scheme. Signature amortization schemes [11, 18, 21, 22, 26, 33, 35] amortize the packet overhead and cost of generating and verifying a signature over many packets by dividing the multicast stream into *blocks*. Each block is then authenticated with a single digital signature.

Signature amortization schemes differ mainly in their method for reliably transmitting the signature to the receivers and individually authenticating each packet in the block. Previous approaches include hash graphs [11, 18, 26, 33], the Wong-Lam scheme [35], and erasure codes [21, 22].

As we discussed in Section 1, these approaches to signature amortization are vulnerable to pollution attacks, signature flooding, and adversarial loss patterns. To defend against adversarial loss patterns, we need a signature amortization scheme that can tolerate arbitrary packet loss within a block up to a predetermined number of packets. SAIDA uses erasure codes to achieve this. However, SAIDA is vulnerable to pollution attacks. PRABS combines distillation codes with the basic approach of SAIDA to resist pollution attacks, signature flooding, and adversarial loss patterns. Before we present the details of PRABS, we first review SAIDA and discuss its vulnerabilities to pollution attacks.

4.1. Signature Amortization using the Information Dispersal Algorithm

In SAIDA, the sender partitions the packet stream into blocks of n consecutive packets. Let $h(\cdot)$ be a cryptographic hash function, $(\mathsf{PKSign}(\cdot,\cdot), \mathsf{PKVerify}(\cdot,\cdot,\cdot))$ be a public key signature scheme, and $(K_{\text{pub}}, K_{\text{priv}})$ be the public/private keypair of the sender. Then for each block $P_j = p_1^j, p_2^j, \ldots, p_n^j$, the sender computes the authentication string $H_j || G_{H_j}$, where $H_j = h(p_1^j)||h(p_2^j)|| \ldots ||h(p_n^j)$ and $G_{H_j} = \mathsf{PKSign}(K_{\text{priv}}, h(H_j))$. Given the hash string H_j and its signature G_{H_j}, a receiver can authenticate any p_i^j in block j by verifying $\mathsf{PKVerify}(K_{\text{pub}}, H_j, G_{H_j}) = \mathbf{true}$ and that $h(p_i^j)$ equals the i-th entry in the hash string H_j.

This process assumes the receiver knows H_j and G_{H_j}. We would like to authenticate every received packet, regardless of the loss pattern of other packets in the block. A naive solution is to include H_j and G_{H_j} with every packet, but this incurs large packet overhead.

SAIDA constructs an (n, t) erasure code over $H_j || G_{H_j}$ and includes one encoding symbol with each packet in the block. Each augmented packet takes the form $p_i^j || s_i^j$, where p_i^j is the i-th packet in the original block and s_i^j is the i-th symbol of the erasure encoding. If no more than t packets are lost in transmission, then the receiver can reconstruct $H_j || G_{H_j}$, verify G_{H_j}, and authenticate each of the received packets.

4.2. Pollution vulnerabilities in SAIDA

SAIDA is vulnerable to pollution attacks. If a single invalid symbol is used in the decoding algorithm, it will fail to reconstruct $H_j || G_{H_j}$. Park, Chong, and Siegel propose using distributed fingerprints to remedy this problem. Distributed fingerprints combine erasure codes with error-correcting codes (ECC) to achieve robustness to symbol modification [14]. In SAIDA, distributed fingerprints augment each s_i^j with a symbol from an (n, t) ECC encoding of $L_j = h(s_1^j)||h(s_2^j)|| \ldots ||h(s_n^j)$, where $h(\cdot)$ is a collision-resistant cryptographic hash function and $\{s_1^j, s_2^j, \ldots, s_n^j\}$ are the erasure encoding symbols of $H_j || G_{H_j}$. The decoder reconstructs L_j using ECC decoding and verifies a candidate symbol $\bar{s}_i^j$ by comparing $h(\bar{s}_i^j)$ to the i-th hash value in L_j.

Park, Chong, and Siegel claim distributed fingerprints prevent DoS in SAIDA.[3] Although distributed fingerprints can handle symbol modification, they were not designed to defend against pollution attacks where many invalid symbols are injected. Since distributed fingerprints rely on ECC, they are vulnerable to the pollution attacks (Section 3.1).

4.3. Using distillation codes to prevent pollution attacks

We now introduce Pollution Resistant Authenticated Block Streams (PRABS). PRABS builds on SAIDA, but uses distillation codes rather than erasure codes to resist pollution attacks.

In PRABS, the sender partitions the packet stream into blocks of n consecutive packets, For block j composed of packets $P_j = p_1^j, p_2^j, \ldots, p_n^j$ the sender computes $H_j \triangleq j||h(p_1^j)||h(p_2^j)||\ldots||h(p_n^j)$. We assume each packet includes its block number and sequence number within the block. Now, rather than encoding $H_j||G_{H_j}$ with an (n,t) erasure code, we use an (n,t) distillation code. More specifically, the sender applies DISTILLATION ENCODE to input $D = H_j$. We define $\textsc{Tag}(\cdot)$, $\textsc{Validate}(\cdot)$, and $\textsc{Strip}(\cdot)$ as follows:

$$
\begin{aligned}
G_D &\triangleq \mathsf{PKSign}(K_{\text{priv}}, h(D)) \\
\textsc{Tag}(D) &\triangleq D||G_D \\
\textsc{Validate}(D||G_D) &\triangleq \textbf{if}(\mathsf{PKVerify}(K_{\text{pub}}, D, G_D)) \\
&\qquad \text{parse } D \text{ as} \\
&\qquad\quad j||h(p_1)||\ldots||h(p_n) \\
&\qquad \textbf{if } j \text{ is fresh } \textbf{return true} \\
&\quad\ \textbf{return false} \\
\textsc{Strip}(D||G_D) &\triangleq D
\end{aligned}
$$

Applying DISTILLATION ENCODE to H_j results in n distillation code symbols $s_1^j, s_2^j, \ldots, s_n^j$. The sender then augments each packet p_i^j in the block with the corresponding symbol s_i^j and multicasts the augmented packets $p_i^j||s_i^j$ to the receivers. This process is repeated for each block.

Let $\{r_1^j, r_2^j, \ldots, r_m^j\}$ be the set of received packets from block j. Since we assume a polluted erasure channel between the sender and the receiver, $\{r_1^j, r_2^j, \ldots, r_m^j\}$ contains some subset of the authentic packets and some number of invalid packets injected by the adversary. Since we are considering pollution attacks on receivers, we are most interested in the case when $|\text{invalid packets}| \gg |\text{valid packets}|$.

The receiver parses each augmented packet r_i^j as $p_i^j||t_i^j$ where p_i^j represents an unaugmented packet of block j and t_i^j represents a symbol of the distillation encoded authentication information H_j. The receiver then applies DISTILLATION DECODE to the received symbols $\{t_1^j, t_2^j, \ldots, t_m^j\}$. In Step 4, the receiver has a set of candidate reconstructions of the form $H_j||G_{H_j}$ and executes $\textsc{Validate}(\cdot)$ on each one to obtain $\mathcal{V}$, the set of valid reconstructions. To account for non-malicious delays and reorderings in the network, we alter DISTILLATION DECODE to output the complete set of valid reconstructions

$$\textsc{Strip}(\mathcal{V}) \triangleq \{\textsc{Strip}(R) : R \in \mathcal{V}\}$$

rather than a single valid reconstruction.

The authenticity property of distillation codes guarantees all $H_j \in \textsc{Strip}(\mathcal{V})$ are authentic, but the receiver still needs to verify the authenticity of the underlying packet stream. For each $H_j \in \textsc{Strip}(\mathcal{V})$, the receiver needs to verify the authenticity of all the packets claiming to be in block j. Recall p_i^j is annotated with its specific position i in block j, so the receiver can authenticate p_i^j by verifying $h(p_i^j)$ is equal to the i-th value in the hash string H_j.

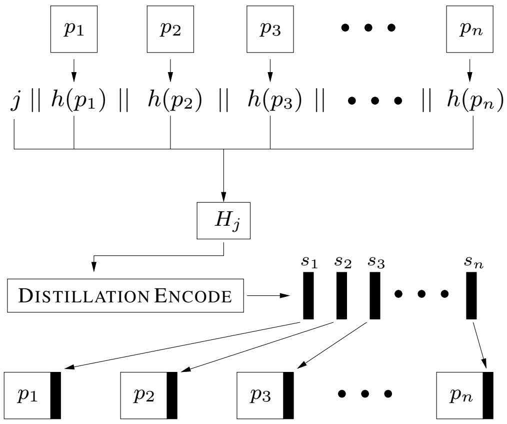
The above description implicitly assumes the adversary mounts the strong pollution attack described in Section 3.4.3 (injecting accumulated random symbol/witness pairs). However, the adversary may also mount a *cut-and-paste attack* where she injects invalid packets augmented with symbol/witness pairs stripped from the valid packets. In the final step of verification described in the previous paragraph, the receiver may have multiple packets, say $p_i'^j, p_i''^j, p_i'''^j$ for a position i that are augmented with the same valid symbol. In this case, the receiver will compute the hash of each of these packets and compare with the i-th position in the hash string H_j to find the authentic packet.

For each block, if no more than t out of n authentic packets are lost in transmission, PRABS can authenticate all received packets in the block regardless of the pattern of loss. Furthermore, in contrast to SAIDA, PRABS is resistant to pollution attacks. The operation of a PRABS sender and receiver is detailed in Figure 5.

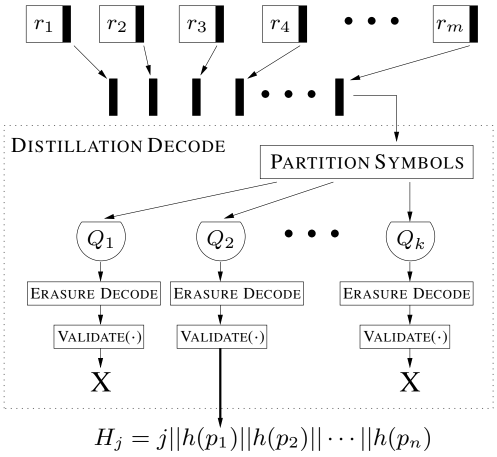
4.4. Practical considerations

Earlier in Section 3, we presented DISTILLATION DECODE as a batch algorithm, where we first collect symbols and then apply the decoding algorithm to all the symbols.

[3] For SAIDA, digital fingerprints are overkill. ECC is typically more expensive than erasure codes, but the additional cost is only noticeable when the input is large. Digital fingerprints use ECC over $L = h(s_1)||h(s_2)||\ldots||h(s_n)$ where the s_i are the erasure encoded symbols of some data D. $|L|$ is relatively small for modest values of n, and thus efficient for ECC. However, when $|D|$ is roughly equal to $|L|$, as it is in SAIDA (for $n = 128$, $|D| = 1408$ vs. $|L| = 1280$), it is more efficient to simply use ECC directly on the input D.



(a) **PRABS sender.** For block j, the PRABS sender executes DISTILLATION ENCODE on the authentication string $H_j = j||p_1||p_2||\ldots||p_n$ for the block and augments each packet with a distillation code symbol.



(b) **PRABS receiver.** The PRABS receiver extracts the distillation code symbols from the received packets and executes DISTILLATION DECODE. The decoder validates the resulting reconstructions, and if sufficiently many valid packets were received, outputs the valid authentication string H_j for the block. Finally, the receiver uses H_j to authenticate each of the received packets.

Figure 5. Pollution Resistant Authenticated Block Streams (PRABS)

In practice, and in particular for PRABS, it is more efficient to implement DISTILLATION DECODE as an on-line algorithm, where the partitions are created dynamically as symbols arrive over the network. When the size of a partition reaches $n - t$, it is erasure decoded and validated.

The chief motivation for the on-line algorithm is to avoid the tough decision of when to apply the batch decoding algorithm. Ideally in the batch algorithm, the receiver should not execute DISTILLATION DECODE until it is confident it has finished receiving all the packets in a particular block. The on-line decoding algorithm avoids this problem by treating each partition independently and decoding it only when it becomes sufficiently large. The only remaining problem is when to release the memory allocated for partitions that never reach $n - t$ symbols. We address this problem in Section 4.5.3.

4.5. Security analysis

We analyze the security of PRABS in terms of packet authenticity, replay protection, and resistance to pollution attacks using our threat model introduced in Section 2.5.

4.5.1 Authenticity

Claim 1. *If a receiver in PRABS receives any $n - t$ valid packets from block j, it can verify the authenticity of all packets in block j.*

Proof. We assume RSA signature generation and verification guarantees authenticity of signed messages, and every receiver has obtained an authentic copy of the legitimate sender's public key. We also assume $h(\cdot)$ is collision-resistant hash function. Then by the authenticity property of distillation codes (Section 3.4.1 and Theorem 3 in Appendix A), every output of DISTILLATION DECODE is guaranteed to be authentic.

Therefore, if a receiver receives any $n - t$ valid packets from block j, then the modified version of DISTILLATION DECODE in Section 4.3 will output the authenticate hash string $H_j = j||h(p_1^j)||h(p_2^j)||\ldots||h(p_n^j)$ of block j. Given any candidate packet $\bar{p}_i^j$ from block j, a receiver can verify its authenticity by checking whether $h(\bar{p}_i^j)$ is equal to the i-th hash entry in the hash string H_j. An adversary able to cause a receiver to accept a forged packet $\bar{p}_i^j \neq p_i^j$ implies she is able to find a collision on $h(\cdot)$ at p_i^j. □

4.5.2 Message reordering and replay

Claim 2. *No valid packet in PRABS will be accepted by a receiver more than once, and for all authenticated received packets, a receiver can reconstruct the order in which they were sent.*

Proof. Let j designate a packet's block and i designate its position in that block. Since there is a one-to-one mapping from (j, i) to the valid packets, after H_j is reconstructed and p_i^j is authenticated and accepted, all future copies of p_i^j can be rejected.

(j, i) also determines a packet's order in the stream. Although an adversary can reorder the delivery of packets, after a receiver authenticates p_i^j, it can determine the proper position of p_i^j in the stream. □

4.5.3 DoS-resistance

Computational DoS-resistance For computational DoS-resistance, we consider the three expensive operations in PRABS: hash function applications, erasure decodings, and signature verifications.

Claim 3. *Let b be the bandwidth of PRABS in blocks per second. Then the most computation an adversary can cause for receivers with a pollution attack with attack factor f is $b \cdot (f+1) \cdot (n \cdot (\log(n)+1)+n)$ hash function applications and $b \cdot (\lfloor \frac{f \cdot n}{n-t} \rfloor + 1)$ erasure decodings and signature verifications.*

Proof. The denial-of-service resistance of PRABS relies on the DoS-resistance properties of distillation codes. For each packet a PRABS receiver receives, it extracts one distillation code symbol and uses it as input to DISTILLATION DECODE. Then by Theorem 2, the adversary can cause the receiver to execute $b \cdot (f+1) \cdot n \cdot (\log(n)+1)$ hash function applications and $b \cdot (\lfloor \frac{f \cdot n}{n-t} \rfloor + 1)$ erasure decodings and signature verifications.

In addition, a receiver must check the authenticity of each received packet p_i^j. This requires checking whether $h(p_i^j)$ is equal to the i-th hash entry in H_j, resulting in at most $b \cdot (f+1) \cdot n$ hash function applications.

Thus, with a legitimate traffic rate of b blocks per second and attack factor f, the most computation an adversary can cause for receivers with a pollution attack is $b \cdot (f+1) \cdot (n \cdot (\log(n)+1)+n)$ hash function applications and $b \cdot (\lfloor \frac{f \cdot n}{n-t} \rfloor + 1)$ erasure decodings and signature verifications. □

For example, consider the scenario of a 1Mb per second stream with $b = 1$, $n = 128$, and 1024 byte packets. With $t = 64$, hash outputs of 80 bits, and 1024 bit RSA signatures, $|(H_j || G_{H_j})| \approx 1408$ bytes, and the resulting erasure code symbols are approximately 22 bytes. For $f = 10$, this requires receivers to execute to at most 11,264 22-byte SHA1 hashes per second, 1408 1024-byte SHA1 hashes per second, and 21 erasure decodings and signature verifications per second. This is relatively insignificant: with these parameters, a 2.4GHz Pentium 4 machine running Linux can compute on average 540,000 22-byte SHA1 hashes per second, 70,000 1024-byte SHA1 hashes per second, 1700 RSA-1024 signature verifications per second, and 3700 (128,64) Reed-Solomon decoding operations per second. We verify this analysis experimentally in Section 5 with an implementation of PRABS.

PRABS is resistant to signature flooding attacks because a signature is distributed among the all packets in the block. To cause a single additional verification operation, an adversary must inject at least $n - t$ packets. In contrast to hash graphs and the Wong-Lam scheme, adversaries can cause an additional verification operation by injecting a single packet.

State-holding DoS-resistance In addition to computational DoS attacks, adversaries can launch state-holding DoS attacks against receivers, attempting to exhaust memory resources. For example, an adversary could accumulate and inject sets of less than $n - t$ invalid packets (symbols) for block sequence numbers far into the future. A naive PRABS receiver will allocate space for these packets and symbols and wait to receive sufficiently many symbols to reconstruct the authentication information. For invalid block sequence numbers much greater than the current valid block sequence number, this attack causes receivers to allocate large amounts of memory held until the valid sequence numbers catch up to the invalid sequence numbers.

One solution to this attack is to limit the amount of memory allocated to PRABS receivers and enforce some reclamation policy on packet (symbol) buffers. However, choosing a reclamation policy can be tricky. We need to be careful legitimate packets awaiting authentication are not freed prematurely by some clever injection of attack traffic.

To prove PRABS is resistant to these state-holding attacks on memory resources, we show an upper bound on the memory requirements for PRABS receivers to achieve the same authentication rate of valid packets under attack as when there is no attack. In our proof, we assume an upper bound d on the maximum end-to-end latency delay imposable by an adversary, and the same upper bound on non-malicious delays normally occurring within the network. We assume the sending rate of the stream is r, and the attacker can inject traffic at a rate up to $f \cdot r$.

Claim 4. *For a rate r stream sending n packets each of size m, attack factor f, and maximum packet delay of d seconds, if a PRABS receiver allocates at least $r \cdot (n \cdot m/r + d) \cdot (f+1)$ bytes of memory, it will not discard any packet that would have been authenticated had there been an infinite amount of memory available.*

Proof. Suppose the PRABS receiver manages its $r \cdot (n \cdot m/r + d) \cdot (f+1)$ byte packet cache with a FIFO replacement policy. We will show that no packet that would have been accepted had there been an infinite sized cache will be discarded. This property will allow us to conclude that a bounded cache does not affect whether a packet is accepted or not.

Suppose that there is a legitimate packet p that is about to be evicted from the cache that would have been accepted with an infinitely sized cache. Since the receiver has not authenticated p, p's partition contains fewer than $n - t$ symbols. For it to be accepted at some later time, the PRABS decoder must receive at least one more packet from its block, since all packets from the same block share accumulator values. But, we know that p has resided in the cache for at least $(n \cdot m/r + d)$ seconds. This is because the cache uses a FIFO replacement policy, its total cache size is $r \cdot (n \cdot m/r) \cdot (f+1)$, and traffic arrives at a rate less than $(f+1) \cdot r$. But, the longest transit delay for a packet is d, and the encoder sends all packets from the same block within $n \cdot m/r$ seconds. Thus, the encoder will not receive any other packet from p's group after $n \cdot m/r + d$ seconds. Since p will never be accepted after $n \cdot m/r + d$ seconds, our assumption that p would have been accepted at some later time is false, and it is safe to discard p. □

4.6. Securely using smaller hash digests with UOWHFs

Using a hash function with an 80 bit output to construct the Merkle hash tree in our distillation code results in $10 \cdot \log(n)$ bytes of overhead per symbol. In this section, we describe an application of UOWHFs that leverages the real-time nature of multicast to reduce this overhead by close to a factor of two without affecting authentication security. This optimization has no significant effect on PRABS's resistance to pollution attacks.

Recall that with the target collision-resistance (TCR) model for UOWHFs (Section 2.6), the sender chooses a particular hash function from a family of TCR hash functions and informs the receivers of the choice before transmission begins. If we assume $h(\cdot)$ is a random oracle, we can construct a TCR hash function by choosing a random salt r and using the first k bits of output from $h_r(x) \triangleq h(r||x||r)$. This is called the *envelope method* [13]. Assuming $h(\cdot)$ and k are agreed in advance, the sender only needs to inform the receivers of the random value r.

To reduce the overhead of distillation code symbols, we would like to to use a hash function with a shorter output, say 40 bits, for constructing the Merkle trees. Unfortunately, shortening the hash output reduces collision-resistance. A 40 bit output provides only about $O(2^{40})$ security, and an adversary is likely to find a collision on $h_r(\cdot)$ during the lifetime of a long lived stream. However, there is no reason to necessarily use the same hash function for every block in PRABS. By revealing a new salt value r at the start of each block's transmission, there is a small bounded amount of time where finding a collision on $h_r(\cdot)$ is useful. After the receivers have successfully received and decoded the valid authentication information for a block, pollution attacks against that particular block become impossible.

To take advantage of this optimization, the encoder and decoder need relatively few changes. The encoder selects a random salt r_j for block j and uses $h_{r_j}(\cdot)$ in the construction of the Merkle hash tree in Step 3 of DISTILLATION ENCODE. To inform receivers of the salt value r_j, the encoder augments each distillation code symbol in block j with the salt value r_j.

To decode, in Step 2a of PARTITION SYMBOLS, the decoder parses each distillation code symbol as (r_j, s', w), and recovers the accumulator value $a = \mathsf{Recover}(s', w)$ using $h_{r_j}(\cdot)$ as the underlying hash function. To prevent adversaries from breaking the accumulator by finding collisions using different salt values, Step 2b of PARTITION SYMBOLS$(\cdot)$ can partition based on both the accumulator value and the advertised salt value.

Security Analysis Since we are shortening only the output of the hash function used in the Merkle tree, and not the output of the hash function used in the authentication string H_j, this change only affects DoS-resistance and not packet authenticity.

For an adversary to launch a successful pollution attack, she must find a collision on $h_{r_j}(\cdot)$ in the Merkle tree over the symbols in block j. If the length of salt values is sufficiently long to prevent long running attacks that iterate over all possible values of r_j, then the adversary must wait until the sender discloses r_j before she tries to find a collision in the Merkle tree.

Since collisions are useless after the receivers have received all the legitimate packets in a block, if we assume adversaries can delay packets by at most d seconds and each block requires c seconds to send, then we must select k such that given r, adversaries have low probability in finding a collision on $h_r(\cdot)$ in $d + c$ seconds. After this time, receivers have presumably received and accepted all the valid packets from the block, and further packets from that block are rejected.

For block size n, the adversary wins if she finds a collision on any of the n symbols in a block. If an adversary hashes $2^{k-\log(n)}$ random values, then she will find a collision on one of the symbols with non-negligible probability. For block size n, we must choose k such that $2^{k-\log(n)}$ is an intractable amount of work for massively parallelized adversaries to complete in $d + c$ seconds. Suppose $n = 128$, one block is transmitted per second, and the maximum adversarial delay is 10 seconds. Given that a 2.4GHz Pentium IV machine can compute roughly 540,000 instances of 22-byte hash function operations per second, choosing $k = 40$ bits requires roughly 1450 machines to complete the necessary work before the receivers have finished receiving the block.

The remaining question is the value of $|r|$. If $|r|$ is small, adversaries can launch long running attacks which iterate over all values of r and hash 2^k random values for each $h_r(\cdot)$. We can bound the effectiveness of long running attacks by using a long per stream salt disseminated to receivers immediately before transmission, but we still must be concerned with long running attacks over the lifetime of the stream. Iterating over all possible values of r requires about $O(2^{|r|+k-\log(n)})$ work to achieve a non-negligible success probability, but a birthday attack can reduce this slightly. By the birthday paradox, if the adversary hashes $2^{k-\log(n)}$ random values for $2^{\frac{|r|}{2}}$ values of r, then the adversary will see a collision after $2^{\frac{|r|}{2}}$ blocks with non-negligible probability. For $n = 128$ and $k = 40$, choosing the $|r| = 64$ bits requires roughly 2^{65} operations for an adversary to be successful after 2^{32} legitimate blocks have been sent.

We stress again this optimization has no effect on packet authenticity. The authentication mechanism is unchanged; hash outputs in the authentication string are 80 bits and signed by a full strength digital signature. If an adversary can delay packets more than d seconds or apply massive computing power to find a collision in the Merkle tree in less than d seconds, she can only cause denial of service and cannot violate authenticity.

Overhead reduction The overhead savings of using a salted hash function construction in the Merkle tree is substantial. Using an unsalted hash function with an 80 bit output results in Merkle hash tree verification sequence lengths of $10 \cdot \log(n)$ bytes. Using the salted hash function with a 40 bit output and a 64 bit salt value yields verification sequences of length $5 \cdot \log(n)$ bytes and an additional 8 bytes for the salt. This provides comparable security for precomputation attacks and saves $(5 \cdot \log(n) - 8)$ bytes per symbol. We show the overhead savings of salting with UOWHFs in Figure 6.

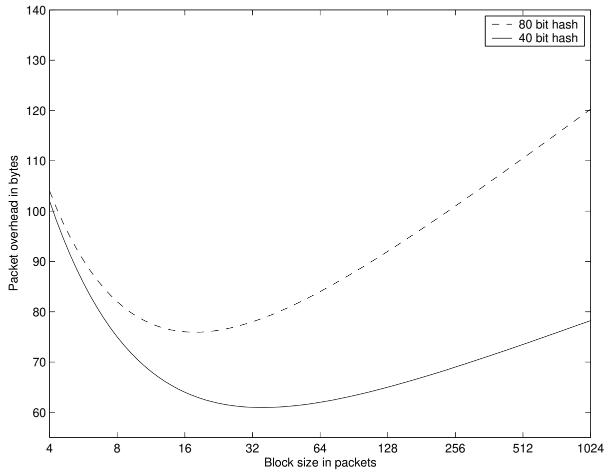


Figure 6. PRABS packet overhead.
This graph shows the effect of block size on packet overhead in PRABS using an $(n, \frac{n}{2})$ distillation code. The total packet overhead is $(10 \cdot n + |G|) \cdot \frac{2}{n} + \log(n) \cdot \frac{k}{8} + |r|$ bytes, with a k bit hash output in the Merkle tree, $|r|$ byte salt, and $|G|$ byte signature. The plot is log scale with $|r| = 8$ and $|G| = 128$. We show the savings of salting and 40 bit hash outputs in the Merkle tree vs. no salting and 80 bit outputs. For smaller block sizes, the erasure encoding dominates the overhead, but becomes small as n grows. For larger n, the witness information for each symbol dominates.

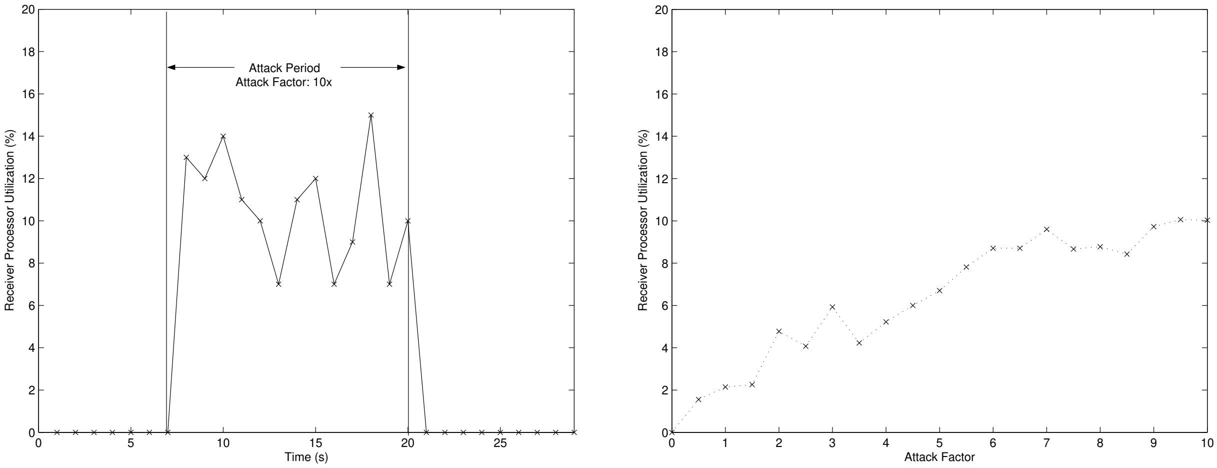
5. Implementation and Measurements

We implemented PRABS and measured its performance. Our goal was to build a multicast authentication protocol that could efficiently operate even when an adversary sends 10 times as much traffic as the original stream.

The protocol and test harness were implemented in 2,300 lines of C++ code. The sending and receiving machines each had a 2.4 GHz Pentium 4 processor and 1GB of RAM. Both machines were running Linux 2.4 kernels and `gcc 2.95`. They are connected by a 100 Mb/s low-latency switched network. We relied on the OpenSSL library[1] for cryptographic functions and a Rizzo's erasure code library[31]. We used RSA-1024 for the TAG$(\cdot)$ and VALIDATE$(\cdot)$ algorithms. We used the SHA1 cryptographic hash function with 40 bit outputs (Section 4.6) for the Merkle tree and 80 bit outputs for the authentication string.

The server sent a stream of data packets at a variety of rates and attack factors. We measured the receiver load when receiving two streams: 1Mb per second and 4Mb per second. Our PRABS stream used 128 packet blocks, where each packet had a 1024 byte data payload. For each stream, we looked at attack factors between 0 and 10. With attack factor 10 against the 4Mb/s stream, the attacker injects 40Mb/s of attack traffic. Recall that the adversary only needs to inject 64 packets to induce a decode operation; by injecting 72 packets, the adversary ensures that 64 packets will arrive at the receiver and can cause the receiver to process 18 times as many malicious blocks as legitimate blocks.

Our tests measure the performance of the client under the worst case. For example, reconstruction data with a systemic erasure code is much slower when using the parity packets (packets 64-127). Our adversary induces a loss



(a) Processor utilization measured once per second to the nearest percent at the receiver while receiving a 4Mb per second stream. During the attack phase, the adversary sends a 10 times as much malicious traffic as legitimate traffic. This corresponds to 18 times as many malicious blocks of traffic. The receiver's processor utilization increases during this phase, but remains below 15%. Outside of the attack phase, authenticating the stream consumes less than 2% of the CPU.

(b) This graph illustrates the relationship between processor utilization and the attack factor. Recall that the attack factor is the bandwidth multiplier of malicious packets that the adversary injects. The processor utilization exhibits a linear relationship with respect to the attack factor. Each data point is an average from 3 separate runs.

Figure 7. Processor utilization while running PRABS.

of 48 legitimate non-parity packets to force the receiver reconstruct its data using mostly the parity packets. Likewise, the adversary chooses to inject parity packets as well to increase the receiver's load using the strong pollution attack detailed in Section 3.4.3.

Our first test measured the processor load on the client as it received a music file. The attacker then began an injection attack on an existing stream with an attack factor of 10. We measured the receiver's process utilization once per second to the nearest percent. The results for the 4Mb/s stream are presented in Figure 7(a). With no attack, the receiver uses only a small fraction of the CPU. Usually it was measured at 0%, and at all times it was under 2%. Under heavy attack, the receiver's CPU load increases to 10%, but always remains below 15%. For a 1Mb/s stream, the receiver's CPU averaged 3.6% during a factor 10 attack. For both streams, the receiver successfully authenticated all received packets.

The second test measured the processor utilization as a function of the attack factor. We display the results for the 4Mb/s stream in Figure 7(b). The graph highlights the linear relationship between the processor utilization and the attack factor, confirming our analysis from Section 3.4.3.

6 Related Work

TESLA is a broadcast authentication scheme with many attractive guarantees: authenticity, low overhead, robustness to loss, and DoS resistance [25, 27]. However, TESLA requires time synchronization between the sender and the receiver. For each block, the sender picks the next key in a one-way key chain and appends a message authentication code [5] to each packet in that block. The sender later publishes the key. Receivers validate the key using the one-way chain and only accept packets authenticated with that key that arrive before it was disclosed.

Previous work has addressed erasure symbol corruption in the context of distributed storage. Krawczyk proposed distributed fingerprints, an application of error correction codes (ECC) in conjunction with erasure codes to detect altered symbols [14]. However, as discussed in Section 3.1, ECC is also vulnerable to pollution attacks. Distributed fingerprints work well when invalid symbols are guaranteed to replace the valid symbols, but not when there is pollution.

Weatherspoon et al. proposed a scheme similar to ours for detecting corrupted symbols in the distributed archival system of Oceanstore [34]. However, similar to distributed fingerprints, they do not consider pollution attacks where additional invalid symbols are injected into the decoding process.

7. Conclusion

Distillation codes enable systems to store or transmit information that is robust against packet loss, pollution attacks, and modification of transmitted packets. We demonstrated the potential of distillation codes by introducing PRABS, an new DoS-resistant multicast authentication protocol. PRABS is secure against a wide variety of pollution attacks without requiring significant overhead, either in the space required to represent symbols or in the computational effort required to encode and decode messages. Distillation codes are fast, general, and secure, but more important, they are designed to face realistic, hostile threat models.

We hypothesize that distillation codes are applicable in a variety of contexts. Consider the example of a distributed Internet-wide file service. A user wishes to store a file across multiple untrusted repositories and hopes to recover his file at a later date. Assume that the user can not trust any single machine to permanently store the entire file or metadata about the file. One approach is to divide the original file into shares, and then sign each of the shares as they are distributed to different machines. By producing shares that store redundant information (along the lines of secret sharing, the information dispersal algorithm, or erasure codes), we can check to ensure that none of the shares have been tampered with. However, if share reconstruction takes place on a heavily loaded file server that is simultaneously reconstructing many different files, this may yield an unreasonable load. In contrast, we can imagine a system that signs the original file, and then uses distillation codes to rapidly reconstruct the file. While further investigation is required, we hypothesize that distillation codes may yield good performance in this situation. Similarly, other cases where data must be segregated and then reconstructed may also be fertile ground for exploring the potential of distillation codes.

Acknowledgements

This work was supported in part by research grants and contracts from the National Science Foundation, the Defense Advanced Research Projects Agency, the US Postal Service, the Army Research Office, the Center for Computer and Communications Security at Carnegie Mellon, and gifts from Bosch, Intel, KDD, and Honda. Michael Luby helped motivate this work and guided us in its early stages. David Wagner provided extensive guidance on our proofs of security and Rob Johnson helped with our security analysis. We thank the anonymous reviewers and Diana Smetters for helpful comments which significantly improved the quality of our paper. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the US Government or any of its agencies.

References

[1] OpenSSL. `http://www.openssl.org/`.

[2] D. Adkins, K. Lakshminarayanan, A. Perrig, and I. Stoica. Taming IP packet flooding attacks. In *Proceedings of Workshop on Hot Topics in Networks (HotNets-II)*, Nov. 2003.

[3] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. In *Proceedings of Workshop on Hot Topics in Networks (HotNets-II)*, Nov. 2003.

[4] N. Baric and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology – EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 480–494, 1997.

[5] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15, 1996.

[6] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology – ASIACRYPT '00*, volume 1997 of *Lecture Notes in Computer Science*, pages 531–545, 2000.

[7] M. Bellare and P. Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In *Advances in Cryptology – CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 470–484, 1997.

[8] J. Benaloh and M. de Mare. One way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology – EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 274–285, 1993.

[9] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Advances in Cryptology – CRYPTO '02*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76, 2002.

[10] V. Gligor. Guaranteeing access in spite of service-flooding attacks. In *Proceedings of the Security Protocols Workshop*, Apr. 2003.

[11] P. Golle and N. Modadugu. Authenticating streamed data in the presence of random packet loss. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2001)*, pages 13–22. Internet Society, Feb. 2001.

[12] M. Goodrich, R. Tamassia, and J. Hasic. An efficient dynamic and distributed cryptographic accumulator. In *Proceedings of Information Security Conference (ISC 2002)*, volume 2433 of *Lecture Notes in Computer Science*, pages 372–388, 2002.

[13] B. Kaliski and M. Robshaw. Message authentication with MD5. *RSA Cryptobytes*, 1(1):5–8, Spring 1995.

[14] H. Krawczyk. Distributed fingerprints and secure informational dispersal. In *Proceedings of 13th ACM Symposium on Principles of Distributed Computing*, pages 207–218. ACM, 1993.

[15] M. Luby. LT codes. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS '02)*, pages 271–282, 2002.

[16] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *Proceedings of 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, pages 150–159, May 1997.

[17] R. Merkle. Protocols for public key cryptosystems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 122–134, Apr. 1980.

[18] S. Miner and J. Staddon. Graph-based authentication of digital streams. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 232–246, May 2001.

[19] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC '89)*, pages 33–43, May 1989.

[20] K. Nyberg. Fast accumulated hashing. In *Fast Software Encryption – Third International Workshop*, volume 1039 of *Lecture Notes in Computer Science*, pages 83–87, 1996.

[21] A. Pannetrat and R. Molva. Efficient multicast packet authentication. In *Proceedings of the Symposium on Network and Distributed System Security Symposium (NDSS 2003)*. Internet Society, Feb. 2003.

[22] J. M. Park, E. Chong, and H. J. Siegel. Efficient multicast packet authentication using erasure codes. *ACM Transactions on Information and System Security (TISSEC)*, 6(2):258–285, May 2003.

[23] J. M. Park, E. K. Chong, and H. J. Siegel. Efficient multicast packet authentication using signature amortization. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 227–240, May 2002.

[24] A. Perrig. The BiBa one-time signature and broadcast authentication protocol. In *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS-8)*, pages 28–37, Philadelphia PA, USA, Nov. 2001.

[25] A. Perrig, R. Canetti, D. Song, and J. D. Tygar. Efficient and secure source authentication for multicast. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2001)*, pages 35–46. Internet Society, Feb. 2001.

[26] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. Efficient authentication and signature of multicast streams over lossy channels. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 56–73, May 2000.

[27] A. Perrig and J. D. Tygar. *Secure Broadcast Communication in Wired and Wireless Networks*. Kluwer Academic Publishers, 2002.

[28] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[29] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[30] L. Reyzin and N. Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In *Seventh Australasian Conference on Information Security and Privacy (ACISP 2002)*, July 2002.

[31] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2):24–36, Apr. 1997.

[32] T. Sander. Efficient accumulators without trapdoor extended abstracts. In *Information and Communication Security, Second International Conference – ICICS '99*, volume 1726 of *Lecture Notes in Computer Science*, pages 252–262, 1999.

[33] D. Song, D. Zuckerman, and J. D. Tygar. Expander graphs for digital stream authentication and robust overlay networks. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 258–270, May 2002.

[34] H. Weatherspoon, C. Wells, P. R. Eaton, B. Y. Zhao, and J. D. Kubiatowicz. Silverback: A global-scale archival system. Technical Report UCB//CSD-01-1139, University of California at Berkeley, 2000.

[35] C. Wong and S. Lam. Digital signatures for flows and multicasts. In *Proceedings on the 6th International Conference on Network Protocols (ICNP '98)*, pages 198–209. IEEE, October 1998.

A. Security Analysis of Distillation Codes: Authenticity

We will show that if the $\textsc{Tag}(\cdot)$ and $\textsc{Validate}(\cdot)$ algorithms satisfy integrity of plaintext, then a distillation code using these algorithms also satisfies integrity of plaintext. We will adapt Bellare and Namprempre's notion of integrity of plaintext (INT-PTXT) [6] to the public key setting. Note that their definition was in the context of encryption and decryption algorithms, while we frame ours in terms of encoding and decoding algorithms.

Definitions & Notation We define a *public key encoding scheme* $\mathcal{PKE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ to consist of three algorithms. The randomized key generation algorithm $\mathcal{K}$ takes a natural number $k \in \mathbb{N}$ as its security parameter and outputs a public key K_{pub} and private key K_{priv}: $(K_{\text{pub}}, K_{\text{priv}}) \overset{R}{\longleftarrow} \mathcal{K}(k)$. The encoding algorithm $\mathcal{E}$ uses the private key K_{priv} to encode a message M into a codetext C, possibly using a randomization source: $C \overset{R}{\longleftarrow} \mathcal{E}_{K_{\text{priv}}}(M)$. The decoding algorithm $\mathcal{D}$ uses the public key K_{pub} to authenticate the codetext C. It returns a plaintext if it can authenticate the codetext, or $\perp$ if it cannot validate the codetext: $P^{\perp} \leftarrow \mathcal{D}_{K_{\text{pub}}}(C)$ where $P^{\perp} \in \{\perp\} \cup \{0,1\}^*$. For all encoding schemes, we require $M = \mathcal{D}_{K_{\text{pub}}}(\mathcal{E}_{K_{\text{priv}}}(M))$

We now provide an authenticity definition for a public key encoding scheme $\mathcal{PKE}(\mathcal{K}, \mathcal{E}, \mathcal{D})$. A verification algorithm $\mathcal{D}^*_{K_{\text{pub}}}$ takes a codetext and returns a boolean value indicating whether the decoding was successful:

$\textsc{Algorithm}\ \mathcal{D}^*_{K_{\text{pub}}} : C \in \{0,1\}^* \mapsto \{0,1\}$
  if $D_{K_{\text{pub}}}(C) \neq \perp$ **return** 1
  return 0

We allow an adversary access to K_{pub}, an encoding oracle, as well as a verification oracle in order to try to break the encoding scheme. The adversary is able to violate the integrity of the encoding scheme if it can produce a codetext whose plaintext was never passed to the encoding algorithm $\mathcal{E}_{K_{\text{pub}}}(\cdot)$. If it is computationally difficult for an adversary to produce such a plaintext, the encoding scheme is said to offer *integrity of plaintext*, abbreviated INT-PTXT.

Definition 2. *Integrity of a public key encoding scheme*

Let $\mathcal{PKE}(\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a public key encoding scheme. Let $k \in \mathbb{N}$ and $\mathcal{A}$ be an adversary that has access to an encoding oracle $\mathcal{E}(\cdot)$ and a verification oracle $\mathcal{D}^*(\cdot)$. Then, consider the following experiment:

EXPERIMENT $\mathbf{Exp}_{\mathcal{PKE},\mathcal{A}} : k \in \mathbb{N} \mapsto \{0,1\}$
$(K_{\text{pub}}, K_{\text{priv}}) \xleftarrow{R} \mathcal{K}(k)$
if $\mathcal{A}_{\mathcal{E}_{K_{\text{priv}}}(\cdot),\mathcal{D}^*_{K_{\text{pub}}}(\cdot)}(k, K_{\text{pub}})$ makes a query
to $\mathcal{D}^*_{K_{\text{pub}}}(\cdot)$ such that:
$\mathcal{D}^*_{K_{\text{pub}}}(C) = 1$ **and** $\mathcal{D}_{K_{\text{pub}}}(C)$ was
never a query to $\mathcal{E}_{K_{\text{priv}}}(\cdot)$
then return 1
else return 0

The *advantage* of the adversary is the probability that the adversary can produce a query to the decoder that returns success for which the corresponding plaincode was never passed to the encoder. Specifically:

$$\mathbf{Adv}_{\mathcal{PKE},\mathcal{A}}(k) \triangleq \Pr\left[\mathbf{Exp}_{\mathcal{PKE},\mathcal{A}}(k) = 1\right]$$

We define the *advantage function of the scheme* in terms of τ, the running time of the adversary, q_e, the number of queries the adversary makes to the encoding algorithm $\mathcal{E}$ with total length μ_e, and q_d, the number of queries the adversary makes to the verification algorithm $\mathcal{D}^*$ with total length μ_d as:

$$\mathbf{Adv}_{\mathcal{PKE}}(k, \tau, q_e, q_d, \mu_e, \mu_d) \triangleq \max_{\mathcal{A}}\{\mathbf{Adv}_{\mathcal{PKE},\mathcal{A}}(k)\}$$

The scheme $\mathcal{PKE}$ satisfies INT-PTXT if $\mathbf{Adv}_{\mathcal{PKE},\mathcal{A}}(k)$ is negligible for any adversary $\mathcal{A}$ with time-complexity polynomial in k.

Distillation Codes We now briefly present distillation codes in the public key encoding framework. Distillation codes use an underlying public key encoding scheme $\mathcal{PKE}^V = (\mathcal{K}^V, \mathcal{E}^V, \mathcal{D}^V)$ that provides integrity protection. To refer to an instantiation of a particular distillation code, we write $\mathcal{PKE}^{DC} = (\mathcal{K}^{DC}, \mathcal{E}^{DC}, \mathcal{D}^{DC})$. This distillation code uses the underlying code $\mathcal{PKE}^V$ to provide integrity protection. The key generation algorithm $\mathcal{K}^{DC}$ returns a public-private keypair $(K^{DC}_{\text{pub}}, K^{DC}_{\text{priv}})$. Since the distillation code can decode despite symbol loss, the decoding algorithm takes a string composed of either symbols or $\perp$ to represent a missing symbol: $s_1^\perp || s_2^\perp || \ldots || s_n^\perp$ where $s_i^\perp \in \{\perp, s_i\}$.

We set r to be the input message size; (n, t) represent the erasure coding parameters: n the number of encoded symbols per message, and t the maximum number of symbols that can be lost for successful reconstruction; m to be the size of the encoded erasure symbols; and f the maximum attack factor, as defined in Section 2.5. We consider all of these as fixed parameters for a particular instantiation of $\mathcal{PKE}^V$.

We abstract distillation code key generation, encoding, and decoding algorithms from Figures 2 and 3. We define the algorithm $DE(\cdot)$ to be steps 2-3 of DISTILLATION ENCODE that erasure encodes the authenticated data and augments them with the accumulator values. We define the algorithm $DC(\cdot)$ to be steps 1-3 of DISTILLATION DECODE from Figure 3 that returns a set of candidate reconstructions.

ALGORITHM $\mathcal{K}^{DC} : k \in \mathbb{N} \mapsto (K^{DC}_{\text{pub}}, K^{DC}_{\text{priv}})$
$K \leftarrow \mathcal{K}^V(k)$
return K
ALGORITHM $\mathcal{E}^{DC}_{K^{DC}_{\text{priv}}} : M \in \{0,1\}^r \mapsto \{s\}^n$
$C \leftarrow DE(\mathcal{E}^V_{K^V_{\text{priv}}}(M))$
return C
ALGORITHM $\mathcal{D}^{DC}_{K^{DC}_{\text{pub}}} : C \in \{\perp\} \cup \{0,1\}^m \mapsto \{0,1\}^r \cup \{\perp\}$
$\mathcal{R} \leftarrow DC(C)$
for each $R \in \mathcal{R}$
if $\mathcal{D}^V_{K^V_{\text{priv}}}(R) \neq \perp$ **return** $\mathcal{D}^V_{K^V_{\text{priv}}}(R)$
return $\perp$

Theorem 3. *If* $\mathcal{PKE}^V = (\mathcal{K}^V, \mathcal{E}^V, \mathcal{D}^V)$ *is* INT-PTXT *secure, then the distillation code* $\mathcal{PKE}^{DC} = (\mathcal{K}^{DC}, \mathcal{E}^{DC}, \mathcal{D}^{DC})$ *is also* INT-PTXT *secure.*

Proof. Assume that there exists an adversary $\mathcal{A}_{\mathcal{E}^{DC}(\cdot),\mathcal{D}^{*DC}(\cdot)}$ that can violate the INT-PTXT property of a distillation code $\mathcal{PKE}^{\mathcal{DC}}$. Then, we will provide a construction for an adversary $\mathcal{B}_{\mathcal{E}^V(\cdot),\mathcal{D}^{*V}(\cdot)}$ that can break any INT-PTXT secure encoding scheme $\mathcal{PKE}^V$. In other words, we will create an adversary $\mathcal{B}$ that makes a query C^V to $\mathcal{D}^{*V}(\cdot)$ such that $\mathcal{D}^V(C^V) \neq \perp$ and $\mathcal{D}^V(C^V)$ was never a query to $\mathcal{E}^V(\cdot)$. We will prove that the advantage for adversary $\mathcal{B}$ will be at least as large as that held by adversary $\mathcal{A}$.

$$\mathbf{Adv}_{\mathcal{PKE}^{DC},\mathcal{A}}(k) \leq \mathbf{Adv}_{\mathcal{PKE}^V,\mathcal{B}}(k) \qquad (1)$$

Furthermore, if $\mathcal{A}$ runs in time t using q_e encoding queries of total length μ_e and q_d verification queries of total length μ_d, then $\mathcal{B}$ will run in the same query size parameters μ_e and μ_d making q_e encoding and $\leq \left(\lfloor \frac{fn}{n-t} \rfloor + 1\right) q_d$ verification queries.

The adversary $\mathcal{B}$ will use the adversary $\mathcal{A}$ to break $\mathcal{PKE}^V$. It will emulate the distillation code encoding and verification process fully so that $\mathcal{A}$ will believe that it is interacting with a true distillation encoder and verifier. Thus, $\mathcal{B}$ will take in the security parameter k and a public key and will output a codetext C. Specifically:

ADVERSARY $\mathcal{B}_{\mathcal{E}^V(\cdot),\mathcal{D}^{*V}} : K_{\text{pub}} \times k \in \mathbb{N} \mapsto \{0,1\}$
for $i = 1 \dots (q_d + q_e)$ **do**
when $\mathcal{A}$ makes a query M to its encoding oracle $\mathcal{E}^{DC}(\cdot)$,
do $\mathcal{A} \Leftarrow DE(\mathcal{E}^V(M))$
when $\mathcal{A}$ makes a query T to its verification oracle $\mathcal{D}^{*DC}(\cdot)$,
do $\{R_1, \dots, R_l\} \leftarrow DC(T)$
for $j = 1, \dots, l$
if $\mathcal{D}^{*V}_{K^V_{\text{pub}}}(R_j) = 1$
$\mathcal{A} \Leftarrow 1$; **return**
$\mathcal{A} \Leftarrow 0$

Suppose, in the course of its run, adversary $\mathcal{A}$ has advantage $\alpha = \mathbf{Adv}_{\mathcal{PKE}^{DC},\mathcal{A}}$. In other words, it succeeds in breaking the distillation code in an α fraction of its executions. Consider such an execution. In this execution, let C denote the first query that $\mathcal{A}$ makes to the verification oracle $\mathcal{D}^{*DC}(C)$ for which it has never made the query $\mathcal{E}^{DC}(\mathcal{D}^{DC}(C))$ with $\mathcal{D}^{DC}(C) \neq \perp$. By construction of $\mathcal{D}^{\mathcal{DC}}$, this means that there exists some R_i and for which $\mathcal{D}^V(R_i) \neq \perp$. Now, to show this violates the INT-PTXT property of $\mathcal{PKE}^V$, we need to verify that $\mathcal{D}^V(R_i)$ was never a query to to $\mathcal{E}^V(\cdot)$. We know that $\mathcal{D}^V(R_i) = \mathcal{D}^{DC}(C)$ and $\mathcal{D}^{DC}(C)$ was never a query to $\mathcal{E}^{DC}(\cdot)$. This means that $\mathcal{D}^V(R_i)$ was never a query to $\mathcal{E}^V(\cdot)$. Thus, the advantage that adversary $\mathcal{B}$ has in breaking $\mathcal{PKE}^V$ is at least as large as $\mathcal{A}$ has in breaking $\mathcal{PKE}$. Thus we have a contradiction since we assumed that $\mathcal{PKE}^V$ is INT-PTXT, implying that there can be no adversary that breaks $\mathcal{PKE}$ with non-negligible probability.

We note that $\mathcal{B}$ will make more queries to the decoding oracle than $\mathcal{A}$. Since a given codetext can produce many candidate reconstructions, each of which needs to be validated, $\mathcal{B}$ will make more queries to its validation oracle. In fact, as argued in Section 3.4.3, there will be at most $\lfloor \frac{fn}{n-t} \rfloor + 1$ candidate reconstructions. Thus, if $\mathcal{A}$ makes q_d validation oracle calls, $\mathcal{B}$ will make $\leq \left(\lfloor \frac{fn}{n-t} \rfloor + 1 \right) q_d$ validation oracle calls. $\square$