Network Coding for Anonymous Broadcast

by

Ivan A. Sergeev

S.B., Electrical Engineering and Computer Science, M.I.T., 2012

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2013

Copyright 2013 Ivan A. Sergeev. All rights reserved.

The author hereby grants to MIT permission to reproduce and to
distribute publicly paper and electronic copies of this thesis document
in whole or in part in any medium now known or hereafter created.

Author

Department of Electrical Engineering and Computer Science

May 24, 2013

Certified by

Muriel Médard

Professor of Electrical Engineering, Thesis Supervisor

Accepted by

Prof. Dennis M. Freeman

Chairman, Masters of Engineering Thesis Committee
Network Coding for Anonymous Broadcast

by

Ivan A. Sergeev

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2013, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis explores the use of network coding for anonymous broadcast. Network coding, the technique of transmitting or storing mixtures of messages rather than individual messages, can provide anonymity with its mixing nature, efficiently disseminate content in multicast and broadcast networks, and resiliently deliver messages despite packet erasure and constrained network resources. While broadcast mediums guarantee receiver anonymity, they are thought to be difficult to emulate efficiently over unicast networks. This thesis introduces NCGAB, a decentralized peer-to-peer overlay network based on network coded gossip that provides a resilient, anonymous broadcast medium. Unlike most anonymous communication systems, NCGAB requires no cryptosystem, no infrastructure of trust, and no special nodes to operate. This thesis also introduces Melting Pad, an algebraic coding scheme with properties of information theoretic security and efficient decodability, designed to protect messages for wide dissemination and for hosting with diminished liability.

Thesis Supervisor: Muriel Médard
Title: Professor of Electrical Engineering
Acknowledgments

I thank my advisor, Prof. Muriel Médard, for giving me the space to run off with my little ideas and network coding side projects, and for her trust in me that something would come of them. Through the guiding hand of her experience and her contagious curiosity, Muriel has revealed to me the joy of research. This work would not have been possible without her flexibility, insight, and support. I also thank my co-advisor, Prof. João Barros of Universidade do Porto, for his direction, advice, and constant flow of ideas on these projects. I thank my mentor and friend, Kerim Fouli, for his uncanny capacity to be there at the right time for me.
Contents

1 Introduction ........................................... 15

2 Network Coding ........................................... 17

3 NCGAB .................................................. 21
   3.1 Introduction ......................................... 21
   3.2 Related Work ......................................... 23
   3.3 System Overview ...................................... 25
      3.3.1 Anonymous ..................................... 26
      3.3.2 Decentralized .................................. 26
      3.3.3 Highly Available ................................. 26
      3.3.4 Resilient ........................................ 27
   3.4 Network Coding Tools ................................. 27
      3.4.1 Network Coded Gossip Protocols ................ 27
      3.4.2 Homomorphic Hashes ............................. 28
   3.5 System Architecture .................................. 29
      3.5.1 Gossip and Decoding Strategy .................. 30
      3.5.2 Ephemeral Message Window ....................... 31
      3.5.3 Amended Gossip Strategy ......................... 32
      3.5.4 Message and Packet Structure ................. 33
      3.5.5 Message Contribution by Application ............ 35
      3.5.6 System Parameters ............................... 35
      3.5.7 System Pseudo-code ............................... 36
### 3.6 Analysis of Anonymity
- **3.6.1 Traffic Analysis by an Omniscient Adversary**
- **3.6.2 Traffic Analysis by a Limited Adversary**

### 3.7 Analysis of Attacks
- **3.7.1 Threat Model, Attack Vectors, and Goal of an Adversary**
- **3.7.2 Inactive Peer**
- **3.7.3 Pollution with Invalid Gossip**
- **3.7.4 Pollution with Old or Underdetermined Gossip**
- **3.7.5 Pollution with Decodable Gossip**
- **3.7.6 Pollution with Colluded Decodable Gossip**

### 3.8 Simulations and Results
- **3.8.1 ncgabsim Simulator**
- **3.8.2 Availability and Delay**
- **3.8.3 Cooperative Peer Simulations**
- **3.8.4 Simulation of Uniform versus Weighted Item Selection**
- **3.8.5 Simulation of an Inactive Peer**
- **3.8.6 Simulation of Pollution with Underdetermined Gossip**
- **3.8.7 Simulation of Pollution with Decodable Gossip**

### 4 Melting Pad

#### 4.1 Introduction

#### 4.2 Interference Alignment

#### 4.3 Melting Pad Coding and Decoding
- **4.3.1 Example**
- **4.3.2 Explanation**
- **4.3.3 Security**

#### 4.4 Confidentiality
- **4.4.1 Maximal Security**
- **4.4.2 Achievability of Maximal Security**

#### 4.5 Applications
4.5.1 Subscriber Content Delivery .......................... 64
4.5.2 Secure Storage on Untrusted Servers ................. 64
4.6 Extensions ................................................. 65
  4.6.1 Grow ..................................................... 65
  4.6.2 Shrink ................................................... 66
  4.6.3 Onion ..................................................... 66

5 Conclusion ...................................................... 69
  5.1 NCGAB Conclusion and Future Work ....................... 69
  5.2 Melting Pad Conclusion and Future Work .................. 70

A ncgabsim Simulator Code .................................... 73
  A.1 ncgabsim.py ............................................... 73
  A.2 ff.py ....................................................... 88
  A.3 stats_process.py ............................................ 93
List of Figures

3-1 NCGAB Overlay Network .................................................. 25
3-2 NCGAB Ephemeral Message Window Structures .................. 29
3-3 Example Message Structure ............................................. 33
3-4 Example Gossip Packet Structure .................................... 34
3-5 NCGAB System Pseudocode ............................................ 37
3-6 NCGAB Overlay Network Graph ....................................... 38
3-7 Cooperative Peer Simulation .......................................... 49
3-8 Uniform versus Weighted Item Random Selection Simulation ... 50
3-9 Inactive Peer Attackers Simulation ................................... 51
3-10 Pollution with Underdetermined Gossip Simulation ............ 52
3-11 Pollution with Decodable Gossip Simulation ...................... 53
4-1 Subscriber Content Delivery Application ............................ 63
4-2 Secure Storage on Untrusted Servers ................................. 64
List of Tables

3.1 NCGAB System Parameters ........................................ 35
3.2 Available ncgabsim Simulation Parameters .......................... 47
3.3 Cooperative Peer Simulation Parameters ............................. 49
3.4 Uniform versus Weighted Item Random Selection Simulation Parameters 50
3.5 Inactive Peer Attackers Simulation Parameters ..................... 51
3.6 Pollution with Underdetermined Gossip Simulation Parameters ... 51
3.7 Pollution with Decodable Gossip Simulation Parameters ............ 52
Chapter 1

Introduction

Anonymity is becoming increasingly scarce on the Internet, but remains in high demand to netizens evading censorship, to individuals who value privacy or freedom of expression, and to operatives who carry out sensitive communications at the risk of their well-being. Internet Service Providers have been known to monitor traffic and blacklist hosts. The trend of cloud computing and its accompanying targeted advertising market have rendered the default Internet experience to be highly stateful, with website visits regularly leaving a trail of footprints in the hands of untrusted and unknown third parties. The property of anonymity in Internet communication has become exclusively “opt-in,” through the use of special systems like anonymizing proxies and anonymizing transport layers.

However, there are few distinct systems for anonymous communication in active use today, reflecting the challenge of their successful implementation and deployment. Many are encumbered by implicit infrastructures of trust, by complicated protocols, by correct implementation of cryptography, and by the requirement for pre-existing coordinating and “Samaritan” nodes to operate.

A simpler and completely decentralized approach to anonymity is broadcast based. Broadcast mediums intrinsically confer receivers with anonymity, but are difficult to scale over an effectively unicast network, like the Internet. Past broadcast based anonymity systems have required complicated network architectures to overcome this scaling problem, and still pose a challenge to implement with their dependency on
cryptography and complexity.

Network coding, the technique of transmitting or storing mixtures of messages rather than individual messages, is well suited to multicast and broadcast applications. It was shown that the technique is required in achieving the multicast capacity of a network [5], which subsequently spawned abundant research that has yielded a myriad of results and applications, such as correcting for network failures [27], correcting errors on a packet erasure channel [33], peer-to-peer content distribution [16], mixing messages for confidentiality [39], and more.

In chapter 2, we present network coding and its mechanics.

In chapter 3, we introduce NCGAB, a decentralized peer-to-peer system based on network coded gossip that provides a resilient, anonymous broadcast medium. Unlike most anonymous communication systems, NCGAB requires no cryptosystem, no infrastructure of trust, and no special nodes to operate. The system’s operation can be summarized in a single page of pseudocode and is easy to implement.

In chapter 4, we introduce Melting Pad, an algebraic coding scheme with information theoretic security designed to protect messages for wide dissemination and hosting with diminished liability. Melting Pad supports flexible modes of secrecy, with either distinct secret keys protecting each source block, or one common key protecting all source blocks. Melting Pad is derived from interference alignment concepts to allowing for efficient separate decoding of every protected block.

In chapter 5, we summarize the contributions of this thesis and outline future work.
Chapter 2

Network Coding

Network coding is the technique of transmitting or storing mixtures of packets rather than individual packets.

In linear network coding, multiple packets are combined into algebraic linear combinations, transmitted or stored, and later decoded into their constituent packets by solving a linear system constructed from their linear combinations. Linear network coding allows for efficient distribution of content to one or more receivers through disparate routes, and for successful delivery of content in spite of lost or missing packets [27] [33].

We demonstrate the mechanics of linear network coding below. We select a finite field \( \mathbb{F}_q \) with \( q = 2^8 \) to carry out algebraic operations within. Packets of 5-bytes are treated as \( l = 5 \) length vectors of elements in this finite field.

\[
\begin{align*}
\mathbf{x}_1 &= \begin{pmatrix} 209 & 121 & 6 & 2 & 156 \end{pmatrix}^T \\
\mathbf{x}_2 &= \begin{pmatrix} 154 & 8 & 52 & 248 & 236 \end{pmatrix}^T \\
\mathbf{x}_3 &= \begin{pmatrix} 247 & 24 & 117 & 93 & 62 \end{pmatrix}^T
\end{align*}
\]

We may form linear combinations of the packets. If we choose the scalar coefficients such that the linear combinations are independent, then we can solve for all three source packets from three linear combinations.
\[
L_1 = 115x_1 + 227x_2 + 202x_3 \\
L_2 = 39x_1 + 88x_2 + 236x_3 \\
L_3 = 198x_1 + 98x_2 + 204x_3 \\
L_1 = \begin{pmatrix} 37 & 205 & 254 & 234 & 148 \end{pmatrix}^T \\
L_2 = \begin{pmatrix} 109 & 88 & 15 & 179 & 54 \end{pmatrix}^T \\
L_3 = \begin{pmatrix} 99 & 197 & 57 & 126 & 29 \end{pmatrix}^T
\]

Linear combinations of packets are collected into a decoding matrix and decoded into their constituent packets with Gaussian elimination. Decoding will solve for none, some, or all of the source packets, depending on the rank of the decoding matrix.

\[
\begin{bmatrix} 115 & 227 & 202 \\ 39 & 88 & 236 \\ 198 & 98 & 204 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{pmatrix} 37 & 205 & 254 & 234 & 148 \end{pmatrix}^T \\
\begin{pmatrix} 109 & 88 & 15 & 179 & 54 \end{pmatrix}^T \\
\begin{pmatrix} 99 & 197 & 57 & 126 & 29 \end{pmatrix}^T
\]

\[
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 115 & 227 & 202 \\ 39 & 88 & 236 \\ 198 & 98 & 204 \end{bmatrix}^{-1} \begin{pmatrix} 37 & 205 & 254 & 234 & 148 \end{pmatrix}^T \\
\begin{pmatrix} 109 & 88 & 15 & 179 & 54 \end{pmatrix}^T \\
\begin{pmatrix} 99 & 197 & 57 & 126 & 29 \end{pmatrix}^T
\]

\[
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{pmatrix} 209 & 121 & 6 & 2 & 156 \end{pmatrix}^T \\
\begin{pmatrix} 154 & 8 & 52 & 248 & 236 \end{pmatrix}^T \\
\begin{pmatrix} 247 & 24 & 117 & 93 & 62 \end{pmatrix}^T
\]

A received linear combination that is linearly independent of pre-existing linear combinations in the decoding matrix is called **innovative**, as it increases the rank of the decoding matrix, contributing new information that may be used to further decode the constituent packets.

Random Linear Network Coding is a linear network coding technique that uses
random coefficients when forming linear combinations of packets. This ensures, with high probability and in a decentralized fashion, that newly formed linear combinations of the same packets continue to be innovative to its receivers. [20] shows that this approach achieves the maximum throughput of a multicast network. We utilize random linear network coding in our decentralized anonymity system to create mixtures of packets that are innovative with high probability.
Chapter 3

NCGAB

3.1 Introduction

Many anonymous transport layers achieve anonymity by multiple hops of source rewriting, in which messages pass through a number of relays to introduce sufficient uncertainty about the sender. These systems are often encumbered by complex infrastructures of trust and require Samaritan nodes to operate, making successful deployments of them difficult, large, and highly visible to adversaries. The source rewriting mechanism often requires cryptography, making its correct implementation a challenge. The mechanism is often stateful, so packet flows will take pre-selected rigid paths, posing the problem of traffic analysis.

An alternative approach to anonymous transport is broadcast, in which messages have no explicit destination and are delivered to all participants. Broadcast mediums guarantee receiver anonymity by design. However, the approach seems unreasonable and difficult to implement efficiently over an effectively unicast network architecture, such as the Internet. Past systems, like \( P^5 \) [37] and Herbivore [18], have addressed the inefficiency in scaling a broadcast medium over a unicast network by partitioning nodes into trees (\( \mathcal{P}^5 \)) or rings of cliques (Herbivore). This results in a complex network architecture and these systems still must rely on cryptography.

Network coding, the technique of transmitting or storing mixtures of messages rather than individual messages, is well suited to multicast and broadcast applica-
tions. It was shown that the technique is required in achieving the multicast capacity of a network [5], which subsequently spawned abundant research that has yielded a myriad of results and applications, such as correcting for network failures [27], correcting errors on a packet erasure channel [33], peer-to-peer content distribution [16], mixing messages for confidentiality [39], and more.

We introduce NCGAB, a peer-to-peer overlay network that leverages network coding to create a resilient, anonymous broadcast medium. Network coding's inherent mixing property provides sender anonymity with no cryptography, and enables efficient dissemination of content with a gossip protocol, altogether making a broadcast approach to anonymity viable and scalable. Its inherent redundancy confers the system with resiliency, even in the presence of active attackers squandering network resources and polluting the medium. NCGAB requires no infrastructure of trust and no cryptosystem. It is completely decentralized, free of coordinating, trusted, and Samaritan nodes and thus immediately deployable. Its implementation is simple and summarized in a single page of pseudocode. Comprehensive simulations show it to be highly resilient, providing an available broadcast medium for 10 to 100 peers, even when 30% of the peers are operated by active attackers.

This paper is outlined as follows: The Related Work section reviews related work on anonymity and network coded gossip. The System Overview section presents a high-level description of the NCGAB system and its properties. The Network Coding Tools section introduces the tools of network coding and homomorphic hashing used by NCGAB to achieve its functionality and guarantees. The System Architecture section outlines the design of the system in detail. The Analysis of Anonymity section investigates the anonymity of participants using the system. The Analysis of Attacks section investigates the efficacy of different attacks in compromising the availability of the system. The Simulations and Results section presents simulations of the system and results that demonstrate its availability and resiliency.
3.2 Related Work

Current anonymity systems can be broadly separated into three categories: specific applications, outproxies, and overlay networks. Specific applications include persistent data stores like Freenet [11] and GNUnet [6], remailers like Mixmaster [34] and Mixminion [12], or file sharing like OneSwarm [22]. These rely on an anonymous transport layer, but do not expose it for more general applications. Outproxies provide generic IP connectivity to a host on the public Internet. Overlay networks provide anonymous communication between a closed set of participants with an application-agnostic transport layer, which applications like file sharing, instant messaging, email, etc. can be built upon.

These systems have employed two fundamental strategies for achieving anonymous transport.

The first strategy, which is used by the majority of anonymity systems, is source address rewriting, in which intermediate nodes function as relays that forward data on behalf of an originator or another relay. This may happen in one-hop, with a trusted centralized proxy like the Anonymizer [1] outproxy, or in multiple hops. In multiple hop source rewriting systems, packet flows either take an indiscriminate random path through the relays, as in the case of the outproxy Crowds [36], or follow a pre-selected path, called tunnels or circuits, through available relays, as in the case of the original mixnet idea by Chaum [10], the Onion Routing scheme [35], the outproxies Tor [14], Hordes [29], Tarzan [15], and the overlay network I2P [2], as well as the overlay network of Tor’s hidden services mode [14]. These multiple hop tunnel systems use some implementation of mixnets or onion routing, a scheme in which the addresses of intermediate relays and the final destination are hidden under layers of encryption that are peeled off at each relay, revealing the address to forward the packet to.

These systems (except for Crowds [36], which operates in plaintext), require cryptography to hide the intermediate and final destination addresses. Most of these systems also require special “Samaritan” nodes to function, which operate as free relays on behalf of the system’s users, and introduce the problem of maintaining a
trusted directory of the relays. In addition, the stateful nature of the multiple hop tunnel approach also makes traffic analysis a serious problem to be addressed.

The anonymous data store Freenet [11] uses a source address rewriting system with multiple hops to retrieve the data associated with a key in the distributed data store. However, it differs from the outproxies and overlay network systems mentioned above in that the path a request takes is determined by a routing table at each peer, instead of a pre-selected path. Each peer which forwards the request for that key closer to a likely owner of the key, who answers with a reply that returns back through the path the request took to its origin.

The other fundamental strategy for achieving anonymous transport has been broadcast, in which messages are delivered to all receivers. Broadcast mediums intrinsically confer receivers with anonymity, and so this shifts the problem of anonymity to concealing the sender. However, broadcast mediums are inefficient to implement over a unicast network, such as the Internet. Some systems have tried to leverage the multicast functionality built into IPv4, like Hordes [29], which uses multiple hop source rewriting in the forward path, but IPv4 multicast for the return path of data. Building an application with multicast today faces the challenges of configuring network hardware.

\( P^5 \) [37] and Herbivore [18] are anonymous overlay networks based on broadcast. They address the problem of scaling a broadcast medium over unicast connections by partitioning participants into a tree, in the case of \( P^5 \), or a ring of small “cliques” in the case of Herbivore. \( P^5 \) users encrypt their messages for their receiver with public-key cryptography, broadcast their messages to a subset of the tree, and use dummy traffic to mask transmission timing. Herbivore users broadcast in their local clique with a generalization of the dining cryptographer networks (DC-nets) [9] protocol, a secure scheme for transmitting information with sender anonymity. Herbivore’s implementation of it requires public-key cryptography.

These systems do not require Samaritan nodes, but they have more complex network architectures (trees and rings of cliques) and protocols, compared to those of the source rewriting systems. They also depend on public-key cryptography.
In the field of network coding and anonymity, [38] extends the idea of fixed transmission schedules for session anonymity, by sending network coded traffic instead of dummy cover traffic. The paper demonstrates significant gains in throughput.

Gossip based protocols for message dissemination originated in [21], and network coded gossip protocols have been explored in [26], [25], [13]. [13] characterizes the benefits in efficiency of gossiping network coded messages rather than individual messages.

3.3 System Overview

The Network Coded Gossip for Anonymous Broadcast (NCGAB) system is a decentralized peer-to-peer overlay network that provides a resilient, anonymous broadcast medium. It is built on a random linear network coded gossip protocol. NCGAB is designed for broadcasting messages among small to medium-sized groups, e.g. 10 to 100 participants, but may also be used as a network layer for unicast or multicast sessions carried within the broadcast. Participants may contribute messages with NCGAB and expect the system to deliver those messages to all other participants with high probability and with timeliness.

NCGAB does not provide data confidentiality; participants are expected to encrypt the data they wish to hide. We assume anonymity and confidentiality to be orthogonal problems.
Figure 3-1 illustrates the star network structure of an NCGAB network, showing NCGAB peers gossiping asynchronously with one another.

### 3.3.1 Anonymous

NCGAB guarantees unconditional receiver anonymity by its broadcast design, and features sender anonymity that improves as the network scales. NCGAB achieves this by strategically introducing new messages and always mixing a participant’s messages with those of other participants.

### 3.3.2 Decentralized

NCGAB utilizes a peer-to-peer network coded gossip protocol to efficiently disseminate messages among its participants. NCGAB is unique in achieving its properties in that it has a completely decentralized architecture, aside from its bootstrapping mechanism. The system does not require a key-based cryptosystem, an infrastructure of trust, Samaritan, coordinating, or trusted nodes to operate. The only trust is in the client software executed by each peer.

### 3.3.3 Highly Available

The primary metric of NCGAB’s successful operation is availability: the average percentage of participants that will successfully decode a message inserted into the broadcast medium by another participant. Simulations show that NCGAB achieves near 100% availability on a cooperative network, with an average delay of under 5 gossip rounds, while supporting a wide range of network sizes without a change in its system parameters.

NCGAB ensures the continued availability of the broadcast medium by its gossiping strategy and participant cooperation, which enforce the ephemerality of messages. We call the broadcast medium ephemeral, because messages do not persistent on the medium, and the active messages on the medium change over time.
3.3.4 Resilient

NCGAB is highly resilient to uncooperative participants, or Byzantine adversaries attempting to compromise the availability of the medium by pollution of valid or invalid content. NCGAB recognizes and discards invalid content, and simulations show it can sustain an availability of well over 60% when 30% of the network is operated by independent adversaries actively polluting the network. This resiliency holds across a wide range of network sizes (10 to 100 peers) without a change in system parameters.

3.4 Network Coding Tools

3.4.1 Network Coded Gossip Protocols

A gossip protocol achieves the dissemination of one or more messages among a group by exchanging “gossip” in multiple rounds. Gossip is considered any message a node previously received from another node. A node sends gossip to, or receives gossip from, a random subset of the group on each round. [13] outlines the protocol of a basic Random Message Selection gossip system, in which a node transmits one of its previously overhead messages to another node in each round. [13] shows that such a Random Message Selection gossip system with n nodes and k distributed messages among the nodes takes \( \Theta(n \log(n)) \) rounds of gossip to disseminate all k messages (where \( k = O(n) \)).

A network coded gossip protocol exchanges linear combinations of messages in its rounds of gossip, instead of whole messages. As sufficient linear combinations are collected, nodes are able to decode messages and encode new linear combinations over a different subset of decoded messages. [13] outlines a protocol of a basic Random Linear Coding gossip system, in which a node transmits a linear combination of previously decoded messages to another node in each round. [13] shows that such a Random Linear Coding gossip system with n nodes and k distributed messages among the nodes take \( O(n) \) rounds of gossip to disseminate all k messages (where
\( k = O(n) \).

NCGAB is based on a Random Linear Network Coding gossip protocol, but incorporates modifications for anonymity and content integrity. These are documented in the System Architecture section, below.

### 3.4.2 Homomorphic Hashes

A homomorphic function is a function that also maps certain operations at the input to certain operations on the output. For example, a function may exhibit this homomorphism:

\[
H(\alpha M_1 + \beta M_2) = H(M_1)^\alpha H(M_2)^\beta
\]

A cryptographic hash function with a homomorphism that maps the hash of a linear combination at its input to some function of constituent message hashes at its output, like in the example homomorphism above, is particularly well suited for verifying the integrity of linear combinations formed with linear network coding.

For example, a receiver may verify a linear combination of message vectors \( M_1 \) and \( M_2 \) with scalar coefficients \( \alpha \) and \( \beta \) it received, by testing the equality in the equation above. That is, by computing the homomorphic hash of the linear combination, and comparing it to the same hash derived from the function’s homomorphic mapping, involving the coefficients \( \alpha \) and \( \beta \) and individual message hashes \( H(M_1) \) and \( H(M_2) \). This allows a receiver to verify the integrity of any linear combination of \( M_1 \) and \( M_2 \), with the knowledge of the constituent message hashes.

[28] presents a homomorphic hash function for rateless erasure codes, [17] presents one for linear network coding, which exhibits the homomorphism exemplified above. [40] [7] [4] [24] describe homomorphic signature schemes. These systems may be used with a preshared key between the sender and the receiver, so the receiver may authenticate linear combinations formed by intermediate untrusted nodes.

NCGAB identifies messages by their homomorphic hashes, and includes this identification with every transmitted linear combination to allow for peers to verify the
integrity of the linear combinations, and thereby prevent the pollution of invalid content by attackers.

### 3.5 System Architecture

In the subsections below, we describe the relevant architectural details of the NCGAB system. We intentionally omit details of underlying transport and network layer protocols (e.g. TCP/IP), as well as the initial bootstrapping mechanism that determines the participating peers of the overlay network, as these choices are flexible and implementation-specific. For now, we assume the NCGAB client is bootstrapped with a list of participating peers, and that the list is static.

We call NCGAB client the software that runs on each participating peer in the NCGAB overlay network. We call a *decoded message* any message that is decoded by an NCGAB client from linear combinations it received from the overlay network. We call a *contributed message* any message that is inserted by an NCGAB client into the
overlay network. We refer to the application as the user of the NCGAB client, which receives decoded messages from, and possibly contributes messages to, the broadcast medium provided by the NCGAB system.

3.5.1 Gossip and Decoding Strategy

NCGAB achieves a shared broadcast medium by network coded gossip. Each participant in the system gossips to a randomly selected subset of the group at regular intervals, asynchronously of the other participants. The size of this random subset is expressed as a percentage. The duration between gossip rounds expressed in seconds. These are the system parameters LOOKUP_PERCENT and GOSSIP_INTERVAL, respectively.

Gossip is pushed by participants: on its gossip round, the participant connects to each member of the randomly chosen subset and transmits gossip. The choice to push gossip, rather than pull it (connect and ask for gossip), is not arbitrary and ensures greater anonymity in the system. We justify this design choice in more detail in the Analysis of Anonymity section.

The gossip transmitted by a participant consists of one linear combination: either a newly formed linear combination from messages the participant has decoded or contributed, called new-gossip, or a linear combination selected from gossip the participant has previously received, called re-gossip. New-gossip introduces new innovative linear combinations of decoded messages into the network, while re-gossip helps disseminate existing linear combinations throughout the network.

All new-gossip is formed from a fixed number of decoded messages. This number is the system parameter CODE_SIZE. Since all re-gossip originates from new-gossip, all linear combinations that exist in the overlay network are coded with CODE_SIZE number of messages.

Participants periodically execute a decoding routine. This routine arranges all decoded messages and all undecoded linear combinations into a matrix and carries out Gaussian elimination to decode new messages. This may yield zero or more newly decoded messages.
3.5.2 Ephemeral Message Window

The gossiping and decoding strategy outlined above is sufficient to implement a shared broadcast medium, and guarantees the anonymity described in the Analysis of Anonymity section, but quickly exhibits unacceptable delay and computational overhead as the number of messages contributed to the medium grows. This is in great part due to newer messages having decreasing probability of selection in new-gossip. In addition, past messages are recoded and re-transmitted among the participants, forming cycles that consume bandwidth. Furthermore, new messages may be coded with very old messages, requiring increasingly large decoding matrices to decode.

In order to ensure the continued availability of the medium, tractable decodability, and reasonable delays, the NCGAB system implements a pruning strategy that limits the number of active messages in the overlay network. This is achieved with a client-side data structure we call an ephemeral message window, shown in Figure 3-2. This data structure is core to the operation of the NCGAB system. Each client maintains two of these windows: one Decoded Window, for messages that are decoded from the network or contributed by the application, and one Gossip Window, for linear combinations that are received from the network.

Items inserted into these windows are tagged with a time-to-live field: a lifetime expressed as an integer number of gossip rounds. On each gossip round, the time-to-live of every item in the Decoded and Gossip Windows is decremented. When an item’s time-to-live reaches zero, it is flagged expired. Expired messages in the Decoded Window are not coded into new-gossip, but are saved and still used for future decoding. Expired linear combinations in the Gossip Window are not selected for re-gossip, not used for future decoding, but are retained for some time to prevent their re-insertion.

Decoded and contributed messages are inserted into the Decoded Window with an initial time-to-live of the system parameter $\text{TTL}_{\text{DECODED}}$. Only new messages may be inserted in the Decoded Window – the time-to-live of expired are not reinstated, nor is the time-to-live of unexpired messages updated, if they are decoded again.
Linear combinations received from the network are inserted into the Gossip Window, filed under the source peer address they originated from, with an initial time-to-live of the system parameter \texttt{TTL\_GOSSIP}. Only new linear combinations may be inserted in the Gossip Window. Like the Decoded Window, the time-to-live of an expired or active linear combinations is not reinstated or updated if it is received again.

The Gossip Window structure differs slightly from the Decoded Window structure by the organization of linear combinations under source peer addresses, but otherwise the time-to-live mechanism functions identically. See Figure 3-2 for an example.

The ephemeral message window structures restrict the lifetime of messages on the broadcast medium which, in turn, quenches any cycles of linear combinations containing messages that all participants have already decoded. This is because the insertion policy for the Decoded Window prohibits inserting messages that have already expired. A message will eventually expire in the Decoded Windows of all participants, and will not be coded into new-gossip. The insertion policy for the Gossip Window also prohibits inserting expired linear combinations, so any remaining gossip circulating the network that contains the message decoded by all will eventually expire as well.

### 3.5.3 Amended Gossip Strategy

The NCGAB gossip strategy is essentially the same as previously described, except that gossip is formed or drawn from the ephemeral windows. Each participant transmits one linear combination, either new-gossip or re-gossip chosen with equal probability, to a random subset of the network on every gossip round. The gossip is selected anew for every peer contacted from the random subset. New-gossip is formed by coding \texttt{CODE\_SIZE} number of unexpired messages from the Decoded Window. Re-gossip is chosen by selecting an unexpired linear combination from the Gossip Window.

The unexpired messages selected to form new-gossip are randomly selected from the Decoded Window, weighted by their time-to-live. Re-gossip is chosen by first randomly selecting a source peer addresses in the Gossip Window, followed by a
random selection of a linear combination under that peer address, weighted by time-
to-live. This re-gossip selection strategy enforces fair re-gossiping in the system, so that a polluting attacker cannot dominate the gossip traffic with only his or her linear combinations.

If there are an insufficient number of messages in the Decoded Window to form a linear combination comprised of CODE\_SIZE messages, then randomly generated messages are added to the Decoded Window until it has at least CODE\_SIZE messages. In simulations, we have found that this is only required at initialization, when the Decoded Window at all peers is empty. In that case, the dummy messages bootstrap the gossiping process.

Randomly selecting messages from a skewed distribution weighted by their time-
to-live, rather than from a uniform distribution, improves the availability of the medium and helps diagonalize the decoding matrix of the participants. Its effect is shown in the Simulations and Results section.

Altogether, the ephemeral message window data structure and corresponding gos-
sip strategy yield an ephemeral broadcast medium that can sustain new messages contributed to the medium over time.

3.5.4 Message and Packet Structure

![Example Message Structure](image)

Figure 3-3: Example Message Structure

A message in NCGAB is represented by a fixed length sequence of bytes with no header. A message is identified only by the homomorphic hash of its contents, which is unique for an instantiation of an NCGAB overlay network with high probability, and enables the Decoded Window to track expired messages and enforce its insertion policy. Figure 3-3 illustrates the structure of a message.
A linear combination in NCGAB is represented by a list of constituent messages hashes, a list of scalar coefficients, and the data containing the linearly combined messages. Linear combinations are also uniquely identifiable, as they are comprised of uniquely identifiable messages. The integrity of a linear combination structure can be verified by computing the homomorphic hash of its data, and comparing it to a second hash derived from the coefficients and constituent message hashes. The second hash is computed with the homomorphism exhibited by the hash function.

Figure 3-4: Example Gossip Packet Structure

Each gossip communication transmits a linear combination in a packet. A packet only contains the encoding of the linear combination structure described above. For example, Figure 3-4 illustrates a gossip packet with a CODE_SIZE of 4, 4-byte message hashes, 1-byte scalar coefficients, and a 1024-byte linear combination.

When an NCGAB client receives a gossip packet, it verifies the hashes of the encoded linear combination. NCGAB clients discard packets with invalid linear combinations.
3.5.5 Message Contribution by Application

The CONTRIBUTE_INTERVAL system parameter specifies the rate at which an application should contribute a new message to the NCGAB to be inserted into the broadcast medium. It is expressed in number of gossip rounds. An application contributes a message to NCGAB by adding the message to the Decoded Window with a time-to-live of TTL_DECODE.

Together, the CONTRIBUTE_INTERVAL and GOSSIP_INTERVAL system parameters determine the maximum throughput of a participant on the broadcast medium, assuming that the other system parameters are selected appropriately and the rest of the participants cooperate.

3.5.6 System Parameters

Table 3.1: NCGAB System Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOSSIP_INTERVAL</td>
<td>Duration of time between gossip rounds</td>
</tr>
<tr>
<td>LOOKUP_PERCENT</td>
<td>Percentage of peers to contact each gossip round</td>
</tr>
<tr>
<td>CODE_SIZE</td>
<td>Number of messages to code into new-gossip</td>
</tr>
<tr>
<td>TTL_DECODE</td>
<td>Initial time-to-live of items in Decoded Window</td>
</tr>
<tr>
<td>TTL_GOSSIP</td>
<td>Initial time-to-live of items in Gossip Window</td>
</tr>
<tr>
<td>CONTRIBUTE_INTERVAL</td>
<td>Number of rounds between message contributions</td>
</tr>
</tbody>
</table>

Table 3.5.6 gathers all of the system parameters in the NCGAB system. It is important to note that these system parameters can only be enforced in client-side software, because the NCGAB system is completely decentralized. Nevertheless, we will show in simulations that it is in the best interest of participants to cooperate on a common set of system parameters to maximize availability of the medium for themselves and all other participants. Clients that do not cooperate with the chosen system parameters are regarded as adversaries, and their effect on the system is analyzed in detail in the Analysis of Attacks section.

The system parameters that govern the ephemerality of the NCGAB broadcast medium and rate of dissemination require tuning to ensure both the stability and successful decodability of messages contributed to the medium. For example,
if the Decoded Window time-to-live parameter, \texttt{TTL\_DECODE}, is too small, or the \texttt{CONTRIBUTE\_INTERVAL} is too high, some messages may expire before sufficient linear combinations of them are formed and gossiped to allow for their decoding by other participants. Alternatively, if the Decoded Window or Gossip Window time-
to-live parameters are too large, the windows may grow too quickly, as messages are contributed to the medium faster than they are expired.

The aggregate behavior of the NCGAB system depends on all of its system parameters. At first glance, it may seem daunting to choose set of parameters that yield a highly available broadcast medium. However, extensive simulations show that it is not difficult to construct a parameter set that achieves a highly available medium for all participants, supporting a network size ranging from 10 to 100 peers.

### 3.5.7 System Pseudo-code

Despite the added complexity of the ephemeral message window data structures, the NCGAB system can still be easily summarized and implemented. Figure 3.5.7 documents the high-level Python-like pseudocode of the system described above.

### 3.6 Analysis of Anonymity

In this section, we clarify our definitions of anonymity and analyze the receiver and sender anonymity of participants in the NCGAB overlay network.

We assume message contents cannot be correlated with a sender or a receiver. We take unconditional \texttt{receiver anonymity} to mean that the traversal of a particular message in the overlay network cannot be correlated with its intended receiver, and unconditional \texttt{sender anonymity} to mean that the traversal of a particular message in the overlay network cannot be correlated with its sender. These properties are assessed from a bird’s eye or omniscient point of view of message traversal over the overlay network.

It is prudent to note that any participation in an overlay network designed for anonymity might be considered as sufficiently implicating an attempt at anonymous
Peers = []
Decoded_Window = Window()
Gossip_Window = Window()

# Gossip receiving thread
def thread_receive():
    while True:
        gossip = network.recv()
        # If the gossip hash verifies
        if validate(gossip) == True:
            if gossip not in Gossip_Window:
                Gossip_Window.add(gossip, TTL_GOSSIP)
                decode()

# Gossip sending thread
def thread_gossip():
    while True:
        # Decrement TTL of objects in windows
        for msg in Decoded_Window:
            if Decoded_Window[msg].ttl > 0:
                Decoded_Window[msg].ttl -= 1
        for lc in Gossip_Window:
            if Gossip_Window[lc].ttl > 0:
                Gossip_Window[lc].ttl -= 1
        # Keep our Decoded Window filled to CODE_SIZE
        if len(Decoded_Window) < CODE_SIZE:
            for i in range(CODE_SIZE - len(Decoded_Window)):
                Decoded_Window.add(RandomMessage(), TTL_DECODE)
        # Look up a random subset of our peers
        subset_peers = lookup_random(Peers, LOOKUP_PERCENT)
        for peer in subset_peers:
            # Either code new-gossip or send re-gossip
            if random.coinflip() == True:
                gossip = Decoded_Window.code_random(CODE_SIZE)
            else:
                gossip = Gossip_Window.choose_random()
            network.send(peer, gossip)
sleep(GOSSIP_INTERVAL)

# Decode the gossip window
def decode():
    matrix = Decoding_Matrix()
    # Add solved rows of messages from Decoded Window
    for msg in Decoded_Window:
        matrix.addSolvedRow(msg)
    # Add unsolved rows of linear combinations
    # from Gossip Window
    for lc in Gossip_Window:
        matrix.addCombination(lc.data, lc.coefficients)
    # Gaussian Elimination
    decoded = matrix.eliminate()
    # Add newly decoded messages to Decoded Window
    for msg in decoded:
        if msg not in Decoded_Window:
            Decoded_Window.add(msg, TTL_DECODE)
    <Deliver decoded msg to application>

# Application hook for contributing a message
def contribute(message):
    Decoded_Window.add(message, TTL_DECODE)
communication. We assume that such participation is tolerated, that the act of participating in the network is not considered a compromise of anonymity, and that the participant's concerns of anonymity are regarding the origination and delivery of messages within the overlay network.

We model the NCGAB overlay network as a graph of participating peers represented as nodes, interconnected with links. A link exists between every node, because a node may gossip with any subset of the entire overlay network. For brevity, to observe a message on a link means to observe a linear combination on the link that references that message in its list of constituent message hashes. See the Message and Packet Structure subsection of the System Architecture section for more information on structure of transmitted linear combinations. Refer to Figure 3-6 for an example model of an NCGAB overlay network with five nodes.

3.6.1 Traffic Analysis by an Omniscient Adversary

First, we consider the threat model of an omniscient adversary that may observe all messages exchanged on all links. This is to assess whether or not NCGAB possesses unconditional receiver anonymity and unconditional sender anonymity, as defined above.

NCGAB trivially guarantees unconditional receiver anonymity, because the system is designed to provide a broadcast medium, which does not employ any concept of specific destinations. The system's goal is to disseminate the contributed message of any participant to every other participant, by having each participant gossip to
a random subset of the network on each round. The random subset contacted by each participant on each round may or may not include the intended destination for a message, but that distinction only exists from a purely semantic, not technical, point of view, and we assume that message contents cannot be correlated with their intended destination.

The NCGAB system does not have sender anonymity when subject to traffic analysis by an omniscient adversary. An omniscient adversary may pinpoint the introduction of a message, identified by its hash, to a linear combination leaving a particular node on the network, before that same linear combination is re-gossiped, or the message is later decoded from sufficient linear combinations and recoded into new-gossip by other peers.

3.6.2 Traffic Analysis by a Limited Adversary

We now consider the threat model of a more limited adversary, which may operate one or more participants on the NCGAB overlay network, and can only observe the messages sent on the links to these participants. Unconditional receiver anonymity is still clearly guaranteed by the same argument as in the case of the omniscient adversary. We investigate the kind of traffic analysis a limited adversary may perform to correlate the original sender with a message, and with what certainty he or she may do so.

**Importance of the Push Gossip Mechanism**

The decentralized nature of NCGAB prevents peers from enforcing the `LOOKUP_PERCENT` parameter, which defines the size of the randomly chosen subset each participant gossips to on each gossip round. This means that an adversary is able to make connections to 100% of the overlay network and push gossip to them, without recourse, on every gossip round, since peers are incapable of detecting or preventing these asynchronous connections. The most a peer can enforce is the `GOSSIP_INTERVAL` parameter on inbound connections, by rejecting inbound connections that occur more frequently than
the parameter specifies.

We provide this context to make the point that if the NCGAB gossiping strategy relied on a pull mechanism, rather than a push mechanism, then an adversary would be able to pull gossip from every peer on every gossip round. This would allow the adversary to isolate the introduction of a particular linear combination containing a new message to its source with higher probability, but not necessarily with the certainty that an omniscient adversary can. There is still a chance that the contacted peer does not code the new message it is responsible for contributing in the gossip pulled from it by the adversary. That peer may then code the new message into gossip that is delivered to another peer that gossip round, asynchronously, which would introduce uncertainty on the origin of the linear combination when the adversary observes it later.

Employing a push mechanism for gossiping limits the information of message ownership available to an adversary, and increases the uncertainty in the sender of a message. A peer will only expose LOOKUP_PERCENT of the overlay network on one round to the introduction of a linear combination containing a message it is responsible for contributing. The linear combinations containing that message will be disseminated second-hand from then on, increasing the uncertainty about the original sender of the message.

Uncertainty Regarding Sender

NCGAB leverages two uncertainties to protect the anonymity of a sender.

The first uncertainty is whether the linear combination gossiped by a peer is new-gossip or re-gossip. If it is new-gossip, the peer must possess the decoded message, because it created a new linear combination containing it. If it is re-gossip, then the peer may or may not posses the decoded message.

The second uncertainty is whether a message that was coded into new-gossip was contributed by that peer or decoded from linear combinations that peer received.

Both of these uncertainties increase with each round after the introduction of a message, because other peers will begin to re-gossip linear combinations containing
it, as well as eventually decode it and code it into new-gossip themselves.

Intuitively, the greater presence an adversary has on the overlay network, the less uncertainty there is in identifying the sender of a particular message on a particular round. In future work, we will develop an expression for the probability of an adversary witnessing the introduction of a message contributed by a particular peer.

3.7 Analysis of Attacks

In this section, we analyze the potential of attacks carried out by one or more adversaries participating in an NCGAB overlay network. The Simulations and Results section including the results from simulations of the more effective attacks discussed here, which pose a potential threat to NCGAB’s operation and availability.

All attacks at the NCGAB system level require the attacker’s participation in the overlay network, otherwise the attacker cannot send or receive gossip to/from the participants. The Analysis of Anonymity section investigated the efficacy of passive traffic analysis, so we focus on active attacks in this section. We consider active attacks to be any action carried out by an adversary with intentional deviation from the NCGAB protocol or the cooperative system parameters, in an effort to compromise the availability of the broadcast medium.

3.7.1 Threat Model, Attack Vectors, and Goal of an Adversary

Our threat model considers one or more independent adversaries participating on the NCGAB overlay network. By independent, we mean that the adversaries do not collude in mounting their attack. We address colluding adversaries in the subsection Pollution with Colluded Decodable Gossip.

We assume that adversaries have unique network addresses and cannot spoof other network addresses. This is to model the typical case of a Wide Area Network in which users are connected through Internet Service Providers that prohibit spoofing. A more
serious adversary may have the power to spoof network addresses, but we put this aside for now. The rigid structure of the NCGAB packets and the simplicity of the system expose few attack vectors for an adversary. Most of the system parameters govern client-side structures that are not directly accessible to an adversary. We assume that an adversary may transmit arbitrary linear combinations to any number of peers (exceeding `LOOKUP_PERCENT`) within a gossip round. The decentralized nature of NCGAB prevent peers from detecting or preventing asynchronous connections exceeding `LOOKUP_PERCENT` of the network. We assume that an adversary cannot contact a peer more than once within a gossip round, as these excessive connections can be detected by a peer.

The goal of the adversary is to disable or diminish the availability of the NCGAB broadcast medium. One strategy to achieve this is to consume the network resources that would otherwise be used to disseminate content. The Inactive Peer subsection describes this. In addition to this, an adversary may try to actively disrupt or dominate the content on the medium. The Pollution attack subsections address various approaches to this. We assume all of these attacks may be multiplied by the adversary, by operating more independent participants on the network (e.g. a Sybil attack).

We simulate the most promising attacks to empirically determine their effect on the NCGAB system, and discuss those results in the Simulations and Results section.

### 3.7.2 Inactive Peer

An inactive peer attacker receives gossip from peers on the network but does not send useful gossip to peers. In effect, this attacker wastes the bandwidth of peers and hampers the dissemination of legitimate content on the network. We note that a disconnected peer may also function as an inactive peer.

Simulations in the Simulation of an Inactive Peer demonstrate that there exist system parameters for which the NCGAB system may tolerate many inactive peers, without affecting the availability of the medium.
3.7.3 Pollution with Invalid Gossip

Pollution with invalid gossip is an attacker’s attempt at corrupting a participant’s decoding of one or more messages by contributing invalid linear combinations that claim to provide information for legitimate messages. An attacker carrying out this attack is also an inactive peer, that is, the attack does not aid in dissemination of legitimate content on the network. A participant that inserts an invalid linear combination into his or her decoding matrix may later decode one or more messages incorrectly with that row.

NCGAB prevents this by identifying messages solely by the hash of their contents, and including this identification with the linear combination encoded in each packet (see Message and Packet Structure subsection). This enables clients to verify received linear combinations by computing the homomorphic hash of the linear combination and comparing it to a second hash derived from the coefficients and constituent message homomorphic hashes included with the linear combination. This makes it exceedingly difficult for an attacker to craft a linear combination with invalid content that still hashes correctly.

We assume that this attack is infeasible, as its detection is enforced by the message hashing incorporated in the NCGAB protocol.

3.7.4 Pollution with Old or Underdetermined Gossip

Pollution with old gossip or underdetermined gossip is an attacker’s attempt at circulating irrelevant content that fills all of the Gossip Windows of participants, which in turn leads to that content’s re-gossip on the network. As before, an attacker carrying out this attack is also an inactive peer. Old gossip are linear combinations of messages the broadcast medium has previously delivered to its participants. Underdetermined gossip are insufficient linear combinations of randomly generated messages, chosen such that the randomly generated messages cannot be decoded.

Old gossip may be decoded by a participant into a valid message, but the message will not be inserted into the participant’s Decoded Window if it has been decoded...
in the past and expired. Underdetermined linear combinations are unlikely to be decoded into messages, by their construction. Therefore, either kind of gossip will only temporarily persist in the Gossip Window of participants.

NCGAB’s selection strategy for re-gossip limits the effectiveness of this attack, because NCGAB will randomly choose from available peer source addresses in the Gossip Window, before randomly choosing a linear combinations under that address. This means that the cooperative portion of the network will fairly re-gossip contributions from adversaries and legitimate participants alike.

Simulations in the Simulation of Pollution with Underdetermined Gossip demonstrate the effect of multiple underdetermined gossiping attackers. We observe that the net effect is similar to that of the Inactive Peer attack, and that NCGAB tolerates it without losing availability.

### 3.7.5 Pollution with Decodable Gossip

Pollution with decodable gossip is an attacker’s attempt at circulating gossip that will be decoded as quickly as possible, in an attempt to flood the Gossip and Decoded Windows of all participants. We consider this to be the most formidable attack of one or more independent attackers on NCGAB.

As before, an attacker carrying out this attack is also an inactive peer. In the pollution with decodable gossip attack, an adversary generates CODE\_SIZE number of random messages, and transmits a linear combination of them to every peer for CODE\_SIZE gossip rounds. After CODE\_SIZE gossip rounds, the peers will successfully decode the generated messages and insert them into their Decoded Window. By then, peers will have also inserted CODE\_SIZE linear combinations into their Gossip Window. The adversary then generates new CODE\_SIZE number of random messages, and repeats the process over the next set of gossip rounds. This is the fastest rate at which an independent adversary can fill the ephemeral message windows of all peers.

Simulations in the Simulation of Pollution with Decodable Gossip section demonstrate the effect of this attack with multiple independent adversaries. We show that increasing the CODE\_SIZE system parameter improves NCGAB’s resiliency to this
attack, and that the system is able to maintain availability even when 30% of the network is operated by independent adversaries carrying out this attack.

### 3.7.6 Pollution with Colluded Decodable Gossip

A derivative of the Pollution with Decodable Gossip attack is one that involves multiple colluding attackers, rather than independent attackers. Independent attackers polluting with decodable gossip can only fill the Decoded Window of all participants with $\text{CODE\_SIZE}$ messages in $\text{CODE\_SIZE}$ gossip rounds. However, every colluding adversary adds an additional opportunity in a gossip round to deliver decodable gossip in less than $\text{CODE\_SIZE}$ gossip rounds, by having the adversaries code over the same messages. We consider this to be the greatest threat to the availability of the NCGAB broadcast medium.

Future work will include simulations that characterize the effect of multiple colluding attackers polluting decodable gossip.

### 3.8 Simulations and Results

The stochastic nature, ephemeral message windows, and multiple system parameters of the NCGAB system make it challenging to model the system analytically and derive closed-form expressions for the effect of its system parameters. We opted instead to analyze NCGAB empirically, by developing a comprehensive simulation of the system.

In this section, we present the basic design of an NCGAB system simulator and the results of a comprehensive simulation the system, which demonstrates the system’s ability to provide a highly available and resilient broadcast medium.

#### 3.8.1 ncgabsim Simulator

The ncgabsim simulator was developed in Python3 [3] and totals approximately 900 lines of code, 460 of which are the implementation of the peer client, and the rest of which are for statistic collection and event logging. The simulated client implements
the architecture described in the System Architecture section, as well the implied finite field arithmetic and Gaussian elimination routines to support random linear coding and decoding. The arithmetic routines comprise the majority of the client code. We chose not to implement homomorphic hashing to verify linear combinations on the network, as its role is straightforward and not relevant in assessing the behavior of the system and the effect of its various parameters. The complete simulator source code is provided in ncgabsim Simulator Code.

The simulator models every peer of the NCGAB network with its own Decoded Window, Gossip Window, and packet input queue. These three containers comprise the entire state of an individual peer. The network is modeled with perfect links, implemented by a list containing the references to each peer’s input queue object. A peer may put a packet in any other peer’s input queue.

The simulation does not model the message contents, but does model the distinction between contributed messages, randomly generated dummy messages, and linear combinations of messages. Messages are tagged with randomly generated unique identifiers, which simulate the hash they would be identified by in the real system. The simulator tracks the origin of all messages for the purpose of collecting statistics of their dissemination. The simulation implements random linear coding, including the underlying finite field arithmetic and Gaussian elimination, for a Rijndael $GF(2^8)$ field. Peers faithfully solve linear combinations with Gaussian eliminations to decode their constituent packets.

The simulation runs by executing rounds of gossip, in which every peer performs the following routine: contribute a new message if its contribution timeout has expired, receive gossip from its input queue and add it to its Gossip Window, attempt to decode its current Gossip Window, and gossip with a random subset of the network. The simulation is a good approximation of the real system, so long as the events it models for each peer (decoding, sending gossip, etc.) are not time-constrained by the GOSSIP_INTERVAL. Simulating rounds of gossip, rather than realtime connectivity, produces results that are more fundamental to the system and independent of timing variation that may depend on implementation. For each round of gossip, the simula-
Table 3.2: Available ncgabsim Simulation Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIM.WARMUP.DECODES</td>
<td>Number of message decodes before recording starts</td>
</tr>
<tr>
<td>SIM.DURATION.INSERTS</td>
<td>Duration of the simulation in terms of messages contributed by peers</td>
</tr>
<tr>
<td>SIM.NUM.PEERS</td>
<td>Total number of peers in the overlay network</td>
</tr>
<tr>
<td>LOOKUP.PERCENT</td>
<td>Percentage of peers to contact each gossip round</td>
</tr>
<tr>
<td>CODE.SIZE</td>
<td>Number of messages to code into new-gossip</td>
</tr>
<tr>
<td>TTL.DECODE</td>
<td>Initial time-to-live of items in Decoded Window</td>
</tr>
<tr>
<td>TTL.GOSSIP</td>
<td>Initial time-to-live of items in Gossip Window</td>
</tr>
<tr>
<td>CONTRIBUTE.INTERVAL</td>
<td>Number of rounds between message contributions</td>
</tr>
</tbody>
</table>

The simulator also collects statistics regarding the size of the Decode and Gossip Windows, the size of decoding matrices, which messages were used in the decoding matrix, and the number of decoded results from Gaussian elimination of a decoding matrix. This data is post-processed to assess the stability of the system by checking that ephemeral message windows do not exhibit unreasonable or unbounded growth, which would pose significant computational overhead for decoding and may indicate instability.

Table 3.8.1 summarizes the parameters available in the simulation for tuning.
3.8.2 Availability and Delay

We define availability as the average percentage of peers that decode an inserted message. It represents, on average, how likely an inserted message will be successfully broadcast to all peers. The delay represents, on average, how many gossip rounds it will take for an inserted message to be decoded by a peer, given that it will be successfully decoded by it.

The availability and delay are computed with the statistics collected from DURATION_INSERTS number of tracked messages.

3.8.3 Cooperative Peer Simulations

The main goal of the cooperative peer simulations were to characterize the typical behavior of the NCGAB system: how its availability and delay vary with respect to the system parameters. The second goal was to find a set of system parameters for which there is nearly 100% availability with reasonable delay.

We began by fixing two parameters: LOOKUP_PERCENT to 0.25, and CODE_SIZE to 4. Setting LOOKUP_PERCENT to a fraction of less than 0.50 was in an effort to limit the demands on each peer for each gossip round, especially for larger networks. Setting CODE_SIZE to 4 seemed to be a reasonable trade-off between sufficient mixture of participant content for sender anonymity, while limiting the computational complexity of decoding at each peer and the runtime of the simulation.

We explored parameters of TTL_DECODE and TTL_GOSSIP with several trial runs of the simulation, until we observed successful decoding on the network for a variety of network sizes. We fixed TTL_DECODE to 30 and TTL_GOSSIP to 10, so decoded packets persist longer than gossip coded from them. These choices yielded a system that was highly scalable, from 10 to 100 peers.

This leaves the CONTRIBUTE_INTERVAL parameter, the duration in rounds of gossip participants should wait before contributing a new message, or the inverse of the maximum rate at which an adversary should introduce messages, as well SIM_NUM_PEERS, as the number of peers in the overlay network to be the free variables. We chose
CONTRIBUTE_INTERVAL to be the independent variable in the cooperative peer simulations, while availability and delay are the dependent variables, and SIM_NUM_PEERS is a parameter. Table 3.8.3 summarizes the parameter sweep. Figure 3.8.3 show availability versus CONTRIBUTE_INTERVAL parametrized by SIM_NUM_PEERS, and delay versus CONTRIBUTE_INTERVAL parametrized by SIM_NUM_PEERS. The plots are formed from averaging 5 runs of the simulation initialized with different seeds. 95% confidence intervals are provided around each data point.

Table 3.3: Cooperative Peer Simulation Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIM_WARMUP_DECODES</td>
<td>50</td>
</tr>
<tr>
<td>SIM_DURATION_INSERTS</td>
<td>100</td>
</tr>
<tr>
<td>SIM_NUM_PEERS</td>
<td>[10, 25, 50]</td>
</tr>
<tr>
<td>LOOKUP_PERCENT</td>
<td>0.25</td>
</tr>
<tr>
<td>CODE_SIZE</td>
<td>4</td>
</tr>
<tr>
<td>TTL_DECODER</td>
<td>30</td>
</tr>
<tr>
<td>TTL_GOSSIP</td>
<td>10</td>
</tr>
<tr>
<td>CONTRIBUTE_INTERVAL</td>
<td>[7, 10, 12, 17, 20, 22]</td>
</tr>
</tbody>
</table>

Figure 3-7: Cooperative Peer Simulation

The results of this simulation demonstrate that the set of system parameters tabulated above and a CONTRIBUTE_INTERVAL of 22, corresponding to the right most points in the plots, yield a broadcast medium with near 100% availability and under 5 gossip rounds of delay for networks that range from 10 to 50 peers.

The simulations below explore the gossip window random selection strategy and the effect of various attackers on the system.
3.8.4 Simulation of Uniform versus Weighted Item Selection

Table 3.4: Uniform versus Weighted Item Random Selection Simulation Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIM.WARMUP.DECODES</td>
<td>50</td>
</tr>
<tr>
<td>SIM.DURATION.INSERTS</td>
<td>100</td>
</tr>
<tr>
<td>SIM.NUM.PEERS</td>
<td>25</td>
</tr>
<tr>
<td>LOOKUP_PERCENT</td>
<td>0.25</td>
</tr>
<tr>
<td>CODE.SIZE</td>
<td>4</td>
</tr>
<tr>
<td>TTL.DECODE</td>
<td>30</td>
</tr>
<tr>
<td>TTL.GOSSIP</td>
<td>10</td>
</tr>
<tr>
<td>CONTRIBUTE.INTERVAL</td>
<td>[7, 10, 12, 15, 17, 20, 22]</td>
</tr>
</tbody>
</table>

NCGAB chooses messages from its Decoded Window and linear combinations from its Gossip Window randomly when forming new-gossip or selecting re-gossip. This simulation compares a uniform versus weighted by time-to-live random selection strategy for items in the Decoded and Gossip Windows. See Figure 3.8.4 for results. The greatly improved availability justifies the choice of a weighted by TTL selection strategy. The plots are formed from averaging 3 runs of the simulation initialized with different seeds. 95% confidence intervals are provided around each data point.

3.8.5 Simulation of an Inactive Peer

This simulation investigates the effect of increasing numbers of inactive peer attackers on the system, as described in Inactive Peer subsection of the Analysis of Attacks section on availability and delay. We use a CONTRIBUTE.INTERVAL of 20, and the other parameters from the cooperative peer simulation, as this resulted in a nearly
Table 3.5: Inactive Peer Attackers Simulation Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIM.WARMUP.DECODES</td>
<td>50</td>
</tr>
<tr>
<td>SIM.DURATION.INSERTS</td>
<td>75</td>
</tr>
<tr>
<td>SIM.NUM.PEERS</td>
<td>[10, 20, 30]</td>
</tr>
<tr>
<td>LOOKUP_PERCENT</td>
<td>0.25</td>
</tr>
<tr>
<td>CODE.SIZE</td>
<td>4</td>
</tr>
<tr>
<td>TTL.DECODE</td>
<td>30</td>
</tr>
<tr>
<td>TTL.GOSSIP</td>
<td>20</td>
</tr>
<tr>
<td>CONTRIBUTE_INTERVAL</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 3-9: Inactive Peer Attackers Simulation

100% available medium in the cooperative case. See Figure 3.8.5 for results. Inactive peers have little effect on availability and delay of the medium. The plots were formed from averaging 3 runs of the simulation initialized with different seeds. 95% confidence intervals are provided around each data point.

3.8.6 Simulation of Pollution with Underdetermined Gossip

Table 3.6: Pollution with Underdetermined Gossip Simulation Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIM.WARMUP.DECODES</td>
<td>50</td>
</tr>
<tr>
<td>SIM.DURATION.INSERTS</td>
<td>75</td>
</tr>
<tr>
<td>SIM.NUM.PEERS</td>
<td>[10, 20, 30]</td>
</tr>
<tr>
<td>LOOKUP_PERCENT</td>
<td>0.25</td>
</tr>
<tr>
<td>CODE.SIZE</td>
<td>4</td>
</tr>
<tr>
<td>TTL.DECODE</td>
<td>30</td>
</tr>
<tr>
<td>TTL.GOSSIP</td>
<td>20</td>
</tr>
<tr>
<td>CONTRIBUTE_INTERVAL</td>
<td>20</td>
</tr>
</tbody>
</table>

This simulation investigates the effect of increasing numbers of inactive peers
that also pollute with underdetermined gossip, as described in Pollution with Old or Underdetermined Gossip subsection of Analysis of Attacks section, on availability and delay. See Figure 3.8.6 for results. Like ordinary inactive peers, inactive peers that also polluting underdetermined gossip have little effect on availability and delay of the medium. The plots were formed from averaging 3 runs of the simulation initialized with different seeds. 95% confidence intervals are provided around each data point.

3.8.7 Simulation of Pollution with Decodable Gossip

Table 3.7: Pollution with Decodable Gossip Simulation Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIM.WARMUP.DECODES</td>
<td>50</td>
</tr>
<tr>
<td>SIM DURATION.INSERTS</td>
<td>75</td>
</tr>
<tr>
<td>SIM.NUM.PEERS</td>
<td>20</td>
</tr>
<tr>
<td>LOOKUP_PERCENT</td>
<td>0.25</td>
</tr>
<tr>
<td>CODE_SIZE</td>
<td>4</td>
</tr>
<tr>
<td>TTL.DECODE</td>
<td>30</td>
</tr>
<tr>
<td>TTL.GOSSIP</td>
<td>10</td>
</tr>
<tr>
<td>CONTRIBUTE_INTERVAL</td>
<td>20</td>
</tr>
</tbody>
</table>

This simulation investigates the effect of increasing numbers of inactive peers that also pollute with decodable gossip, as described in the Pollution with Decodable Gossip subsection of the Analysis of Attacks section, on availability and delay. See Figure 3.8.7 for results. Inactive peers polluting decodable gossip have a significant effect on the availability of the system. The plots were formed from averaging 3 runs of the simulation initialized with different seeds. 95% confidence intervals are
provided around each data point.

Figure 3-11: Pollution with Decodable Gossip Simulation

However, additional simulations show that increasing the CODE_SIZE parameter of the system can compensate for the effect of pollution with decodable gossip. Figure ?? presents the results of the same simulation, but with CODE_SIZE of 6, instead of 4. This shows that the system can be made resilient to multiple independent attackers polluting with decodeable gossip.
Chapter 4

Melting Pad

4.1 Introduction

The investigation into the security afforded by random linear network coding (RLNC) is ongoing [31] [39] [32] [19], but the systems for confidentiality built with RLNC to date concern protecting packet streams in an active session between sources and sinks. This work explores the use of RLNC to code data into a secure form suitable for wide dissemination or persistent storage. It achieves this with a basic application of interference alignment.

We introduce Melting Pad, an algebraic coding scheme that protects multiple data blocks while allowing for their separate decoding. It features two modes of secrecy: one in which protected blocks can be separately decoded by distinct secret keys, and one in which protected blocks can be separately decoded by a common secret key. The mode is chosen simply by the designation of whether the coefficients or a seed block are secret.

Related work includes protocols based on random linear network coding for security and their analysis. [39] presents SPOC, a RLNC protocol that provides data confidentiality by protecting the coefficients of linear combinations, and still allows intermediate nodes to perform network coding operations. [32] analyzes the confidentiality afforded by SPOC in more detail. [30] presents a secure RLNC protocol inspired by SPOC for wireless video streaming. [8] debuts interference alignment for
This chapter is outlined as follows: The Interference Alignment section introduces the basic concept of interference alignment with network coding. The Melting Pad Coding and Decoding section presents the Melting Pad encoding and decoding scheme. The Confidentiality section makes brief statements on the confidentiality of Melting Pad. The Applications section proposes several applications of the coding scheme. The Extensions section introduces several extensions to the scheme.

4.2 Interference Alignment

Interference alignment in linear network coding is the technique of coding packets with strategically introduced linear dependence, called interference, to enable more efficient decoding for a subset of the source packets.

Decoding one network coded source packet often requires solving a system that involves many linear combinations. For example, take \( x_1, x_2, \) and \( x_3 \) to be three vectors representing source packets, and \( \alpha_{ij} \) to be scalar coefficients chosen in forming linear combinations of the source packets. Three source packets can be encoded into linear combinations as

\[
\begin{align*}
L_1 &= \alpha_{1,1}x_1 + \alpha_{1,2}x_2 + \alpha_{1,3}x_3 \\
L_2 &= \alpha_{2,1}x_1 + \alpha_{2,2}x_2 + \alpha_{2,3}x_3 \\
L_3 &= \alpha_{3,1}x_1 + \alpha_{3,2}x_2 + \alpha_{3,3}x_3
\end{align*}
\]

\( x_1, x_2, \) and \( x_3 \) may be decoded from the three linear combinations by solving:

\[
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
= \begin{bmatrix}
\alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} \\
\alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} \\
\alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3}
\end{bmatrix}^{-1}
\begin{bmatrix}
L_1 \\
L_2 \\
L_3
\end{bmatrix}
\]

In the general case, decoding any one or all of the source packets \( x_1, x_2, \) and \( x_3 \) requires all three linear combinations \( L_1, L_2, \) and \( L_3. \)
Linear combinations of $x_1$, $x_2$, and $x_3$ may be formed in such a way so that $x_1$ and $x_2$ only require two linear combinations to decode, and $x_3$ requires three. For example, take $\alpha_i$, $\beta_i$, and $\gamma_i$ to be scalars, and form linear combinations as

\[
K_1 = \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 \\
K_2 = \beta_1 x_1 + \gamma_1(\alpha_2 x_2 + \alpha_3 x_3) \\
K_3 = \beta_2 x_2 + \gamma_2(\alpha_1 x_1 + \alpha_2 x_3)
\]

Then, $x_1$, $x_2$, and $x_3$ may be decoded with:

\[
x_1 = \left(\frac{K_1 - K_2}{\gamma_1}\right) \frac{1}{\alpha_1 + \beta_1/\gamma_1} \\
x_2 = \left(\frac{K_1 - K_3}{\gamma_2}\right) \frac{1}{\alpha_2 + \beta_2/\gamma_2} \\
x_3 = \frac{1}{\alpha_3} \left(\frac{\alpha_1(K_1 - K_2) - \alpha_2(K_1 - K_3)}{\alpha_1 \beta_1/\gamma_1} - \frac{\alpha_2(K_1 - K_3)}{\alpha_2 \beta_2/\gamma_2}\right)
\]

In this case, $x_1$ and $x_2$ each require two linear combinations for decoding instead of three, while $x_3$ requires three. It is said that the linear combinations $K_1$ and $K_2$ have the interference terms $\alpha_2 x_2 + \alpha_3 x_3$ aligned by $\gamma_1$. Likewise, $K_1$ and $K_3$ have the interference terms $\alpha_1 x_1 + \alpha_3 x_3$ aligned by $\gamma_2$.

### 4.3 Melting Pad Coding and Decoding

Take $x_1, ..., x_n$ to be $n$ input vectors of length $l$ representing source blocks, $\alpha_i$ and $\beta_i$ to scalar coefficients, and $m_0, ..., m_n$ to be $n + 1$ output vectors. All vector elements and scalars belong to a finite field $F_q$, with $q = 2^m$. $\alpha_i$ and $\beta_i$ are non-zero randomly selected, uniformly distributed scalars from $F_q$.

Melting Pad codes $n$ source blocks $x_1, ..., x_n$ into $n + 1$ protected blocks $m_0, ..., m_n$ in the following way:
\[ m_0 = \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \ldots + \alpha_n x_n \]

\[ m_1 = \beta_1 x_1 + \gamma_1 \left( \sum_{i \neq 1} \alpha_i x_i \right) \]

\[ m_2 = \beta_2 x_2 + \gamma_2 \left( \sum_{i \neq 2} \alpha_i x_i \right) \]

\[ \vdots \]

\[ m_n = \beta_n x_n + \gamma_n \left( \sum_{i \neq n} \alpha_i x_i \right) \]

The \( n \) source blocks \( x_1, \ldots, x_n \) can then be decoded with:

\[
\begin{align*}
  x_1 &= \left( m_0 - \frac{m_1}{\gamma_1} \right) \frac{1}{\alpha_1 + \beta_1/\gamma_1} \\
  x_2 &= \left( m_0 - \frac{m_2}{\gamma_2} \right) \frac{1}{\alpha_2 + \beta_2/\gamma_2} \\
  &\vdots \\
  x_n &= \left( m_0 - \frac{m_n}{\gamma_n} \right) \frac{1}{\alpha_n + \beta_n/\gamma_n}
\end{align*}
\]

### 4.3.1 Example

For example, if \( n = 3 \), then we can encode \( x_1 \), \( x_2 \), and \( x_3 \) with Melting Pad to:

\[
\begin{align*}
  m_0 &= \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 \\
  m_1 &= \beta_1 x_1 + \gamma_1 (\alpha_2 x_2 + \alpha_3 x_3) \\
  m_2 &= \beta_2 x_2 + \gamma_2 (\alpha_1 x_1 + \alpha_3 x_3) \\
  m_3 &= \beta_3 x_3 + \gamma_3 (\alpha_1 x_1 + \alpha_2 x_2)
\end{align*}
\]

and decode with:
\[
x_1 = \left( m_0 - \frac{m_1}{\gamma_1} \right) \frac{1}{\alpha_1 + \beta_1/\gamma_1}
\]
\[
x_2 = \left( m_0 - \frac{m_2}{\gamma_2} \right) \frac{1}{\alpha_2 + \beta_2/\gamma_2}
\]
\[
x_3 = \left( m_0 - \frac{m_3}{\gamma_3} \right) \frac{1}{\alpha_3 + \beta_3/\gamma_3}
\]

### 4.3.2 Explanation

The Melting Pad coding scheme forms block \( m_0 \), called the *seed block*, as a random linear combination of all source blocks. This seed block can be thought of as a "self-keying" pad made from the linearly combined sum of the source blocks. Each source block \( x_i \) is padded by this seed block to form a protected block \( m_i \). This is done by aligning all terms in the seed block, except the source block’s own \( \alpha_i x_i \), as interference under the scalar \( \gamma_i \). This interference alignment yields a 2x2 system for every pair of seed block and source block, summarized as:

\[
m_0 = \alpha_i x_i + P
\]
\[
m_i = \beta_i x_i + \gamma_i P
\]

where \( P \) is the “pad,” the linearly combined sum of all other source blocks. This system will be invertible, and therefore each protected block will be decodable, so long as \( \alpha_i, \beta_i, \) and \( \gamma_i \) are selected such that the determinant \( \alpha_i \gamma_i - \beta_i \) is non-zero. We assume that any construction of a Melting Pad will select random coefficients until this is true.

### 4.3.3 Security

The security in Melting Pad comes from designating the per source block coefficients \( \alpha_i, \beta_i, \) and \( \gamma_i \) to be secrets, designating the seed block to be secret, or both to be secret. The choice of which to designate as the secret depends on the application.
Below, we illustrate the possible designations of secrets with the previously presented \( n = 3 \) example.

If we choose to designate the per source block coefficients to be the secrets, then each block may be decoded separately with a unique key comprised of the \( \alpha_i, \beta_i, \) and \( \gamma_i \) coefficients:

<table>
<thead>
<tr>
<th>Ciphertext</th>
<th>Secret Key</th>
<th>Decoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_0 )</td>
<td>(-)</td>
<td>(-)</td>
</tr>
<tr>
<td>( m_1 )</td>
<td>( \alpha_1, \beta_1, \gamma_1 )</td>
<td>( f(m_0, m_1, \alpha_1, \beta_1, \gamma_1) \rightarrow x_1 )</td>
</tr>
<tr>
<td>( m_2 )</td>
<td>( \alpha_2, \beta_2, \gamma_2 )</td>
<td>( f(m_0, m_2, \alpha_2, \beta_2, \gamma_2) \rightarrow x_2 )</td>
</tr>
<tr>
<td>( m_3 )</td>
<td>( \alpha_3, \beta_3, \gamma_3 )</td>
<td>( f(m_0, m_3, \alpha_3, \beta_3, \gamma_3) \rightarrow x_3 )</td>
</tr>
</tbody>
</table>

If we choose to designate only the seed block to be secret, then all blocks can be decoded with a common key, the seed block:

<table>
<thead>
<tr>
<th>Ciphertext</th>
<th>Secret Key</th>
<th>Decoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-)</td>
<td>( m_0 )</td>
<td>(-)</td>
</tr>
<tr>
<td>( m_1, \alpha_1, \beta_1, \gamma_1 )</td>
<td>(-)</td>
<td>( f(m_0, m_1, \alpha_1, \beta_1, \gamma_1) \rightarrow x_1 )</td>
</tr>
<tr>
<td>( m_2, \alpha_2, \beta_2, \gamma_2 )</td>
<td>(-)</td>
<td>( f(m_0, m_2, \alpha_2, \beta_2, \gamma_2) \rightarrow x_2 )</td>
</tr>
<tr>
<td>( m_3, \alpha_3, \beta_3, \gamma_3 )</td>
<td>(-)</td>
<td>( f(m_0, m_3, \alpha_3, \beta_3, \gamma_3) \rightarrow x_3 )</td>
</tr>
</tbody>
</table>

We may also choose to designate both the seed block and per source block coefficients to be secret.

The size of the key, and therefore the absolute information theoretic security of the scheme, will differ between the two designations. Using the coefficients as the key results in many smaller keys, but allows for decoding the protected blocks separately. Using the seed block as the key results in a larger key, but can decode all protected blocks.

### 4.4 Confidentiality

This section presents basic statements on the confidentiality of source blocks when protected by the Melting Pad scheme.
The information theoretic security of Melting Pad depends on the statistical properties of its source data blocks. We first investigate the Maximal Security of Melting Pad, and then consider what kind of source blocks can achieve it.

4.4.1 Maximal Security

In this subsection, we show that the Melting Pad scheme has maximal security when the linear combinations it produces, \( m_1, \ldots, m_n \), are statistically independent and uniformly distributed, and their coefficients are drawn from a uniform distribution.

We first consider the case of designating the per source block coefficients to be the secret keys. In this case, an attacker has access to all \( m_1, \ldots, m_n \) linear combinations, and is attempting to decode \( x_i \), which is formed by

\[
x_i = \left( m_0 - \frac{m_i}{\gamma_i} \right) \frac{1}{\alpha_i + \beta_i/\gamma_i}
\]

The right multiplicand \( \frac{1}{\alpha_i + \beta_i/\gamma_i} \) amounts to a scalar, so this equation can be rewritten in terms of the two scalars \( \epsilon_1 \) and \( \epsilon_2 \):

\[
x_i = \left( m_0 - \frac{m_i}{\epsilon_1} \right) \epsilon_2
\]

Since the coefficients are drawn from a uniform distribution, the attacker has no information about the secret key. In addition, if \( m_0 \) and \( m_i \) are independent, they provide no information on \( x_i \). Therefore, the attacker must resort to guessing \( \epsilon_1 \) and \( \epsilon_2 \) with brute force enumeration to decode \( x_i \). These scalars live in the finite field \( \mathbb{F}_q \) with \( q = 2^m \) elements, so the effective search space for an attacker attempting brute force decoding is \( 2^{2m} \).

We now consider the case of designating the seed block to be the secret key. In this case, an attacker has access to \( m_1, \ldots, m_n \) linear combinations and the \( \alpha_i, \beta_i, \gamma_i \) coefficients, but not the \( m_0 \) linear combination. The attacker is attempting to decode \( x_i \) and knows \( m_i, \alpha_i, \beta_i, \) and \( \gamma_i \) used in its encoding.
If \( m_0 \) and \( m_i \) are independent, \( m_i \) provides no information on \( m_0 \) or \( x_i \). In addition, because \( m_0 \) is uniformly distributed, the attacker has no information about the secret key. The attacker is left with guessing \( m_i \), which is a vector of length \( l \) with elements in finite field \( \mathbb{F}_q \) with \( q = 2^m \). The effective search space for an attacker attempting brute force decoding is \( 2^{lm} \).

When Melting Pad exhibits maximal security, the search space and corresponding difficulty of brute force decoding a Melting Pad protected block can be made arbitrarily large. In the case of per source block coefficients used as secret keys, this is by the choice of the finite field size \( q = 2^m \). In the case of the seed block used as a secret key, it is by choice of source block length \( l \) and finite field size \( q = 2^m \).

### 4.4.2 Achievability of Maximal Security

The previous subsection showed that the Melting Pad linear combinations must be independent and uniformly distributed for maximal security. However, the statistical properties of these linear combinations depend on the statistical properties of their source blocks \( x_1, ..., x_n \).

First, we consider the requirement for uniformly distributed linear combinations. It is clear that linear combinations of uniformly distributed source blocks will be uniformly distributed, and that in the general case, the distribution of a linear combination is the convolution of its source block distributions. The best case for maximal security with Melting Pad is uniformly distributed source blocks, as this guarantees uniformly distributed linear combinations \( m_0, ..., m_n \).

Second, we consider the requirement for statistically independent linear combinations. This is a more difficult requirement to investigate, but we can establish its achievability in the case of uniformly distributed and statistically independent source blocks.

[23] presents a generalized version of the Skitovich-Darmois theorem for finite Abelian groups, which proves that if linear combinations of independent random variables with values in a finite Abelian group are independent, then the random variables must be uniformly distributed. While our question concerns the converse: if indepen-
dent, uniformly distributed source blocks will yield independent linear combinations,
the theorem proves that both properties can exist simultaneously.

The best case for maximal security in Melting Pad is uniformly distributed, inde-
pendent source blocks, but this is an unrealistic requirement for typical data. Future
work will investigate how linear combinations of source blocks with arbitrary distri-
butions and dependence and affect security, especially as the number of source blocks
increase.

4.5 Applications

In this section, we present two applications that are well-suited to the Melting Pad
coding scheme and demonstrate the utility of two of the secret designations described
in subsection Security. In the Subscriber Content Delivery application, a content
provider transmits content on a broadcast medium to all subscribers, but portions of
the content should only be accessible to its entitled subscribers. In the Secure Storage
on Untrusted Servers application, content is securely stored on multiple untrusted
servers.
4.5.1 Subscriber Content Delivery

Figure 4-1 illustrates a subscriber content delivery system. Such systems often use a downstream broadcast medium, which the Melting Pad scheme is well-suited for because of the separate decodability of protected blocks that it offers. This application of Melting Pad uses per source block coefficients as the secret key to each protected block, which enables fine-grain and agile access control to subscribers in the system. A content provider may deliver new secret keys to its subscribers through a back-channel, which immediately enable a subscriber’s decoding of the protected blocks associated with the delivered keys. Furthermore, a channel provider can regularly rotate secret keys to allow for a means of revoking channel access. Secret keys can be generated deterministically with a seeded secure pseudorandom number generator to limit the communication required for key assignment.

4.5.2 Secure Storage on Untrusted Servers

Figure 4-2 illustrates the secure storage of data blocks on multiple untrusted servers. This application of Melting Pad uses the seed block as the secret key to each of the protected blocks. The Melting Pad scheme enables the separate decoding of each protected block with one common key.
4.6 Extensions

In this section, we present extensions to the Melting Pad scheme that enable additional applications. “Grow” and “Shrink” allow for encoding an additional source block into an existing Melting Pad and removing a decoded source block from an existing Melting Pad, respectively, at the expense of publicizing the per source block \( \gamma_i \) coefficient. “Onion” presents a variation of Melting Pad scheme that features onion decoding, which automatically shrinks the Melting Pad on each decoded source block. “Onion” does not require publicizing any additional coefficients.

4.6.1 Grow

By publicizing the per source block coefficient \( \gamma_i \) with each protected block, a Melting Pad can be extended with a new source block \( x_{n+1} \). The extension procedure is:

1. Select non-zero random coefficients \( \alpha_{n+1}, \beta_{n+1}, \) and \( \gamma_{n+1} \)

2. Form the additional protected block \( m_{n+1} \)

3. Amend \( m_0 \) by adding the term \( \alpha_{n+1}x_{n+1} \)

4. Amend each \( m_i \) for \( i = 1, \ldots, n \) by adding the term \( \gamma_i \alpha_{n+1}x_{n+1} \)

which yields the amended Melting Pad,

\[
\begin{align*}
m_0' &= m_0 + \alpha_{n+1}x_{n+1} \\
m_i' &= m_i + \gamma_i\alpha_{n+1}x_{n+1} \\
&\vdots \\
m_n' &= m_n + \gamma_n\alpha_{n+1}x_{n+1} \\
m_{n+1}' &= \beta_{n+1}x_{n+1} + \gamma_{n+1}(m_0)
\end{align*}
\]
4.6.2 Shrink

By publicizing the per source block coefficient $\gamma_i$ with each protected block, a Melting Pad can be shrunk from $n$ to $n - 1$ protected blocks upon decoding $x_k$. The retraction procedure is:

1. Remove $m_k$

2. Amend $m_0$ by subtracting the term $\alpha_k x_k$

3. Amend each $m_i$ for $i \neq k$ by subtracting the term $\gamma_i \alpha_k x_k$

which yields the amended Melting Pad,

$$m_0' = m_0 - \alpha_k x_k$$
$$m_i' = m_i - \gamma_i \alpha_k x_k$$
$$\vdots$$
$$m_k = \text{Removed}$$
$$\vdots$$
$$m_n' = m_n - \gamma_n \alpha_k x_k$$

4.6.3 Onion

A slight variation on the Melting Pad coding scheme, called Melting Pad onion, constructs protected blocks that decode like an onion: decoding the outer $n$th block, removing it, and amending $m_0$, yields a Melting Pad onion containing $n - 1$ protected blocks. The decoding equation for $x_n$ remains unchanged from the original scheme, and this scheme does not require publicizing additional per source block coefficients.

The modified coding scheme codes blocks like this:
\[ \mathbf{m}_0 = \alpha_R \mathbf{R} + \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 + \alpha_3 \mathbf{x}_3 + \ldots \alpha_n \mathbf{x}_n \]
\[ \mathbf{m}_1 = \beta_1 \mathbf{x}_1 + \gamma_1 (\alpha_R \mathbf{R}) \]
\[ \mathbf{m}_2 = \beta_2 \mathbf{x}_2 + \gamma_2 (\alpha_R \mathbf{R} + \alpha_1 \mathbf{x}_1) \]
\[ \vdots \]
\[ \mathbf{m}_n = \beta_n \mathbf{x}_n + \gamma_n (\alpha_R \mathbf{R} + \sum_{i=1}^{n-1} \alpha_i \mathbf{x}_i) \]

The onion decoding procedure is:

1. Decode \( \mathbf{x}_n = \left( \mathbf{m}_0 - \frac{\mathbf{m}_n}{\gamma_n} \right) \frac{1}{\alpha_n + \beta_n / \gamma_n} \), as usual

2. Remove \( \mathbf{m}_n \)

3. Amend \( \mathbf{m}_0 \) by subtracting the term \( \alpha_k \mathbf{x}_n \)

which yields the amended Melting Pad onion,

\[ \mathbf{m}_0' = \mathbf{m}_0 - \alpha_k \mathbf{x}_n \]
\[ \mathbf{m}_i' = \mathbf{m}_i \]
\[ \vdots \]
\[ \mathbf{m}_{n-1}' = \mathbf{m}_{n-1} \]
\[ \mathbf{m}_n = \text{Removed} \]

Much like onion routing [35], this extension may be appropriate for a decentralized trust system. For example, an intermediate node that possesses the key to the outer most block receives the onion, decodes the outer block, subtrats it out, and then forwards the smaller onion to the next node, which has the key for the next outer block.
Chapter 5

Conclusion

5.1 NCGAB Conclusion and Future Work

In chapter 3, we motivated the issues in source rewriting anonymity systems with multiple hops and considered an alternate class of anonymity systems based on broadcast. This approach has previously been met with challenges in scaling a broadcast medium over unicast networks.

The technique of network coding was introduced, as well as its accompanying tools of network coded gossip and homomorphic hashing. We described the architecture of NCGAB, a decentralized peer-to-peer system based on network coded gossip that provides a resilient, anonymous broadcast medium. NCGAB is unique in that it does not require a cryptosystem, infrastructure of trust, or special nodes to operate, and has a simple implementation summarized on Figure 3.5.7. NCGAB’s anonymity, efficiency, and resiliency is owed to its core protocol based on network coded gossip.

In the Analysis of Anonymity section, we investigated the anonymity of participants using NCGAB, showing that it has unconditional receiver anonymity due to its broadcast design, and conjectured that its sender anonymity scales with size of the cooperative network. The Analysis of Attacks section comprehensively described the threats of active attackers attempting to compromise the system.

Simulations showed NCGAB to be highly scalable on a single set of system parameters, supporting networks ranging from 10 to 100 peers. Simulations also showed
that NCGAB was highly resilient to the attackers described in the Analysis of Attacks section.

Future work on NCGAB will simulate the effect of attackers with colluded decodable pollution, quantify NCGAB’s sender anonymity with more rigor, further explore the behavior of its system parameters, and investigate the practical implementation of a homomorphic hash function to ensure the integrity of gossip exchanged on the NCGAB overlay network.

5.2 Melting Pad Conclusion and Future Work

In chapter 4, we introduced Melting Pad, an algebraic coding scheme with information theoretic security properties, efficient decoding, and flexible modes of secrecy. Melting Pad codes $n$ source blocks into $n + 1$ protected blocks, which can be decoded individually, either each with separate, small keys or all with a common large key, depending on what information is designated secret. Melting Pad is derived from interference alignment concepts to achieve its property of efficient decodability.

Two potential applications of Melting Pad were presented: one for subscriber content delivery over a broadcast medium, which called for agile channel access control and was well suited to Melting Pad’s secret key per source block mode, and one for distributed untrusted storage, which mapped well to a Melting Pad’s common secret key mode.

We introduced extensions to the Melting Pad scheme that allow for its growth and shrinking, at the expense of publicizing one of the per source block coefficients, and an extension that enabled onion decoding.

We presented the preliminary results of an investigation into the confidentiality of Melting Pad, establishing the requirements for its maximal secrecy and the corresponding difficulty for an attacker to break it. We showed that when Melting Pad has maximal secrecy, an attacker must resort to a brute force search for the secret key. This search space can be made arbitrarily large by choice of the finite field size $q = 2^m$ in the case of per source block secret keys, or by a combination of choosing
the field size $q = 2^n$ and the source block vector length $l$ in the case of a common secret key.

Future work on Melting Pad will investigate the information theoretic guarantees of Melting Pad in the more general case of source blocks, especially arbitrarily distributed and statistically dependent ones. We suspect that the mixing property of random linear combinations may help to homogenize the distribution of the linear combinations as the number of source blocks grow, resulting in acceptable security.
Appendix A

ncgabsim Simulator Code

A.1 ncgabsim.py

```python
#!/usr/bin/python3

# NCGAB Simulator - Ivan A. Sergeev

import threading
import random
import json
import sys
import copy
import os
import queue

import ff

import termcolor

simConfig = 4

SimTemplate = {
    'NAME': '',
    'DESC': '',
    'LOOKUP_PERCENT': 0.26,
    'CODE_SIZE': 4,
    'TTL_DECODE': 30,
    'TTL_GOSSIP': 10,
    'SIM_NUM_PEERS': 0,
    'SIM_NUM_EVIL_PEERS': 0,
    'CONTRIBUTE_INTERVAL': 0,
    'SIM_WARMUP_DECODES': 50,
    'SIM_DURATION_INSERTS': 100,
}
```

if simConfig == 1:
    NumPeersSweep = [10, 25, 50, 100]
    ContributeIntervalSweep = [7.0, 10, 12.0, 15.0, 17.0, 20.0, 22.0]
    SeedSweep = [0x1, 0x2, 0x3]

    SimParamsList = []
    for numPeers in NumPeersSweep:
        for contributeInterval in ContributeIntervalSweep:
            for seed in SeedSweep:
                name = "Sim-Nd-Rd-S7d" % (numPeers, contributeInterval, seed)
                params = copy.deepcopy(SimTemplate)
                params["NAME"] = name
                params["SEED"] = seed
                params["SIM_NUM_PEERS"] = numPeers
                params["CONTRIBUTE_INTERVAL"] = contributeInterval
                SimParamsList.append(params)

if simConfig == 2:
    NumPeersSweep = [25]
    ContributeIntervalSweep = [7.0, 10, 12.0, 15.0, 17.0, 20.0, 22.0]
    SeedSweep = [0x1, 0x2, 0x3]

    SimParamsList = []
    for numPeers in NumPeersSweep:
        for contributeInterval in ContributeIntervalSweep:
            for seed in SeedSweep:
                name = "SimSelect-Nd-Rd-S7d" % (numPeers, contributeInterval, seed)
                params = copy.deepcopy(SimTemplate)
                params["NAME"] = name
                params["SEED"] = seed
                params["SIM_NUM_PEERS"] = numPeers
                params["CONTRIBUTE_INTERVAL"] = contributeInterval
                SimParamsList.append(params)

# Evil Peer Simulations

SimTemplate = {
    'NAME': "Evil-Base",
    'DESC': "",
    'LOOKUP_PERCENT': 0.25,
    'CODE_SIZE': 4,
    'TTL_DECODE': 30,
    'TTL_GOSSIP': 20,
    'SIM_NUM_PEERS': 0,
    'SIM_NUM_EVIL_PEERS': 0,
    'CONTRIBUTE_INTERVAL': 20,
'SIM_EVIL_PEER_TYPE': 'a',
'SIM_WARMUP_DECODES': 50,
'SIM_DURATION_INSERTS': 75,
'PRINT_LOG': False,
}

# Inactive, Underdetermined Gossip
if simConfig == 2:
    EvilPeerTypeSweep = ['inactive', 'underdetermined']
    NumPeersSweep = [10, 20, 30]
    pNumEvilPeersSweep = [0.0, 0.10, 0.20, 0.30]
    SeedSweep = [0x1, 0x2, 0x3]
    SimParamsList = []
    for evilPeerType in EvilPeerTypeSweep:
        for numPeers in NumPeersSweep:
            for pNumEvilPeers in pNumEvilPeersSweep:
                for seed in SeedSweep:
                    name = "Evil-Ys-N/d-E%0.2f-Sd" % (evilPeerType, numPeers, pNumEvilPeers, seed)
                    # Decodable with CODE_SIZE = 6
if simConfig == 4:
    EvilPeerTypeSweep = ['decodable']
    NumPeersSweep = [25]
    pNumEvilPeersSweep = [0.0, 0.10, 0.20, 0.30]
    SeedSweep = [0x1, 0x2, 0x3]
    SimParamsList = []
    for evilPeerType in EvilPeerTypeSweep:
        for numPeers in NumPeersSweep:
            for seed in SeedSweep:
                name = "Evil-Ys-CS4-X%0.2f-%d" % (evilPeerType, numPeers, seed)
for pNumEvilPeers in pNumEvilPeersSweep:
    for seed in SeedSweep:
        name = "Evil-%s-CS6-Nd-E%0.2f-Sd" % (evilPeerType, numPeers, pNumEvilPeers, seed)
        params = copy.deepcopy(SimTemplate)
        params['NAME'] = name
        params['SEED'] = seed
        params['SIMNUMPEERS'] = numPeers
        params['SIMNUM_EVIL_PEERS'] = int(pNumEvilPeers * params['SIMNUMPEERS'])
        params['SIM_EVIL_PEER_TYPE'] = evilPeerType
        params['CODE_SIZE'] = 6
        SimParamsList.append(params)

# Colluded Decodable with CODE_SIZE = 8
if simConfig == 5:
    EvilPeerTypeSweep = ['colluded']
    NumPeersSweep = [20]
    pNumEvilPeersSweep = [0.0, 0.10, 0.20, 0.30]
    SeedSweep = [0x1, 0x2, 0x3]
    SimParamsList = []
    for evilPeerType in EvilPeerTypeSweep:
        for numPeers in NumPeersSweep:
            for pNumEvilPeers in pNumEvilPeersSweep:
                for seed in SeedSweep:
                    name = "Evil-%s-CS6-Nd-EO.2f-Sd" % (evilPeerType, numPeers, pNumEvilPeers, seed)
                    params = copy.deepcopy(SimTemplate)
                    params['NAME'] = name
                    params['SEED'] = seed
                    params['SIMNUMPEERS'] = numPeers
                    params['SIMNUM_EVIL_PEERS'] = int(pNumEvilPeers * params['SIMNUMPEERS'])
                    params['SIM_EVIL_PEER_TYPE'] = evilPeerType
                    params['CODE_SIZE'] = 8
                    SimParamsList.append(params)

class Stats:
    def __init__(self, simParams, simEventStop):
        # Statistics collected
        self.message_inserts = []
        self.message_decodes = {}
        self.matrix_reduces = []
        self.window_sizes = []
        self.last_message_exists = {}
        self.round_finished = 0

        # Simulation parameters
        self.simParams = simParams
        self.simEventStop = simEventStop

    def message_track(self, rnd, decoded_objects, gossip_objects):
        if len(self.message_decodes) < self.simParams['SIM_WARMUP_DECODES']:
            return
        if len(self.message_inserts) < self.simParams['SIM_DURATION_INSERTS']:
            return

        # If the last message disappeared in the previous round, signal
        # simulation stop

    def _init__ (self, simParams, simEventStop):
if rnd > 0 and (rnd - 1) in self.last_message_exists:
    if self.last_message_exists[rnd - 1] == False:
        self.simEventStop.set()
        return

# If we already saw the last message in this round
if rnd in self.last_message_exists and self.last_message_exists[rnd] == False:
    return

# Look for all messages inserted inside peers on the network
for (_., last_pid) in self.message_inserts:
    # Look for the inserted message in the decoded objects
    for p in decoded_objects:
        if str(p) == last_pid:
            self.last_message_exists[rnd] = True
            return

    # Look for the inserted message in the gossip objects
    for lc in gossip_objects:
        for p in lc.messages:
            if str(p) == lastpid:
                self.last_message_exists[rnd] = True
                return

self.last_message_exists[rnd] = False

def message_insert(self, rnd, nid, pid):
    if len(self.message_decodes) < self.simParams['SIM_WARMUP_DECODES']:
        return

    # If we have already collected our simulation number of inserts
    if len(self.message_inserts) >= self.simParams['SIM_DURATION_INSERTS']:
        return

    self.message_inserts.append( (rnd, nid, pid) )

def message_decode(self, rnd, nid, pid):
    # If we hadn’t started recording this PID yet
    if pid not in self.message_decodes:
        self.message_decodes[pid] = []

    # If this nid already decoded this pid in the past
    if nid in [ n for (_, n) in self.message_decodes[pid] ]:
        return

    self.message_decodes[pid].append( (rnd, nid) )

def matrix_reduce(self, rnd, nid, numrows, numcols, numsolved):
    self.matrix_reduction. append( (rnd, nid, numrows, numcols, numsolved) )

def window_size(self, rnd, nid, solved_size, gossip_size):
    self.window_sizes. append( (rnd, nid, solved_size, gossip_size) )

def finished(self, rnd):
    self.round_finished = rnd

# -----------------------------

def dump(self):
    i = 0
    while True:
        path = "data/%s-ld.data" % (self.simParams['NAME'], i)
if not os.path.exists(path): break
i += 1

f = open(path, "w")
data = {'simParams': self.simParams,
       'message_inserts': self.message_inserts,
       'message_decodes': self.message_decodes,
       'matrix_reduces': self.matrix_reduces,
       'window_sizes': self.window_sizes,
       'round_finished': self.round_finished}
f.write(json.dumps(data))
f.close()
return path

class Log:
    def __init__(self, simParams):
        self.elog = []
        self.simParams = simParams

    def log(self, rnd, etype, enid, emsg):
        e = {}
        e['time'] = rnd
        e['type'] = etype
        e['nid'] = enid
        e['msg'] = emsg
        ejson = json.dumps(e)
        self.elog.append(ejson)

        if self.simParams['PRINT-LOG']:
            if etype == "join" or etype == "leave": print(termcolor.colored(ejson, "yellow"))
            elif etype == "receive": print(termcolor.colored(ejson, "yellow"))
            elif etype == "insert": print(termcolor.colored(ejson, "red"))
            elif etype == "reduce": print(termcolor.colored(ejson, "blue"))
            elif etype == "decode": print(termcolor.colored(ejson, "green"))
            else: print(termcolor.colored(ejson, "white"))

    def dump(self):
        i = 0
        while True:
            path = "logs/%s-X%d.log" % (self.simParams['NAME'], i)
            if not os.path.exists(path): break
            i += 1

        f = open(path, "w")
        for e in self.elog:
            f.write(e + "\n")
        f.close()
        return path

class Network:
    def __init__(self, simlog, simstats):
        self.simlog = simlog
        self.simstats = simstats
        self.network = {}
def join(self, rnd, nid, q):
    self.network[nid] = q
    self.simlog.log(rnd, "join", nid, "")
    self.simlog.log(rnd, "leave", nid, "")

def leave(self, rnd, nid):
    if nid in self.network:
        del self.network[nid]
        self.simlog.log(rnd, "leave", nid, "")

def lookup_random(self, nid, n):
    choices = list(self.network.keys())
    choices.remove(nid)
    random.shuffle(choices)
    return [self.network[choices[i]] for i in range(n)]

class Message:
    def __init__(self, pid):
        self.pid = pid

    def __str__(self):
        return self.pid

    def __eq__(self, other):
        if (self.pid == other.pid):
            return True
        return False

class DummyMessage(Message):
    def __init__(self):
        Message.__init__(self, "R\,\,04x\% random.getrandbits(32))

class RealMessage(Message):
    def __init__(self, nid):
        Message.__init__(self, "M\,\,04x\% (nid, random.getrandbits(32))")

class EvilMessage(Message):
    def __init__(self, nid):
        Message.__init__(self, "E\,\,04x\% (nid, random.getrandbits(32))")

class RLC(Message):
    def __init__(self, messages):
        coefs = [random.randint(0, 255) for p in messages]
        pid = "LC/\,\,02x\% (\,\,\,\,\,\,\,\,",".join([p.pid for p in messages]) + ")" + ",",.join(["%02x\% c for c in coefs") + ")"
        Message.__init__(self, pid)
        self.messages = messages
        self.coefs = coefs

# choose weighted random objects, scores:
scores_cdf = []
for s in scores[:]:
    scores_cdf.append(sum(scores_cdf) + s)
random_score = random.random()*scores_cdf[-1]
for index in range(len(scores_cdf)):
    if scores_cdf[index] > random_score:
        break
class Window:
    def __init__(self, keep_expired=False):
        self.window = []
        self.ttl = {}
        self.keep_expired = keep_expired

    def add(self, p, ttl):
        if type(p) == type(None) or p in self.window:
            return False

        self.window.append(p)
        self.ttl[str(p)] = ttl
        return True

    def prune(self):
        if self.keep_expired:
            return

        expiredList = list(filter(lambda x: self.ttl[str(x)] == 0, self.window))

        for e in expiredList:
            self.window.remove(e)
            del self.ttl[str(e)]

        return expiredList

    def tick(self):
        # Decrement the TTL for each object in the window
        for x in self.window:
            self.ttl[str(x)] = max(0, self.ttl[str(x)] - 1)

        return self.prune()

    def live_objects(self):
        return list(filter(lambda x: self.ttl[str(x)] > 0, self.window))

    def objects(self):
        return self.window[:]

    def __str__(self):
        s = "Window"
        for x in self.window:
            s += "	" + str(x) + " TTL: " + str(self.ttl[str(x)]) + "\n"
        return s

class Decoded_Window(Window):
    def __init__(self):
        Window.__init__(self, keep_expired=True)

    def choose_random_uniform(self, n):
        choices = self.live_objects()
        random.shuffle(choices)
        return choices[0 : min(n, len(choices))]

    def choose_random(self, n):
        choices = self.live_objects()
scores = [self.ttl[str(c)] for c in choices]
n = min(n, len(choices))
chosen = []
for i in range(n):
k = choose_weighted_random(choices, scores)
chosen.append(choices[k])
delete choices[k]
delete scores[k]
return chosen

class GossipWindow(Window):
    def __init__(self):
        Window.__init__(self, keep-expired=False)
        # Keep active objects in a dictionary by source as well
        self.active-objects-by-source = {}

def add(self, src, p, ttl):
    if Window.add(self, p, ttl) == True:
        # Create a new list for the source if it's not in our dictionary
        if src not in self.active-objects-by-source:
            self.active-objects-by-source[src] = []
        # Add the message to the source's list
        self.active-objects-by-source[src].append(p)
        return True
    return False

def prune(self):
    expiredList = Window.prune(self)
    # For each expired object
    for e in expiredList:
        # If the object exists in this source's list
        if e in self.active-objects-by-source[src]:
            # Delete it
            self.active-objects-by-source[src].remove(e)
            # Delete the source's list if it's empty now
            if len(self.active-objects-by-source[src]) == 0:
                del self.active-objects-by-source[src]
        break
    return expiredList

def choose_random_uniform(self):
    # Choose a random source
    source = random.choice(list(self.active-objects-by-source.keys()))
    # Gather a list of objects by this source
    objects = self.active-objects-by-source[source]
    return random.choice(objects)

def choose_random(self):
    # Choose a random source
    source = random.choice(list(self.active-objects-by-source.keys()))
    # Gather a list of objects by this source
    choices = self.active-objects-by-source[source]
    scores = [self.ttl[str(c)] for c in choices]
```python
return choices[choose_weighted_random(choices, scores)]

def fast_rref(self, m, b, x):
    done = False
    pi = 0

    # Iterate through each row
    for j in range(len(m)):
        # While we do not have a pivot for this row
        while m[j][pi] == 0:
            # Find a row below to swap with for a pivot at pi
            for k in range(j+1, len(m)):
                if m[k][pi] != 0:
                    # Swap with this row
                    (m[j], m[k]) = (m[k], m[j])
                    (b[j], b[k]) = (b[k], b[j])
                    break

            # Increment pivot index if we could not find a row to swap with
            if m[j][pi] == 0:
                pi += 1

        # If there is no pivots left, we're done reducing
        if pi == len(m[0]):
            done = True
            break

    if done:
        break

    # Divide through to have a pivot of 1
    m[j] = [ff.FiniteFieldArray.ff_elem_div(m[j][i], m[j][pi]) for i in range(len(m[0]))]

    # Eliminate above & below
    for k in range(len(m)):
        if k != j and m[k][pi] != 0:
            m[k] = [ff.FiniteFieldArray.ff_elem_sub(m[k][i],
                ff.FiniteFieldArray.ff_elem_mul(m[j][i], m[k][pi])) for i in range(len(m[0]))]

    # Move onto the next pivot
    pi += 1

    # If there is no pivots left, we're done reducing
    if pi == len(m[0]):
        break

    solved = []
    used = []

    for i in range(len(m)):
        # If this row has only one non-zero entry
        reduced_coeffs = [1*(m[i][j] != 0) for j in range(len(m[0]))]
        if sum(reduced_coeffs) == 1:
            # Add the solution to our solved list
            solved.append(x[reduced_coeffs.index(1)])
            # Add the decoded LC to our used list
            used.append(b[i])

    return (solved, used)

def solve(self, decoded_window):
```

unsolved_message_map = {}

# Get a list of decoded messages
decoded_messages = decoded_window.objects()

# Make a list of undecoded linear combinations
undecoded_lc = []
for lc in self.live_objects():
    # If we've decoded this entire linear combination, don't add it
    if sum([p in decoded_messages for p in lc.messages]) == len(lc.messages):
        continue
    undecoded_lc.append(lc)

# Put together a list of all decoded messages referenced by undecoded
# linear combinations
ref_decoded_messages = []
for lc in undecoded_lc:
    for p in lc.messages:
        if p in decoded_messages and p not in ref_decoded_messages:
            ref_decoded_messages.append(p)

# Assemble x of mx=b
x = {}
# x maps messages to column indices
for p in ref_decoded_messages:
    x[str(p)] = len(x)

# Assign a col index to each referenced decoded message
for p in ref_decoded_messages:
    x[str(p)] = len(x)

# Assign a col index to each undecoded message
for lc in undecoded_lc:
    for p in lc.messages:
        if str(p) not in x:
            x[str(p)] = len(x)
            unsolved_message_map[str(p)] = p

# Assemble n and b of mx = b
m = []
b = []

# Create a row for each solved message
for p in ref_decoded_messages:
    r = [0] * len(x)
    r[x[str(p)]] = 1
    m.append(r)
    b.append(p)

# Create a row for each linear combination
for lc in undecoded_lc:
    r = [0] * len(x)
    for i in range(len(lc.messages)):
        pid = str(lc.messages[i])
        coef = lc.coefs[i]
        r[x[pid]] = coef
        m.append(r)
        b.append(lc)

# Information about the size of this reduce attempt
b_pids = [str(p) for p in b]
numrows = len(m)
numcols = len(m[0]) if numrows > 0 else 0
(solved, used) = self.fast_ref(m, b, sorted(x, key=x.get))

# Remove previously decoded objects from our solution
for p in ref_decoded_messages:
    if str(p) in solved:
        solved.remove(str(p))
    if p in used:
        used.remove(p)

# Map message ids to message objects in our solution
for i in range(len(solved)):
    solved[i] = unsolved_message_map[solved[i]]

# Information about the solution of this reduce attempt
s_pids = [str(p) for p in solved]
return (m_numrows, m_numcols, b_pids, s_pids, solved)

# Map message ids to message objects in our solution
for i in range(len(solved)):
    solved[i] = unsolved_message_map[solved[i]]

# Information about the solution of this reduce attempt
s_pids = [str(p) for p in solved]
return (m_numrows, m_numcols, b_pids, s_pids, solved)

class EvilPeer():
    def __init__(self, nid, network, simlog, simstats, simParams):
        # Our unique peer ID
        self.nid = nid
        # Network, Log, Stats handles
        self.network = network
        self.simlog = simlog
        self.simstats = simstats
        # Simulation parameters
        self.simParams = simParams

        # Create a solved message window
        self.decoded_window = DecodedWindow()

        # Create an input queue
        self.queue = queue.Queue()

        # Join the network
        self.network.join(0, nid, self.queue)

class EvilPeer_Inactive(EvilPeer):
    def simulate(self, rnd):
        # Process all received gossip
        while not self.queue.empty():
            try: (src, gossip) = self.queue.get(False)
            except queue.Empty: break
            # Throw it away...

class EvilPeer_Underdetermined(EvilPeer):
    def simulate(self, rnd):
        # Process all received gossip
        while not self.queue.empty():
            try: (src, gossip) = self.queue.get(False)
            except queue.Empty: break
            # Throw it away...

        # Send our own gossip
        dests = self.network.lookup_random(self.nid, self.simParams['SIM_NUM_PEERS'] - 1)
        for d in dests:
            # Code some gossip based on our Decoded Window
            gossip = []
            # Code an RLC of new messages
gossip.append(RLC([EvilMessage(self.nid) for i in range(self.simParams['CODE_SIZE'])]))

# Transmit to the destination
d.put((self.nid, gossip))

class EvilPeer_Decodable(EvilPeer):
    def simulate(self, rnd):
        # Update the TTLs of our object windows
        self.decoded_window.tick()

        # Keep our decoded window filled with our own messages
        for i in range(self.simParams['CODE_SIZE'] - len(self.decoded_window.live_objects())):
            self.decoded_window.add(EvilMessage(self.nid), self.simParams['CODE_SIZE'])

        # Process all received gossip
        while not self.queue.empty():
            try: (src, gossip) = self.queue.get(False)
            except queue.Empty: break

            # Throw it away...

        # Send our own gossip
        dests = self.network.lookup_random(self.nid, self.simParams['SIM_NUM_PEERS'] - 1)
        for d in dests:
            # Code some gossip based on our Decoded Window
            gossip = []

            # Send RLCs of our current decoded window
            gossip.append(RLC(self.decoded_window.choose_random(self.simParams['CODE_SIZE'])))

            # Transmit to the destination
            d.put((self.nid, gossip))

class Peer:
    def __init__(self, nid, network, simlog, simstats, simParams):
        # Our unique peer ID
        self.nid = nid

        # Network, Log, Stats handles
        self.network = network
        self.simlog = simlog
        self.simstats = simstats

        # Simulation parameters
        self.simParams = simParams

        # Create an input queue
        self.queue = queue.Queue()

        # Create a solved message window
        self.decoded_window = DecodedWindow()

        # Create a gossip window
        self.gossip_window = Gossip_Window()

        # Initialize our window with dummy messages
        for i in range(self.simParams['CODE_SIZE']):
            self.decoded_window.add(DummyMessage(), self.simParams['TTL_DECODE'])

        # Choose a random insert message timeout
        self.insert_message_timeout = int(random.randint(0, int(self.simParams['CONTRIBUTE_INTERVAL'])))

        # Join the network
        self.network.join(0, nid, self.queue)
def simulate(self, rnd):
    # Update the TTLs of our object windows
    self.decoded_window.tick()
    self.decoded_window.prune()
    self.gossip_window.tick()
    self.gossip_window.prune()
    # Keep track of the last message disappearing
    self.simstats.message_track(rnd, self.decoded_window.live_objects(), self.gossip_window.live_objects())
    # Log our window sizes
    self.simstats.window_size(rnd, self.nid, len(self.decoded_window.objects()), len(self.gossip_window.objects()))
    # Update our insert message timeout counter
    if self.insert_message_timeout > 0:
        self.insert_message_timeout -= 1
        # Introduce a new message if our random insert timeout expired, but only
        # if we've have a Decoded Window of at least CODESIZE
        if self.insert_message_timeout == 0:
            p = RealMessage(self.nid)
            # Add a real message to our decoded window
            if self.decoded_window.add(p, self.simParams['TTL_DECODE']):
                # Log the insert
                self.simlog.log(rnd, "insert", self.nid, str(p))
                self.simstats.message_insert(rnd, self.nid, str(p))
            # Choose a new insert message timeout
            self.insert_message_timeout = int(random.expovariate(1.0 / float(self.simParams['CONTRIBUTE_INTERVAL'])))
            self.insert_message_timeout = int(self.simParams['CONTRIBUTE_INTERVAL'])
    # Process all received gossip
    while not self.queue.empty():
        try: (src, gossip) = self.queue.get(False)
        except queue.Empty: break
        # Add the linear combinations to our gossip window
        for lc in gossip:
            self.gossip_window.add(src, lc, self.simParams['TTL_Gossip'])
        # Log the receive
        self.simlog.log(rnd, "receive", self.nid, "src: %d, gossip: %s" % (src, str([str(p) for p in gossip])))
    # Try to solve some gossip
    (m_numrows, m_numcols, b_pids, s_pids, solved) = self.gossip_window.solve(self.decoded_window)
    # Log the reduce attempt
    self.simstats.matrix_reduce(rnd, self.nid, m_numrows, m_numcols, len(solved))
    self.simlog.log(rnd, "reduce_info", self.nid, "%s to %s" % (str(b_pids), str(s_pids)))
    # Add the decoded messages to our Decoded Window
    for p in solved:
        if self.decoded_window.add(p, self.simParams['TTL_DECODE']):
            # Log the decodes
            self.simlog.log(rnd, "decode", self.nid, str(p))
            if isinstance(p, RealMessage):
                self.simstats.message_decode(rnd, self.nid, str(p))
    # Fill up Decoded Window with dummy messages if it is short
    for i in range(self.simParams['CODE_SIZE'] - len(self.decoded_window.live_objects())):
        self.decoded_window.add(DummyMessage(), self.simParams['TTL_DECODE'])
# Look up LOOKUP_PERCENT subset of peers on the network
dests = self.network.lookup_random(self.nid, int(self.simParams['LOOKUP_PERCENT'])*self.simParams['SIM_NUM_PEERS'])
for d in dests:
    gossip = []

    if random.getrandbits(1):
        # Code new-gossip from our Decoded Window
        gossip.append(RLC(self.decoded_window.choose_random(self.simParams['CODE_SIZE'])))
    else:
        # Choose re-gossip from our Gossip Window
        if len(self.gossip_window.live_objects()) > 0:
            gossip.append(self.gossip_window.choose_random())

    # Transmit to the destination
    d.put((self.nid, gossip))

if __name__ == '__main__':
    ff.FiniteFieldArray.ff_precompute()
    for si in range(len(SimParamsList)):
        simParams = SimParamsList[si]
        # If we have already completed this simulation, skip it
        if os.path.exists("data/Xs-0.data %s" % simParams['NAME']):
            continue
        random.seed(simParams['SEED'])

        # Simulation stop event set by simStats
        simEventStop = threading.Event()

        # Simulation objects
        simStats = Stats(simParams, simEventStop)
        simLog = Log(simParams)
        simNetwork = Network(simLog, simStats)

        simPeers = []
        # Add cooperative peers to the network
        for i in range(simParams['SIM_NUM_PEERS'] - simParams['SIM_NUM_EVIL_PEERS']):
            simPeers.append(Peer(i, simNetwork, simLog, simStats, simParams))

        # Add evil peers to the network
        for i in range(simParams['SIM_NUM_EVIL_PEERS']):
            if simParams['SIM_EVIL_PEER_TYPE'] == "inactive":
                evilPeer = EvilPeer_Inactive
            elif simParams['SIM_EVIL_PEER_TYPE'] == "underdetermined":
                evilPeer = EvilPeer_Underdetermined
            elif simParams['SIM_EVIL_PEER_TYPE'] == "decodable":
                evilPeer = EvilPeer_Decodable
            else:
                evilPeer = EvilPeer_Inactive

            simPeers.append(evilPeer(i + (simParams['SIM_NUM_PEERS'] - simParams['SIM_NUM_EVIL_PEERS']) - i, simNetwork, simLog, simStats, simParams))

        print("Starting simulation %d / %d: %s" % (si+1, len(SimParamsList), simParams['NAME']))

        roundCount = 0
        while True:
            # Simulate the peers in a different order each round
            random.shuffle(simPeers)
for n in simPeers:
    n.simulate(roundCount)

sys.stdout.write("\r%d. %d -- Round %d\n")
(len(simStats.message_inserts), len(simStats.message_decodes), roundCount+1))

# Stop the simulation if we've collected enough data
if simEventStop.is_set():
    break

roundCount += 1

# Log the finish at this round count
simStats.finished(roundCount)
simLog.log(roundCount, "finish", 0, '')

print()

# Dump stats
print("Wrote stats to %s" % simStats.dumpo)

# Dump log
print("Wrote log to %s" % simLog.dumpo)

---

A.2 ff.py

# Finite Field Arithmetic Implementation

class FiniteFieldArray():
    ### Predefined finite field characteristics
    FIELD_SIZE = 8
    PRIMITIVE_POLY = 0x1B
    GENERATOR = 0x03

    ELEMENT_NUM = 0
    ELEMENT_MASK = 0
    ff_exp_table = []
    ff_log_table = []
    ff_sqrt_table = []
    ff_mul_table = {}
    ff_div_table = {}

    @staticmethod
    def ff_precompute():
        # Setup constants associated with this field size
        FiniteFieldArray.ELEMENT_NUM = 2**FiniteFieldArray.FIELD_SIZE
        FiniteFieldArray.ELEMENT_MASK = FiniteFieldArray.ELEMENT_NUM-1
        MSB_MASK = (FiniteFieldArray.ELEMENT_MASK >> 1)+1

        # Slow finite field multiplication used to precompute exponent
        # and logarithm tables
        def ff_slow_mul(a, b):
35     p = 0
36     for i in range(FiniteFieldArray.FIELD_SIZE):
37         if (b & 0x1):
38             p ^= a
39     if (a & MSB_MASK):
40         a <<= 1
41         a &= FiniteFieldArray.ELEMENT_MASK
42     else:
43         a <<= 1
44     a &= FiniteFieldArray.ELEMENT_MASK
45     b >>= 0x1
46     return p

# Precompute exponent and logarithm table
FiniteFieldArray.ff_exp_table = [0] * FiniteFieldArray.ELEMENT_NUM
FiniteFieldArray.ff_log_table = [0] * FiniteFieldArray.ELEMENT_NUM
FiniteFieldArray.ff_exp_table[0] = 0x01
for i in range(1, FiniteFieldArray.ELEMENT_NUM):
    FiniteFieldArray.ff_exp_table[i] = ffslowmul(FiniteFieldArray.ff_exp_table[i-1], FiniteFieldArray.GENERATOR)
    FiniteFieldArray.ff_log_table[FiniteFieldArray.ff_exp_table[i]] = i

# Slow finite field square root used to precompute square root table
def ff_slow_sqrt(x):
    for i in range(FiniteFieldArray.ELEMENT_NUM):
        if FiniteFieldArray.ff_elem_mul_slow(i, i) == x:
            return i
    raise ValueError("Xd has no squareroot in (2**%d, Ox%02X, Ox%02X)!
\% (x, FiniteFieldArray.PRIMITIVE_POLY, FiniteFieldArray.GENERATOR))

# Precompute square root table
FiniteFieldArray.ff_sqrt_table = [0] * FiniteFieldArray.ELEMENT_NUM
for i in range(FiniteFieldArray.ELEMENT_NUM):
    try:
        FiniteFieldArray.ff_sqrt_table[i] = ff_slow_sqrt(i)
    except ValueError:
        FiniteFieldArray.ff_sqrt_table[i] = None

# Precompute multiplication table
for a in range(FiniteFieldArray.ELEMENT_NUM):
    FiniteFieldArray.ff_mul_table[a] = {}
    for b in range(FiniteFieldArray.ELEMENT_NUM):
        FiniteFieldArray.ff_mul_table[a][b] = FiniteFieldArray.ff_elem_mul_slow(a, b)

# Precompute division table
for a in range(FiniteFieldArray.ELEMENT_NUM):
    FiniteFieldArray.ff_div_table[a] = {}
    for b in range(FiniteFieldArray.ELEMENT_NUM-1):
        FiniteFieldArray.ff_div_table[a][b+1] = FiniteFieldArray.ff_elem_div_slow(a, b+1)

# Element-wise finite field addition, subtraction, multiplication, and division
@staticmethod
def ff_elem_add(a, b):
    return a - b
@staticmethod
def ff_elem_sub(a, b):
    return a - b

@staticmethod
def ff_elem_mul(a, b):
    if a == 0 or b == 0: return 0
    # a * b = exp(log(a) * log(b)) = exp(log(a) + log(b))
    exp_index = FiniteFieldArray.ff_log_table[a] + FiniteFieldArray.ff_log_table[b]
    exp_index %= FiniteFieldArray.ELEMENT_MASK
    return FiniteFieldArray.ff_exp_table[exp_index]

@staticmethod
def ff_elem_mul_s(a, b):
    if a == 0 or b == 0:
        return 0
    # a * b = exp(log(a) * log(b)) = exp(log(a) + log(b))
    exp_index = FiniteFieldArray.ff_log_table[a] + FiniteFieldArray.ff_log_table[b]
    exp_index %= FiniteFieldArray.ELEMENT_MASK
    return FiniteFieldArray.ff_exp_table[exp_index]

@staticmethod
def ff_elem_div(a, b):
    if b == 0:
        raise ZeroDivisionError

    a_div_b = a * (1/b)
    b_inverse = FiniteFieldArray.ff_exp_table[FiniteFieldArray.ELEMENT_MASK - FiniteFieldArray.ff_log_table[b]]
    return FiniteFieldArray.ff_elem_mul(a, b_inverse)

@staticmethod
def ff_elem_div_s(a, b):
    if b == 0:
        raise ZeroDivisionError
    return FiniteFieldArray.ff_div_table[a][b]

@staticmethod
def ff_elem_sqrt(a):
    return FiniteFieldArray.ff_sqrt_table[a]

########################################################################

@staticmethod
def ff_bytearray_to_elemarray(x):
    if FiniteFieldArray.FIELD_SIZE != 8:
        raise ValueError("ff-bytearray-to-elemarray unimplemented for FF 2**%d\n" % FiniteFieldArray.FIELD_SIZE)
    return list(x)

@staticmethod
def ff_elemarray_to_bytearray(x):
    if FiniteFieldArray.FIELD_SIZE != 8:
        raise ValueError("ff_elemarray_to_bytearray unimplemented for FF 2**%d\n" % FiniteFieldArray.FIELD_SIZE)
    return bytearray(x)

########################################################################

    def __init__(self, x = None):
        self.raw_bytes = bytearray([])
        if x is not None:
            raw_bytes = bytearray(x)
            if len(raw_bytes) % (FiniteFieldArray.FIELD_SIZE/8) != 0:
                raise ValueError("invalid byte array size for FF 2**%d % FiniteFieldArray.FIELD_SIZE")
            self.raw_bytes = raw_bytes

    def __str__(self):
        return str(self.raw_bytes)

    def __repr__(self):
        return str(self.raw_bytes)

    def __eq__(self, other):
        return self.raw_bytes == other.raw_bytes

    def __hash__(self):
        return hash(self.raw_bytes)

    def __add__(self, other):
        if isinstance(other, int):
            return FiniteFieldArray.ff_elemarray_add(self, other)
        else:
            return FiniteFieldArray.ff_elemarray_add(self, other.raw_bytes)

    def __sub__(self, other):
        if isinstance(other, int):
            return FiniteFieldArray.ff_elemarray_sub(self, other)
        else:
            return FiniteFieldArray.ff_elemarray_sub(self, other.raw_bytes)

    def __mul__(self, other):
        if isinstance(other, int):
            return FiniteFieldArray.ff_elemarray_mul(self, other)
        else:
            return FiniteFieldArray.ff_elemarray_mul(self, other.raw_bytes)

    def __truediv__(self, other):
        if isinstance(other, int):
            return FiniteFieldArray.ff_elemarray_div(self, other)
        else:
            return FiniteFieldArray.ff_elemarray_div(self, other.raw_bytes)

    def __pow__(self, other):
        if isinstance(other, int):
            return FiniteFieldArray.ff_elemarray_pow(self, other)
        else:
            return FiniteFieldArray.ff_elemarray_pow(self, other.raw_bytes)

    def __neg__(self):
        return FiniteFieldArray.ff_elemarray_neg(self.raw_bytes)

    def __invert__(self):
        return FiniteFieldArray.ff_elemarray_invert(self.raw_bytes)

    def __lt__(self, other):
        return self.raw_bytes < other.raw_bytes

    def __le__(self, other):
        return self.raw_bytes <= other.raw_bytes

    def __gt__(self, other):
        return self.raw_bytes > other.raw_bytes

    def __ge__(self, other):
        return self.raw_bytes >= other.raw_bytes

    def __len__(self):
        return len(self.raw_bytes)

    def __len__(self):
### representation

```python
# FiniteFieldArray[i] = scalar
def __getitem__(self, index):
    return self.raw_bytes[index]

# len(FiniteFieldArray) = len(self.raw_bytes)
def __len__(self):
    return len(self.raw_bytes)

# FiniteFieldArray + FiniteFieldArray = FiniteFieldArray
def __add__(self, other):
    if not isinstance(other, FiniteFieldArray):
        raise TypeError("other is not a FiniteFieldArray")
    elif len(self.raw_bytes) != len(other.raw_bytes):
        raise ValueError("other is not same size")

    my_elearray = FiniteFieldArray.ff_bytearray_to_elearray(self.raw_bytes)
    other_elearray = FiniteFieldArray.ff_bytearray_to_elearray(other.raw_bytes)
    sum_elearray = [FiniteFieldArray.ff_elem_add(my_elearray[i], other_elearray[i])
                    for i in range(len(my_elearray))]
    return FiniteFieldArray(FiniteFieldArray.ff_elearray_to_bytearray(sum_elearray))

# FiniteFieldArray - FiniteFieldArray = FiniteFieldArray
def __sub__(self, other):
    return self.__add__(other)

# FiniteFieldArray * FiniteFieldArray = scalar (inner product)
# FiniteFieldArray * scalar = FiniteFieldArray
def __mul__(self, other):
    if isinstance(other, FiniteFieldArray):
        if len(self.raw_bytes) != len(other.raw_bytes):
            raise ValueError("other is not same size")

        my_elearray = FiniteFieldArray.ff_bytearray_to_elearray(self.raw_bytes)
        other_elearray = FiniteFieldArray.ff_bytearray_to_elearray(other.raw_bytes)
        x = 0
        for i in range(len(my_elearray)):
            x = FiniteFieldArray.ff_elem_add(x, FiniteFieldArray.ff_elem_mul(my_elearray[i], other_elearray[i]))
        return x
    elif isinstance(other, int):
        if other < 0 or other > FiniteFieldArray.ELEMENTNUM-1:
            raise ValueError("other is outside of finite field")

        my_elearray = FiniteFieldArray.ff_bytearray_to_elearray(self.raw_bytes)
        mul_elearray = [FiniteFieldArray.ff_elem_mul(my_elearray[i], other) for i in range(len(my_elearray))]
        return FiniteFieldArray(FiniteFieldArray.ff_elearray_to_bytearray(mul_elearray))
    else:
        raise TypeError("unknown other")

# FiniteFieldArray * FiniteFieldArray = scalar (inner product)
# scalar * FiniteFieldArray = FiniteFieldArray
def __rmul__(self, other):
    return self.__mul__(other)

# FiniteFieldArray / scalar = FiniteFieldArray
def __truediv__(self, other):
    if not isinstance(other, int):
        raise TypeError("other is not a scalar integer")
    elif other < 0 or other > FiniteFieldArray.ELEMENTNUM-1:
        raise ValueError("other is outside of finite field")

    my_elearray = FiniteFieldArray.ff_bytearray_to_elearray(self.raw_bytes)
    div_elearray = [FiniteFieldArray.ff_elem_div(my_elearray[i], other) for i in range(len(my_elearray))]
    return FiniteFieldArray(FiniteFieldArray.ff_elearray_to_bytearray(div_elearray))
```

# FiniteFieldArray == FiniteFieldArray

```python
def __eq__(self, other):
    if not isinstance(other, FiniteFieldArray): return False
    return self.raw_bytes == other.raw_bytes
```

# str(FiniteFieldArray)

```python
def __str__(self):
    hexstr = ''
    for i in range(len(self.raw_bytes)):
        hexstr += ('%02x' % self.raw_bytes[i]) + '
    #if i == 0: continue
    #elif (i+1) % 64 == 0 and i+1 != len(self.raw_bytes): hexstr += '
    #elif (i+1) % 2 == 0: hexstr += ''
    return hexstr
```

```python
def ff_rref(m, b):
    pi = 0
    for j in range(len(m)):
        # While we do not have a pivot for this row
        while m[j][pi] == 0:
            # Find a row below to swap with for a pivot at pi
            for k in range(j+1, len(m)):
                if m[k][pi] != 0:
                    # Swap with this row
                    (m[j], m[k]) = (m[k], m[j])
                    (b[j], b[k]) = (b[k], b[j])
                    break
            # Increment pivot index if we could not find a row to swap with
            if m[j][pi] == 0:
                pi += 1

        # If there is no pivots left, we're done reducing
        if pi == len(m[0]):
            return (m, b)

        # Divide through to have a pivot of 1
        m[j] /= m[j][pi]

        # Eliminate above & below
        for k in range(len(m)):
            if k != j and m[k][pi] != 0:
                m[k] = m[k] - (m[j]*m[k][pi])

        # Move onto the next pivot
        pi += 1

        # If there is no pivots left, we're done reducing
        if pi == len(m[0]):
            break
    return (m, b)
```

```python
def ffsolutions(a, x, b):
    (ap, bp) = ff_rref(a, b)
```

92
```python
solved = []
used = []
for i in range(len(ap)):
    r = ap[i]
    # If this row has only one non-zero entry
    reduced_coeffs = [r[j] != 0 for j in range(len(r))]
    if sum(reduced_coeffs) == 1:
        # Add the solution to our solved list
        solved.append(r[reduced_coeffs.index(True)])
        # Add the decoded LC to our used list
        used.append(b[i])
return (solved, used)

def ff_rref_testo:
    m = [FiniteFieldArray([33, 247, 109, 71, 139]),
         FiniteFieldArray([97, 221, 102, 127, 72]),
         FiniteFieldArray([101, 126, 186, 90, 103]),
         FiniteFieldArray([59, 234, 145, 197, 122]),
         FiniteFieldArray([75, 9, 213, 9, 3])]
    b = [1, 2, 3, 4, 5]
    (mp, bp) = ff_rref(m, b)
    for i in range(len(mp)):
        print(mp[i], '	', b[i])
    print('')

    m = [FiniteFieldArray([33, 247, 109, 71, 139]),
         FiniteFieldArray([97, 221, 102, 127, 72]),
         FiniteFieldArray([101, 126, 186, 90, 103]),
         2*FiniteFieldArray([97, 221, 102, 127, 72])]
    b = [1, 2, 3, 4]
    (mp, bp) = ff_rref(m, b)
    for i in range(len(mp)):
        print(mp[i], '	', b[i])
    print('')

    m = [FiniteFieldArray([0, 247, 109, 71, 139]),
         FiniteFieldArray([97, 221, 102, 127, 72]),
         FiniteFieldArray([101, 126, 186, 90, 103]),
         2*FiniteFieldArray([97, 221, 102, 127, 72])]
    b = [1, 2, 3, 4]
    (mp, bp) = ff_rref(m, b)
    for i in range(len(mp)):
        print(mp[i], '	', b[i])
    print('')
```

---

**A.3 stats_process.py**

```bash
#!/usr/bin/python3
```
# Plotter for NCGAB Simulator Results

```python
import sys
import json
import numpy
import pylab

def avg(x):
    return (sum(x) / float(len(x))) if len(x) > 0 else 0.0

def ci_95(x):
    std = numpy.std(x)
    ci = 1.96*std/numpy.sqrt(len(x))
    return ci

class StatsProcessO:
    def __init__(self, filename):
        f = open(filename)
        data = json.loads(f.read())
        print("Loaded %s %s" % (data['simParams']['NAME'], data['simParams']['DESC']))
        self.simParams = data['simParams']
        self.message_inserts = data['message_inserts']
        self.message_decodes = data['message_decodes']
        self.matrix_reduces = data['matrix_reduces']
        self.window_sizes = data['window_sizes']
        f.close()

    def dump_decodes(self):
        for i in range(len(self.message_inserts)):
            (itime, nid, pid) = self.message_inserts[i]
            print("[%.6f]: nid %d, insert %s % (itime, nid, pid))
            ndecodes = 0
            for (dtime, nid) in self.message_decodes: pid):
                print("\t[%.6f]: nid %d, decode" % (dtime - itime, nid))
                ndecodes += 1
                print("\t\tnumber of decodes: %d / %d = %.2f%%\n" % (ndecodes,
                    self.simParams['SIM_NUM_PEERS'] - 1, 100.0 * float(ndecodes)/(self.simParams['SIM_NUM_PEERS']-1)))

    def dump_reduces(self):
        for (itime, nid, m, n, ns) in self.matrix_reduces:
            print("[%.6f]: nid %d, reduce %d to %d\n" % (itime, nid, m, n))

    def dump_window_sizes(self):
        for (t, nid, ss, gs) in self.window_sizes:
            print("[%.6f]: nid %d, solved size %d, gossip size %d\n" % (t, nid, ss, gs))

    def compute_delays(self):
        delay_min = []
        delay_max = []
        delay_avg = []

        # For each inserted message
        for (itime, pid) in self.message_inserts:
            # If it was decoded
            if pid in self.message_decodes and len(self.message_decodes[pid]) > 0:
                delay_min.append(abs(itime - self.message_decodes[pid][0]))
                delay_max.append(abs(itime - self.message_decodes[pid][-1]))
                delay_avg.append(avg(self.message_decodes[pid]))
```

# Calculated the delays to decoding

delays = [(dtime - itime) for (dtime, _) in self.message_decodes[pid]]

# Add it to our tables
delay_min.append(delays[0])
delay_max.append(delays[-1])
delay_avg.append(sum(delays) / float(len(delays)))

return (delay_min, delay_max, delay_avg)

def compute_pdecodes(self):
    times = []
pids = []
pdecode = []

    # For each inserted message
    for (time, _, pid) in self.message_inserts:
        times.append(time)
pids.append(pid)
        if pid in self.message_decodes and len(self.message_decodes[pid]) > 0:
            pdecode.append(100.0 * (float(len(self.message_decodes[pid]))) / 
                float(self.simParams['SIM_NUM_PEERS']-self.simParams['SIM_NUM_EVIL_PEERS']-1))
        else:
            pdecode.append(0.0)

    return (times, pids, pdecode)

def compute_avg_delay(self):
    (delay_min, delay_max, delay_avg) = self.compute_delays()
    avg_delay_min = avg(delay_min)
    avg_delay_max = avg(delay_max)
    avg_delay_avg = avg(delay_avg)

    print("\tAvg Delay: %.6f" % avg_delay_avg)
    print("\tMin Delay: %.6f" % avg_delay_min)
    print("\tMax Delay: %.6f" % avg_delay_max)

    print ()

def print_stats(self):
    print(self.simParams['NAME'])

    print("\tNum inserts: %d\n" % len(self.message_inserts))

    (_, _, pdecode) = self.compute_pdecodes()
    print("\tDecode Percent: %.2f\n" % avg(pdecode))

    (delay_min, delay_max, delay_avg) = self.compute_delays()
    avg_delay_min = avg(delay_min)
    avg_delay_max = avg(delay_max)
    avg_delay_avg = avg(delay_avg)

    print("\tAvg Delay: %.6f\n" % avg_delay_avg)
    print("\tMin Delay: %.6f\n" % avg_delay_min)
    print("\tMax Delay: %.6f\n" % avg_delay_max)

    print ()


########################################################

def print_stats(self):
    print(self.simParams['NAME'])

    print("\tNum inserts: %d\n" % len(self.message_inserts))

    (_, _, pdecode) = self.compute_pdecodes()
    print("\tDecode Percent: %.2f\n" % avg(pdecode))

    (delay_min, delay_max, delay_avg) = self.compute_delays()
    avg_delay_min = avg(delay_min)
    avg_delay_max = avg(delay_max)
    avg_delay_avg = avg(delay_avg)

    print("\tAvg Delay: %.6f\n" % avg_delay_avg)
    print("\tMin Delay: %.6f\n" % avg_delay_min)
    print("\tMax Delay: %.6f\n" % avg_delay_max)

    print ()

########################################################
def plot_pdecodes_vs_time(self):
    (times, pids, pdecode) = self.compute_pdecodes()
    avg_pdecode = avg(pdecode)

    pylab.figure()
    pylab.plot(times, pdecode, 's')
    pylab.vlines(times, 0, pdecode)
    pylab.hlines(avg_pdecode, times[0], times[-1])
    pylab.title('Final Decoding of Inserted messages')
    pylab.xlabel('Time')
    pylab.ylabel('Percent Peers Decode')
    pylab.ylim([0, 101])

def plot_window_sizes_vs_time(self):
    time = {}
    solved_size = {}
    gossip_size = {}

    for (t, nid, ss, gs) in self.window_sizes:
        if nid not in time:
            time[nid] = []
            solved_size[nid] = []
            gossip_size[nid] = []

        time[nid].append(t)
        solved_size[nid].append(ss)
        gossip_size[nid].append(gs)

    pylab.figure()
    for nid in time.keys():
        pylab.plot(time[nid], solved_size[nid])
    pylab.title('Solved Window Size')
    pylab.xlabel('Time')
    pylab.ylabel('Items')

    pylab.figure()
    for nid in time.keys():
        pylab.plot(time[nid], gossip_size[nid])
    pylab.title('Gossip Window Size')
    pylab.xlabel('Time')
    pylab.ylabel('Items')

#------------------------------------------------------------------------------------------

class StatsPlot():
    def __init__(self, plotFilePrefix, statsProcList):
        self.statsProcList = statsProcList
        self.plotFilePrefix = plotFilePrefix

    def plot_pdecode_vs_throughput(self):
        data = {}
        for sim in self.statsProcList:
            params = sim.simParams
            numPeers = params['SIM_NUM_PEERS']
            contributeInterval = params['CONTRIBUTE_INTERVAL']

            if numPeers not in data:
                data[numPeers] = {} 

            if contributeInterval not in data[numPeers]:
                data[numPeers][contributeInterval] = []

            data[numPeers][contributeInterval].append(sim.compute_avg_pdecode()[0])

            #------------------------------------------------------------------------------------------
pylab.figure()
for numPeers in sorted(data.keys()):
    x = sorted(data[numPeers].keys())
    y = [numpy.mean(data[numPeers][r]) for r in x]
    ci = [ci_95(data[numPeers][r]) for r in x]
    pylab.errorbar(x, y, yerr=ci, fmt="s-", label="N=%d" % numPeers)
    pylab.xlabel('Contribution Interval')
    pylab.ylabel('Average Percent Decode')
    pylab.title('Availability vs. Contribution Interval')
    pylab.legend(loc='lower right')
    pylab.savefig(self.plotFilePrefix + "-availability.eps")

def plot_delay_vs_throughput(self):
    data = {}
    for sim in self.statsProcList:
        params = sim.simParams
        numPeers = params['SIM_NUM_PEERS']
        contributeInterval = params['CONTRIBUTE_INTERVAL']
        if numPeers not in data:
            data[numPeers] = {}
        if contributeInterval not in data[numPeers]:
            data[numPeers][contributeInterval] = []
        data[numPeers][contributeInterval].append(sim.compute_avg_delay()[0])

    pylab.figure()
    for numPeers in sorted(data.keys()):
        x = sorted(data[numPeers].keys())
        y = [numpy.mean(data[numPeers][r]) for r in x]
        ci = [ci_95(data[numPeers][r]) for r in x]
        pylab.errorbar(x, y, yerr=ci, fmt="s-", label="N=%d" % numPeers)
        pylab.xlabel('Contribution Interval')
        pylab.ylabel('Average Delay')
        pylab.title('Delay vs. Contribution Interval')
        pylab.legend()
        pylab.savefig(self.plotFilePrefix + "-delay.eps")

def plot_pdecode_vs_evil_nodes(self):
    data = {}
    for sim in self.statsProcList:
        params = sim.simParams
        numPeers = params['SIM_NUM_PEERS']
        percentEvil = 100.0 * params['SIM_NUM_EVIL_PEERS'] / float(params['SIM_NUM_PEERS'])
        if numPeers not in data:
            data[numPeers] = {}
        if percentEvil not in data[numPeers]:
            data[numPeers][percentEvil] = []
        data[numPeers][percentEvil].append(sim.compute_avg_pdecode()[0])

    pylab.figure()
    for numPeers in sorted(data.keys()):
        x = sorted(data[numPeers].keys())
        y = [numpy.mean(data[numPeers][r]) for r in x]
        ci = [ci_95(data[numPeers][r]) for r in x]
        pylab.errorbar(x, y, yerr=ci, fmt="s-", label="N=%d" % numPeers)
        pylab.xlabel('Percent Attackers')
        pylab.xlim([-1, 31])
247    pylab.ylabel('Average Percent Decoded')
248    pylab.ylim([0, 105])
249    pylab.title('Availability vs. Percent Attackers')
250    pylab.legend(loc='lower right')
251    pylab.savefig(self.plotFilePrefix + '-availability.eps')
252
253 def plot_delay_vs_evil_nodes(self):
254    data = {}
255    for sim in self.statsProcList:
256        params = sim.simParams
257        numPeers = params['SIM_NUM_PEERS']
258        percentEvil = 100.0 * params['SIM_NUM_EVIL_PEERS'] / float(params['SIM_NUM_PEERS'])
259        if numPeers not in data:
260            data[numPeers] = {}
261        if percentEvil not in data[numPeers]:
262            data[numPeers][percentEvil] = []
263            data[numPeers][percentEvil].append(sim.compute_avg_delay()[0])
264
265    pylab.figure()
266    for numPeers in sorted(data.keys):
267        x = sorted(data[numPeers].keys())
268        y = [numpy.mean(data[numPeers][r]) for r in x]
269        ci = [ci_95(data[numPeers][r]) for r in x]
270        pylab.errorbar(x, y, yerr=ci, fmt='s-', label="N=d = %d" % numPeers)
271    pylab.ylabel('Average Delay')
272    pylab.xlim([-1, 31])
273    pylab.ylim([0, 25])
274    pylab.title('Delay vs. Percent Attackers')
275    pylab.legend()
276    pylab.savefig(self.plotFilePrefix + '-delay.eps')
277
278 if __name__ == '__main__':
279    if len(sys.argv) < 3:
280        print("Usage: <plot output prefix> <simulation data files ...>")
281        sys.exit(1)
282
283 # From http://www.scipy.org/Cookbook/Matplotlib/LaTeX_Examples
284 fig_width_pt = 253.0
285 inches_per_pt = 1.0/72.27
286 golden_mean = (numpy.sqrt(5)-1.0)/2.0
287 fig_width = fig_width_pt/inches_per_pt
288 fig_height = fig_width*golden_mean
289 fig_size = [fig_width, fig_height]
290 params = {'backend': 'ps',
291            'axes.labelsize': 9,
292            'text.fontsize': 9,
293            'legend.fontsize': 9,
294            'xtick.labelsize': 7,
295            'ytick.labelsize': 7,
296            'text.usetex': True,
297            'figure.figsize': fig_size,
298            'figure.subplot.bottom': 0.15,
299            'font.size': 9,
300            'font.family': 'serif'}
301 pylab.rcParams.update(params)
302
303 plotFilePrefix = sys.argv[1]
304 simsToProcess = sys.argv[2:]
305
306 if len(simsToProcess) == 1:
statsProc = StatsProcess(simsToProcess[0])
statsProc.print_stats()
statsProc.plot_pdecodes_vs_time()
statsProc.plot_window_sizes_vs_time()
pylab.show()

elif len(simsToProcess) > 1:
    statsProcList = [StatsProcess(p) for p in simsToProcess]
    for s in statsProcList:
        s.print_stats()
        p = StatsPlot(plotFilePrefix, statsProcList)
        if "Evil" in sys.argv[2]:
            p.plot_pdecode_vs_evil_nodes()
            p.plot_delay_vs_evil_nodes()
        else:
            p.plot_pdecode_vs_throughput()
            p.plot_delay_vs_throughput()
        pylab.show()
Bibliography


