OPTIMIZING AND DEBUGGING DISTRIBUTED SYSTEMS WITH DECLARATIVE LANGUAGES

A Dissertation
submitted to the Faculty of the
Graduate School of Arts and Sciences
of Georgetown University
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
in Computer Science

By

Yuankai Zhang, M.S.

Washington, DC
April 23, 2019
Copyright © 2019 by Yuankai Zhang
All Rights Reserved
OPTIMIZING AND DEBUGGING DISTRIBUTED SYSTEMS WITH DECLARATIVE LANGUAGES

Yuankai Zhang, M.S.

Dissertation Advisor: Wenchao Zhou, Ph.D.

ABSTRACT

Distributed systems are pervasively adopted in recent years to keep up with the ever-increasing needs of data-oriented applications. The increasingly demanding requirements of scalability, availability, and reliability bring tremendous challenges and opportunities to distributed systems research. Over the decades, people have proposed and implemented numerous distributed protocols and systems to solve emerging challenges. As more components are integrated into new systems, the complexity of these systems increases at a high pace. The inter-components overhead starts to impact the overall system performance as system complexity rises. At the same time, more complex systems tend to fail more frequently. The elevated cost and failure rate call for urgent research attention.

In this dissertation, we investigate novel methods to optimize and debug distributed systems with declarative languages. We first design and implement DeDoS, a distributed runtime platform that manages system resources in a fine-granularity fashion. Applications deployed on DeDoS are split into multiple Minimum Splittable Units (MSUs), where each MSU is separately replicable anywhere in the system that has enough resources.

With the runtime platform, we then focus on optimizing resource utilization for distributed programs written in Network Datalog (NDlog). NDlog proposes a straightforward approach for engineers to write distributed programs in a declarative language with a remarkably concise yet expressive format. Prior works have shown the effectiveness of
NDlog in implementing complex distributed protocols. We propose and implement a compiler that automatically translates NDlog programs into MSUs, and schedules and potentially replicates MSUs according to the workloads observed at runtime such that system resources are prioritized to the portion of the program that has the highest demand. One particular challenge that we solve is to strategically place MSUs such that we can minimize the communication cost. We present an algorithm that solves the optimal MSU placement problem by reducing to a min-cut problem on constructed NDlog execution graphs.

With increased scale and more complex protocols, debugging distributed systems also becomes a daunting task. Prior work has shown that network provenance significantly assists network administrators in diagnosing and debugging network systems by explaining system behavior and revealing the dependencies between system states. We extend existing network provenance systems to preserve privacy. We examine the privacy aspect of network provenance by proposing adversary models, designing encryption schemes with Searchable Symmetric Encryption, implementing the privacy-preserving network provenance system, and analytically and experimentally evaluating the security and performance of the proposed scheme.

INDEX WORDS: Distributed Systems, Network Provenance, Declarative Language, Resource Optimization, Private Debugging
ACKNOWLEDGMENTS

First and foremost, I’m deeply indebted to my advisor, Dr. Wenchao Zhou, for his unreserved support and continuous guidance during my journey of pursuing a Ph.D. at Georgetown. Without his help, this dissertation would not be possible and my journey would be endless. He always inspires and motivates me with his knowledge, enthusiasm, diligence, and patience. Dr. Zhou is more than an advisor, but also a lifetime mentor to me.

I would also like to express my most sincere gratefulness to my other committee members, Dr. Boon Thau Loo, Dr. Clay Shields, and Dr. Micah Sherr for their valuable time, insightful comments and helpful advices. I’ve worked with Boon during my Master study at Penn. His CIS 553 class is simply my all time favorite. He was also the one who interested me to academic research in Computer Science. Clay was the DGS when I joined Georgetown. He introduced me to the broader security research area. My publication with highest citation to date, and possibly forever, comes from an idea of him. Micah is de facto my co-advisor at Georgetown. He is a co-author of all my publications. His insightful views and high standards are often the key to success. And he is the DGS when I graduate from Georgetown with a Ph.D., what a journey.

I simply cannot thank my collaborators enough for our collaborations in many research projects: Tavish Vaidya, Dr. Ang Chen, Akshay Sriraman, Dr. Andreas Haeberlen, Dr. Linh Thi Xuan Phan, Dr. Nicholas Carlini, Pratyush Mishra, Dr. David Wagner, Dr. Adam O’Neill, Henri Maxime Demoulin, Isaac Pedisich, Dr. Nik Sultana, Bowen Wang, Jingyu Qian, Bob DiMaiolo, Chirag Shah, Hui Lyu, Shaobo Wang, Dr. Chen Chen, and Dr. Zaoxing Liu.
I would also like to thank many professors for their great courses and discussions that helped me grow: Dr. Jeremy Fineman, Dr. Calvin Newport, Dr. Grace Hui Yang, Dr. Lisa Singh, Dr. Eric Burger, Dr. Bala Kalyanasundaram, Dr. Mark Maloof and Dr. Matt Blaze.

My sincere thankfulness also extends to fellows at Georgetown Computer Science Department for their warm support and inspiring discussions that greatly helped my research and life: Dr. Sicong Zhang, Dr. Jiyun Luo, Dr. Yifang Wei, Dr. Tonghe Wang, Dr. Brad Moore, Dr. Henry Tan, Mohammad Zaheri, Elchin Asgarli, Zhiwen Tang, Hao-Ren Yao, Shiqi Liu, Hongkai Wu, Lingqiong Tan, Jianxian Wu, Yunyun Chen, Jie Zhou, Chenxi Lu, Daodao Wang, Weiye Deng, Yanan Zhu, Weitong Liu, Xi Yang, Yingyu Tao, Fa Wang, Zimo Guo and Diwen Xiong.

I would also like to give my thanks to many friends in Northern Virginia for organizing game nights that relieves me from stressed research: Dr. Shanhe Yi, Liu Yin, Dr. Xiaocheng Zou, Kevin Qin, Yamin Li, Yi Zhou, Li Huang, Ziyou Li, Tianshi Wang, Huida Tao, Yiran Zhou, Ang Shen, Yiran Wang, Jieling Ding and Zheming Deng.

During my years as Ph.D. student, I interned at VeriSign Labs and Google. I would like to thank my referrers, mentors and collaborators: Dr. Andrew West, Shumon Huque, Allison Mankin, Rui Li, Dr. Yanhua Sun and Dr. Sunghwan Ihm.

Special thanks to my friends with whom I spent countless nights gaming: Dr. Yang Jiang, Dr. Zhihao Jiang, Yuqi Yang, Cheng Huang and Xiao Yi.

Last but not least, I owe my deepest gratitude to my family for their unconditional support, thoughtful understanding, and most importantly, endless love. My parents supported my education all the way to my Master’s degree. They truly believe and have undoubted faith in me that someday I would succeed. I’m also hugely indebted to my lovely wife, Xiao Ren, for her encouragement and accompany. She is very smart, strong and supportive. I truly can’t wait to start the next journey with her. I dedicate this dissertation to my beloved family.
TABLE OF CONTENTS

CHAPTER

1 Introduction ................................................................. 1
  1.1 Optimize Distributed Systems ........................................ 3
  1.2 Debugging Distributed Systems ....................................... 5
  1.3 Research Questions .................................................. 6
  1.4 Contributions ........................................................ 7
  1.5 Outline ............................................................... 8

2 Background, System Model and Related Work .......................... 9
  2.1 Declarative Language .................................................. 9
  2.2 Related Work ........................................................ 13

3 System Resource Optimization ........................................... 18
  3.1 Declarative Dispersion-Oriented Software Design .................. 18
  3.2 Resource Allocation .................................................. 23
  3.3 Case Studies .......................................................... 25
  3.4 Evaluation ............................................................. 26
  3.5 Summary .............................................................. 30

4 Network Resource Optimization ......................................... 31
  4.1 Network Inefficiency .................................................. 31
  4.2 Proposed Solution ..................................................... 33
  4.3 Implementation and Evaluation ...................................... 46
  4.4 Summary .............................................................. 53

5 Privacy Preserving Debugging ............................................ 54
  5.1 Overview and Goals .................................................... 54
  5.2 Structured Encryption for Colored Subgraphs ...................... 57
  5.3 Privacy Preserving Network Provenance Design ................... 66
  5.4 Evaluation ............................................................. 76
  5.5 Summary .............................................................. 86

6 Summary and Future Directions ......................................... 87
  6.1 Summary .............................................................. 87
  6.2 Future Directions ..................................................... 88
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Network topology in example.</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>The compiled workflow for rule $r_2$.</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Example use case of Declarative Dispersion-Oriented Software.</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>The throughput when experiencing high load.</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Requests latency and success rate during varying high loads.</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Two options for evaluating $r_2$.</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>A Min-Cut based solution to minimize communication cost.</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Overview of proposed optimization.</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Query graph and communication graph.</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Hyper graph and labeling.</td>
<td>35</td>
</tr>
<tr>
<td>4.6</td>
<td>Min-Cut graph and result.</td>
<td>36</td>
</tr>
<tr>
<td>4.7</td>
<td>Min-Cut on hypergraph.</td>
<td>37</td>
</tr>
<tr>
<td>4.8</td>
<td>Generate query graph.</td>
<td>39</td>
</tr>
<tr>
<td>4.9</td>
<td>Construct hyper graph.</td>
<td>40</td>
</tr>
<tr>
<td>4.10</td>
<td>Label on hyper graph.</td>
<td>41</td>
</tr>
<tr>
<td>4.11</td>
<td>Minimum Splittable Unit placements and flow graph.</td>
<td>44</td>
</tr>
<tr>
<td>4.12</td>
<td>Additional Minimum Splittable Units and data flow.</td>
<td>49</td>
</tr>
<tr>
<td>4.13</td>
<td>Optimization latency.</td>
<td>50</td>
</tr>
<tr>
<td>4.14</td>
<td>Optimization savings.</td>
<td>51</td>
</tr>
<tr>
<td>4.15</td>
<td>Optimization savings with comparison.</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>Structured encryption scheme for colored subgraphs $GE_1$.</td>
<td>64</td>
</tr>
</tbody>
</table>
5.2 Searchable Symmetric Encryption scheme with fragmentation. 67
5.3 An example provenance graph. 68
5.4 Answering provenance query. 70
5.5 Views from different parties of provenance graph in Figure 5.3. 71
5.6 Components of Q’s view in Figure 5.5. 71
5.7 Bandwidth utilization of provenance queries. 78
5.8 Latency distribution. 80
5.9 Communication overhead with varying network sizes. 81
5.10 Communication overhead with varying fragmentation sizes. 82
5.11 Bandwidth utilization of 100 node running 1000 queries in Chord. 83
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Network Datalog packet processing application overheads.</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Provenance dictionaries before fragmentation.</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>Fragmentation of the inverted-indices (dictionaries) from Table 5.1.</td>
<td>72</td>
</tr>
<tr>
<td>5.3</td>
<td>Experiment configurations.</td>
<td>77</td>
</tr>
</tbody>
</table>
Distributed systems are becoming universally pervasive since the new Millennium. From emails to e-commerce, from search engines to social networks, from mobile computing to artificial intelligence, distributed systems are playing an essential role in many scenarios. Partly pushed by FOMO (the Fear of Missing Out), people check their social media and other channels to the interconnected e-world more frequently. As a result, people spend a significant amount of their time “staying e-connected” more than ever; billions of emails, search queries and social media logins are being sent and processed on a daily basis. On the other side, the Internet of Things connects billions of devices to the cloud 24x7. Smart devices around the globe continuously stream data to the cloud for processing and storage. The amount of data generated and processed by these applications is extremely large. All these services rely heavily on distributed systems since they provide many benefits, for example, more reliability and availability against catastrophic disasters \[89\], and better localities for services that benefit from lower latency, e.g., realtime virtual reality \[6\] and competitive gaming \[5\].

With the unprecedented level of requirements in scalability, availability and reliability of distributed systems, it is profoundly important that systems are highly optimized for resource utilization. Wasting valuable resources that could have been utilized by other components would miss tremendous opportunities for improved performance. For instance, a database backed web application may consist of three webservers, two database servers and one load balancer as the entry point. When all three webservers are under heavy CPU work
load, it leaves no space for a load balancer to further distribute incoming requests, regardless of the fact that the database servers have many idle cycles. If some CPU intensive workloads could be offloaded to database servers, the overall throughput would improve significantly [19, 22, 28, 29]. Take simple packet forwarding application as another example.

Network packets, or more generally network messages, need to be forwarded to different locations for processing according to “some” rules specified by the application. Such rules could be network routing tables for IP packets, or load balancing policies for load-balanced service requests. To provide high throughput and failure resistance, such forwarding applications are themselves distributed as well. In these forwarding applications, there are plenty of possible bottlenecks or hotspots that limit the overall performance. For instance, routing table entry matching or load balancing policy lookup may involve significant CPU or memory intensive operations. Spreading such operations across different machines with carefully designed coordination could improve overall throughput. Other more complicated applications could easily have orders of magnitudes more potential bottlenecks or hotspots. Adding more resources would not scale infinitely. Fully utilize available resources in the system is a promising direction for quest of a scalable solution.

Equally important, when the systems fail, either due to inadequate resources, software bugs or misconfiguration, it is critical that engineers and administrators be able to quickly identify the root cause of the problem, since even a short period of service outage could lead to huge financial impact [2]. Debugging distributed systems has long been notorious due to the distributed nature and large scale. The root cause of a failure could be an event happened in the system that is seemingly irrelevant to the error. It requires reliability engineers to first reasonably trace back to the event and then find a cure for the failure. The former step isn’t always straightforward, especially in a system that processes millions of event in a second. Take the packet forwarding application example once more. In the event that a packet, potentially malicious, was routed to an incorrect destination, there could be multiple
reasons for the unexpected behavior, e.g., the routing table might have been compromised or misconfigured, or the firewall might have failed to detect and block the packet. If the respondent team was provided with a complete history and causality of events and states in the system, they would be able to locate the root cause in a more timely manner. The history and causality of events and states are captured by provenance, which we will discuss shortly.

Declarative networking has been well studied by systems research community [9, 21, 23, 59, 61, 62, 63, 64, 67, 101, 104]. Declarative language has many strengths that make it a perfect fit for our system model. Programs written in declarative language are very concise. And they can be easily compiled to general programs for efficient execution. In addition, with minimal efforts, provenance information can be easily extracted from declarative language programs.

In this dissertation, we optimize and debug distributed systems with declarative languages. We first briefly discuss opportunities for optimization in distributed systems, and then describe some challenges in current debugging techniques.

1.1 Optimize Distributed Systems

With the current level of demand in scalability and availability, monolithic server software design simply cannot scale well enough to support the fast increasing system loads. The monolithic server software cannot keep up with the pace of data explosion due to the inefficiency of wasting valuable and scarce resources that could be used by other components of the program/application [28, 29].

A straightforward mechanism is simply to deploy more resources. This is often the de facto solution deployed in production systems: during high load, the service is automatically replicated as virtual machines (VMs) or lightweight containers, on multiple machines
to scale “elastically” to the extra load. Replicating all of the VM’s or container’s resources, regardless of which are being consumed, is enormously costly, making this approach unusable for most service providers. For example, if only a database JOIN operator is under heavy load, the replication of an entire VM/DB mitigates the performance, but does so at an enormous overhead.

In light of the limitations of existing solutions, we propose a radically different approach called DeDoS. We advocate software development in a modular fashion, such that components can be moved and replicated independently.

DeDoS provides a framework which allows programmers to construct more resilient applications through the use of fine-grained, modular components. Ideally, each component handles some small, focused aspect of an application that may be resource hungry. Example components include database JOIN operation, network operation, or I/O operation. Crucially, with DeDoS, programmers do not have to worry about most of the deployment specifics: DeDoS offers an adaptive controller that makes real-time decisions on placing these components within physical resources (e.g., machines in a datacenter), and then adaptively clones, merges, or migrates them in order to meet service-level agreement (SLA) objectives. When SLA objectives are about to be violated, individual components that are overloaded are replicated.

The DeDoS architecture offers two benefits. First, the fine-grained components make it easier for the service provider to deploy all available resources on all machines during peak load, exactly as needed. For instance, DeDoS can respond to a sudden hashing load by temporarily enlisting other machines with only spare CPU cycles to help with hashing. Second, the replication approach is not functionality-specific and can thus potentially mitigate unexpected hot spot. Once DeDoS recognizes that a component is overloaded or its throughput appears to drop, it can respond by replicating that particular component. This potentially allows a flexible and automatic response to varying hotspots in the system.
1.2 Debugging Distributed Systems

System administrators often find themselves trying to debug the system without much meaningful insights. They are usually lost in large collections of logs and can only rely on prior experience to find the needle in haystack. Network provenance [100] allows network administrators to determine the precise causes and effects of network state, significantly easing otherwise burdensome tasks such as failure diagnostics, forensic analysis, network debugging, and network accounting. Efficient methods of storing and querying provenance [102] have enabled practical network provenance systems, spurring provenance products that have been deployed in enterprise networks [76].

To date, deployments of network provenance systems have been restricted to single administrative domains due in part to the security and privacy issues that arise when data are shared across administrative boundaries. However, since the Internet is governed by distributed protocols (BGP and DNS are notable examples), secure provenance schemes that operate across networks could provide enormous benefit. For example, a provenance-enabled Internet control-plane enables network administrators to reason about its own routing rules and better diagnose cross-domain routing faults such as bad gadgets [103]; with provenance support, participants of distributed hash tables can better understand sudden workload shifts and performance glitches.

Existing work investigates secure provenance systems in distributed environments that are robust against malicious or compromised endpoints or network elements [68, 103]. There, the focus has been on detecting false provenance information. In this dissertation, we argue that authenticity is not by itself sufficient for practical cross-domain provenance systems. Provenance data collected on individual devices contains rich information that reveals details of system executions and decision making processes, some of which might be considered sensitive or confidential and should not be visible by unauthorized parties in
the distributed system. When disseminating provenance information, we need to take into consideration organizations’ privacy preferences. Prior position papers [13, 46] have previously identified such need, and suggested that partial and differential provenance access should be granted in a privacy-preserving manner.

1.3 Research Questions

Given the challenges and opportunities mentioned above, in this dissertation, we ask, investigate and answer the following research questions:

- To improve the resource utilization, one potential method is to make local resources globally available. Current VM/container solutions do not share local spare resources efficiently due to the larger footprint. A natural question would be How would finer modularity in distributed systems affect resource utilization? Finer modularity reduces resource requirement for each self-contained components. It enables offloading small components to other locations in the system with enough resources. On the other hand, migrating large number of tiny components in the network may incur overheads in scheduling that degrade the overall performance. We should ensure that the overheads are kept at a minimal level.

- Different from other system resources, network bandwidth utilization requires careful coordination and scheduling between all machines. Researches have looked at bringing computation to where data reside to avoid sending data across the network. A particular interesting question along the line is to investigate How do different placements of components in distributed systems affect network overhead and utilization? An optimal placement of components sends less data over the network to save valuable bandwidth. Such global optimal decision requires a global view and some optimization algorithms. Obtaining the global view and executing
the optimization algorithm would inevitably incur overheads. It is of paramount important to ensure such overheads would not introduce intolerable latency to the system.

- Provenance has shown to be valuable in distributed systems debugging. One limiting factor for wider applicability of network provenance is non-privacy preserving maintenance and query. To tackle this particular problem, we ask *Is it possible to enable privacy preserving debugging in distributed systems with network provenance?* The security model as well as leakage profile is essential to preserving privacy. Overly strict security model leaks less information, but incur more computation and storage. A too loose model, on the other hand, leaks more information and provides less privacy guarantees. The encryption scheme also requires careful design to be secure against adversaries. The incurred latency in provenance maintenance and query should not affect the usability of the system.

Throughout this dissertation, we answer each question with strong evidence obtained from our experiments.

### 1.4 Contributions

Prior works have looked at distributed system resource optimization and efficient debugging. To further improve resource utilization and enable privacy preserving debugging, in this dissertation, we make the following contributions to enrich existing researches:

- **DeDoS framework** We propose DeDoS architecture and present design challenges and our approach to create applications as a dataflow of MSUs. We enable automatic compilation of an application written in declarative language to deploy on DeDoS. We implement the DeDoS prototype and evaluate it in a case study that shows DeDoS improves system throughput while incur minimal overhead.
- **Min-Cut based algorithm for optimizing network communication** We propose and formally prove a novel Min-Cut based algorithm that strategically places MSUs in the network to better allocate network resources such that network communication overhead is minimized. We integrate the optimization algorithm into the compiler that automatically translates NDlog programs into DeDoS applications, and evaluate the performance and overhead of the Min-Cut based algorithm.

- **Privacy preserving debugging** To assist preserving privacy in distributed system debugging with network provenance, we propose a security model to better capture privacy leakage profile and a novel SSE based encryption scheme to maintain and query provenance. To demonstrate the feasibility of our solution, we evaluate our implemented prototype and shows that our system preserves privacy in network provenance while incur reasonable overheads.

1.5 **Outline**

The rest of this dissertation is organized as follows: In Chapter 2, we provide some backgrounds on declarative language, describe our system model, and discuss related works. Then we propose the DeDoS system framework that aims to further optimize distributed systems by better utilizing available system resources in Chapter 3. With the DeDoS platform, we propose a novel algorithm that optimizes network communication overheads in distributed systems by strategically place components in the network in Chapter 4. In Chapter 5, we explore a novel technique to allow engineers and administrators to debug distributed systems in a privacy preserving manner. Finally in Chapter 6, we summarize the dissertation and briefly discuss some interesting possible directions for future work.
In this Chapter, we first provide some background and introduce our system model for declarative language in §2.1. Then we discuss some related works for resource management in distributed systems and utilizing Network Provenance for diagnosis in §2.2.

2.1 Declarative Language

Network Datalog (NDlog) proposes a declarative approach towards specifying and implementing network applications [65]. It eases a programmer’s task when writing complex network applications, and provides a significant reduction in code size. For instance, the Chord [85] distributed hash table can be written in NDlog in 47 lines of code [62].

In NDlog, system states are modeled as a set of tables. Each table contains tuples that are either input by users or derived from other tuples. Each NDlog program consists of a set of rules, which define how tuples are derived from other tuples. The rules are in the form of “rid head :- body1, body2, ..., bodyn.” The rule body consists of a list of literals. Each literal is either a predicate or an expression. Predicate literal contains a list of attributes, within which one of them is annotated with an @ sign, which means this predicate is located at this location/node. Expression literal involves functions applied to attributes that could be evaluated to a boolean result. Intuitively, the rule head is generated if all literals in the rule body are present or evaluate to true.

We use the following NDlog program as a running example throughout this chapter to illustrate model.
This NDlog program computes all pair-wise paths between each node pair in the network. Rule \( r_1 \) specifies that if there is a link between node \( S \) and \( D \), insert a path record at \( S \) with \([S, D]\) as path. Rule \( r_2 \) indicates that if there exists a path between node \( Z \) and \( D \), and a path between node \( S \) and \( Z \), then insert a path record at \( S \) that extends \( P \) by pretending \( S \) at the beginning of \( P \). This rule effectively extends existing path to one more hop.

We further assume a network topology shown in Figure 2.1. Each machine locally maintains a \textit{link} and \textit{path} table of its own. For simplicity, we omit the link weights and assume they have identical costs.

2.1.1 NDLOG Compilation

Each NDlog program contains a few NDlog rules, whose semantics are explained above. The compilation procedure of NDlog program consists of the following three main steps:

- Localization
- \{Event, Condition, Action\}ization
- Code Generation
The first step is referred to as “Localization”, it involves a localization rewrite procedure: if all predicates in the rule body are located at the same location, then it simply keep such rules intact. For rules that have predicates located at different locations, these predicates need to be shipped to a same location for the JOIN operation. Each predicate in NDlog rule body is rewritten in a way that all predicates will have the same location specifier. The new location specifier is usually chosen to be the JOIN key.

In our running example, rule r1 remains unchanged, and r2 will be rewritten as
\[
\text{path}(\@S, \text{D}, P) \leftarrow \text{link}(S, \@Z), \text{path}(\@Z, \text{D}, P2), P=\text{concat}(S,P2) \quad 1.
\]

We can observe two points for the new rule: 1) \(\text{link}(S, \@Z)\)’s location specifier has changed from S to Z. This reflects shipping \text{link} tuple from S to Z. 2) all predicates now are all located at Z, so we can process joining \text{link} and \text{path} at Z locally.

After JOIN, the new rule specifies that the JOIN result \text{path}(S,D,P) to be shipped to S and stored there. It is easy to see that the new rule involves first shipping \text{link}(S,Z) to Z and then shipping \text{path}(S,D,P) back to S. If we had chosen the JOIN to take place at S, it only involves shipping \text{path}(Z,D,P2) to S. We will discuss this inefficiency in §4.1 in more detail.

The second step is “ECA” rewrite, which further rewrites the rules into Event, Condition, Action rules. Each rule is rewritten into several new rules that triggers corresponding executions based on events/conditions/actions happened in the system.

For a rule of the form \(h : - b_1, b_2, \ldots, b_n\), a delta rule is generate for each predicate, where the \(k^{th}\) delta rule is of the form \(\Delta h : - b_1, b_2, \ldots, b_{k-1}, \Delta b_k, b_{k+1}, \ldots, b_n\). The \(\Delta b_k\) is an insertion or deletion of tuple in \(b_k\).

For instance, \(r1\) will be rewritten into

\[
\text{rlEca0Ins A_ADD<path(@S,D,P)> : - E_INSERT<link(@S,D)>, P=[S,D].}
\]

\[1\] There is another helper rule \text{link’}(S, \@Z) := \text{link}(S, Z). generated as well, which represents sending \text{link} tuple across network.
Here “A_” and “E_” represent Action and Event. The first new rule specifies if there is a new link tuple inserted into the system, generate a corresponding path tuple. The second new rule indicates that if a link tuple is deleted from the system, delete the corresponding path tuple as well. We will provide more details for r2’s ECA rewrites later in §4.2.5.

From now on, we will use one delta rule to represent such two rules.

The third step takes these new ECA rules and generate a C++ function for each rule. In our example, rule r1Eca0Ins will generate the following function:

```cpp
class R1Eca0Ins {
public:
    void R1Eca0Ins (Ptr<Tuple> link)
    {
        Ptr<Tuple> result = link;
        result->Assign (Assignor::New("P1",
            FAppend::New (VarExpr::New ("link_attr1"))));
        result->Assign (Assignor::New("P2",
            FAppend::New (VarExpr::New ("link_attr2"))));
        result->Assign (Assignor::New("P",
            FConcat::New (VarExpr::New ("P1"), VarExpr::New ("P2"))));
        result = result->Project (PATH,
            strlist ("link_attr1", "link_attr2", "P"),
            strlist ("path_attr1", "path_attr2", "path_attr3"));
        Insert (result);
    }
};
```

Listing 2.1: Compiled C++ function
Figure 2.2 depicts the workflow compiled from rule r2. Intuitively, a new path derivation may be triggered by either learning a path announcement (upper strand), or a change in links (lower strand). We will discuss in detail how to strategically choose between each strand to optimize the network consumption in Chapter 4.

2.2 RELATED WORK

In this section, we briefly discuss some related work in distributed resource management and distributed system debugging with a focus on network provenance.

2.2.1 RESOURCE MANAGEMENT

Distributed system resource management has long been an active research area. Mesos [47] proposes a fine-grained resource sharing platform in data center environments for heterogeneous concurrent computing frameworks. In Mesos, a master is placed between computing frameworks and system resources. The master proposes resource offers to framework schedulers, who then decide whether to accept or reject the proposed offer.
Hadoop YARN [88] proposes to decouple programming model from resource management. YARN separates resource management from the computing framework’s own scheduler. For each cluster, a ResourceManager monitors resource utilization and resolves any potential resource contentions. An ApplicationManager takes care of programming logics and requests system resources from ResourceManager.

Ghodsi et al. propose Dominant Resource Fairness [40] to ensure fairness in resource sharing. In this work, the authors propose DRF, a general approach to ensure fairness in multiple types of resources. DRF achieves many beneficial properties, including: sharing incentives, where users are encouraged to share idle resources, strategy proofness, where users cannot game the algorithm, envy-free, where every user is satisfied with their shares, and Pareto efficient, where no idle resources are wasted.

Borg [90] manages clusters at Google scale and aims to achieve high resource utilization. In Borg, allocated resources are reserved and remain assigned regardless of its activeness. When resources are overcommitted, higher priority jobs preemptively take their allowed quota from other low priority jobs. Such method avoids need of policies like DRF mentioned above.

Quasar [27] aims to increase resource utilization by removing resource reservation and performing classification techniques before actual scaling. Quasar takes a different approach from previous mentions works. It does not rely on resource reservations and instead require users to indicate their resource constraints. Then Quasar would use a classifier to determine what actions would lead to optimal resource allocation and then executes the resource assignments.

Xie et al. propose Time-Interleaved Virtual Clusters [96] to better model cloud application network requirement and aims to reduce tenant’s cost. The authors first profile popular network applications’ traffic pattern, and discovered that they send large portion of traffics during only 30% to 60% of the application’s execution period. So they propose a
fine-grained time varying network reservation abstraction. Their evaluation over prototype implementation shows improvements of network utilization and reduction of user costs.

Finally, we remark that Jennings et al. provide a rich survey on recent cloud resource management in [50]. We refer interested readers to their comprehensive survey.

DeDoS is conceptually related to the trend toward fine-grained granularity decomposition of functions seen in Function-as-a-Service platforms [35]. However, those platforms [1, 3, 7, 8] have often constraints that make them unsuitable for the deployment of stateful, long-lived services. Similarly, DeDoS is not a Micro-services platform. We envision that DeDoS can be integrated to those platforms.

2.2.2 DISTRIBUTED SYSTEM DEBUGGING

In distributed system debugging, some prior researches have utilized recordings of system events to reveal causalities between such events. Some of these systems, such as Friday [37], WiDS Checker [60], and those proposed in [36, 56, 83], record and replay events to show how system states have evolved into a faulty land. Others also propose tools and techniques for root cause analysis in [32, 45, 66]. Static analysis is an important technique in system debugging. Systems such as Batfish [33], NetPlumber [53], Libra [98], VeriFlow [54], rcc [31], and Header Space Analysis [52], statically analyze the network program or configuration to find the bugs and errors. Some other works, such as proposed by Scott et al. [81, 82], Zeng et al. [97], and Wundsam et al. [95], dynamically test and debug networks in real time. Another line of work [10, 15, 57, 75], formally verifies Software Defined Network programs for correctness.

2.2.3 PROVENANCE

Provenance provides an easy and secure approach to extract the causalities between system states. It is a concept initiated from the database community [14, 42, 49, 69, 79, 92], but it
has recently been applied in several other areas, including distributed systems [93, 94, 102, 103], storage systems [73, 78], operating systems [39], and mobile platforms [30]. Our work is mainly related to projects that use network provenance for diagnostics. In this area, ExSPAN [102] was the first system to maintain network provenance at scale; SNP [103] added integrity guarantees in adversarial settings, DTaP [105] a temporal dimension, and Y! [94] support for negative events. Bao et al. developed efficient schemes for labeling workflow provenance [11, 12]. Glavic et al. presented query rewriting techniques for processing provenance in [41]. Huang et al. [48] and Meliou et al. [70, 71] utilized provenance to explain reason for missing answers. Chapman et al. [17] and Tran et al. [87] provide reasons for missing answer in SQL queries.

Position papers, such as those by Hasan et al. [46] and Bertino et al. [13], highlight the need and challenges for privacy in provenance systems. Privacy concerns of scientific workflow provenance are raised by Davidson et al. [26]. We propose a solution to these challenges by performing queries on encrypted provenance graphs in a privacy-preserving manner.

There have been multiple proposals and standards for provenance models; notable examples include Open Provenance Model (OPM) [72] and PROV [43]. While these provenance models vary in their detail (e.g., OPM introduces a comprehensive model that captures the causal relationships between three types of entities: artifacts, processes, and agents), they all share a common graph-based notation in which edges in the graph represent dependency or causal relationships among the vertices. This allows for an opportunity to develop a scheme that can be generally applied to a wide range of provenance models. The proposed privacy-preserving scheme employs structured encryption for general graph representation of provenance, and therefore is applicable to any graph-based provenance model. While in this dissertation, we focus on the network provenance model, the proposed private provenance scheme can be extended to support other graph-based
provenance models by keeping additional annotations with the edges and vertices in the graph.
CHAPTER 3

SYSTEM RESOURCE OPTIMIZATION

Although the semantics of NDlog have been extensively studied and well-understood, its classic execution model (i.e., the semi-naïve evaluation) does not consider scheduling strategies that help maximize the utility of resources. In this chapter, we explore a novel approach that prioritizes resources allocation to the parts of the program that have the highest demand, such that the system is dynamically scalable with the workload. We automatically split an NDlog program into components of DeDoS, and strategically place, schedule and potentially replicate these components according to the workloads observed at runtime.

3.1 DECLARATIVE DISPERSION-ORIENTED SOFTWARE DESIGN

Figure 3.1: Example use case of Declarative Dispersion-Oriented Software. The software is built using MSUs (a), represented as a dataflow graph (b). MSUs are then scheduled on the available machines (c). When one of the components is overloaded (d), DeDoS disperses the overload by generating additional instances on other machines (e).

A DeDoS application consists of several components called minimum splittable units (MSUs) (Figure 3.1). Each MSU is responsible for some particular functionality. For
instance, an NDlog application might contain a table operation MSU, a JOIN MSU, a network MSU, etc. (Figure 3.1a).

Related MSUs communicate with each other. For instance, a tuple may enter the system at a network MSU, be inserted by a table operation MSU, and potentially joined with other tuples by the JOIN MSU. Collectively, the MSUs form a dataflow graph that contains a vertex for each MSU and an edge for each communication channel (Figure 3.1b).

Each DeDoS deployment contains a central controller which provides an API for programmers to deploy their application. The controller can either receive a pre-computed allocation, or perform an initial allocation plan to decide how many instances of each MSU should exist, and which machines they should run on, based on the requested performance requirements and available resources (Figure 3.1c).

Additionally, the controller continuously collects runtime statistics about available resources and the performance of each MSU. If it detects that some MSU instances are overloaded (e.g., due to some unexpected workload; Figure 3.1d), it can create additional instances of these MSUs, placing them on machines where resources are still available (Figure 3.1e). Thus, the service can adapt itself against varying system workload with all available resources.

3.1.1 Minimum Splittable Units

When designing an application for DeDoS, the question of defining the granularity and boundaries of MSUs arises. While smaller MSUs can result in a more precise response during high load—since it allows DeDoS to replicate only the functions that is actually overloaded—too small and numerous MSUs can result in unacceptable overheads because of the delay introduced on the execution path. This tradeoff has already been unveiled in the past with, for example, the fall of mainframe computers and the rise of microservices [34, 38].
The general approach we advocate is based on the microservices design [77]: MSU split points are appropriate when there are loose couplings between components, functional domains are clearly encapsulated, and individual components are provably stables.

For components that commonly become bottlenecks, it is also advantageous to purposefully demarcate MSUs to most optimally respond to the high load. For example, to adapt to peak loads, the JOIN operation or hashing component in NDlog could be isolated into its own MSU.

However, a key benefit of DeDoS is that it does not require a priori knowledge of the possible hotspots in the system. Hence, in many instances, programs may not be perfectly spliced. Indeed, this is our expectation and observation in practice. In such instances, MSUs may contain features unrelated to the bottleneck, resulting in non-optimal resource allocation. However, we emphasize that such duplication will always be preferable and is very likely far better than naïve replication. In general, we posit that splitting software components following a microservices-like programming paradigm will yield significant performance improvement while incurring limited overheads. We empirically measure these overheads in §3.4.

NDlog ECA Rules to MSUs Next we briefly describe how to wrap NDlog rules into MSUs. In NDlog, each rule has a very clear boundary that is a natural fit for MSU separation. Each MSU takes as input an event of a tuple that is either a local insert, local delete or remote action. The MSU then performs a series of modular operations based on the event type and tuple. At the end, the MSU executes an action that is either a local insert, local delete or remote action, which further triggers other rules. For instance, in our running example shown in 2.1, rule \( r_1 Eca0Ins \) takes as input an insertion event of a tuple, constructs a list object, and then insert the new path tuple into its local table. The execution workflow involves several assignments and one projection in a serial fashion. The
MSU boundary could choose to be as fine as between each operation, or at a coarse level that between each ECA rule. Here we shown an example of splitting between each rule. In other words, each rule is wrapped into an MSU. After ECA rewrite, all predicates in one rule’s rule body are located at the same place. So for each MSU, the rule body location determines where this MSU is placed, and rule head location determines the type of next MSU. In our example, rule \texttt{r1Eca0Ins} has a final action of adding path tuple. The corresponding next MSU will be a rule that consumes or acts upon insertion of path tuple.

### 3.1.2 Inter-MSU Communication

MSUs communicate with each other by exposing an API that can be called by other MSUs. The API functions are asynchronous and one-way. This enables efficient event-driven implementations (analogous to SEDA [91]). If a call needs to return a value, this is handled by another call in the reverse direction.

Communicating MSU instances can reside on different machines. DeDoS makes this transparent to the MSUs by injecting a bit of “glue code” that converts calls into a local function call (if the callee is on the same machine) or a network packet (if the callee is remote).

### 3.1.3 Routing Tables

When an MSU instance of type \textit{X} wants to invoke a function on another MSU of type \textit{Y}, \textit{X} does not need to know where the instances of \textit{Y} are currently located. DeDoS handles routing by maintaining a routing table, configured by the controller, which contains information about MSU types and implements customizable load balancing policies and routing functions. By default, DeDoS spreads the load evenly among MSU instances of the same type, enforcing instance affinity to related packets (e.g., from the same flow or user
session). As we show below, we can extend this policy to implement queue-length based routing.

3.1.4 DeDoS runtime API

So far, we have treated the dataflow graph as largely static. However, the controller can also dynamically create new MSU instances. To make this possible, each machine in a DeDoS deployment runs the DeDoS runtime. When it is first started, the runtime process is an empty shell: it contains the code for all the MSUs that the system could create, but none of this code is active yet. The runtime listens for commands from the controller. The add command creates a new MSU instance, while remove deletes one. Those operations involve adjusting the routing table of connected MSUs. MSU also expose an API, to execute their main function, or access their internal state. The full API is documented online [4].

3.1.5 Support for existing applications

DeDoS is a new platform which aims at improving applications’ resilience from the very beginning of their development process. Consequently, our focus is on enabling new applications using DeDoS’ model. Nevertheless, we do not require applications to be written from scratch in order to benefit from DeDoS’ defense mechanisms. We provide a proof of concept in our case study (§3.3) by splitting a webserver into MSUs. Since DeDoS does not require the entire software to be partitioned, rewriting existing code can start small by only carving out the most likely overloaded component while the rest of the application runs as a single MSU.
3.2 Resource Allocation

To ensure that the applications meet their SLAs, DeDoS needs a way to make and enforce resource allocations to MSU instances at runtime. DeDoS performs resource allocation at two layers: each machine schedules MSU instances locally based on their resource needs, whereas a central controller is responsible for decisions requiring a global view, such as cloning or merging MSU instances. To enable runtime adaption, each machine has an agent that continuously monitors local MSUs, periodically submits statistics to the controller, and is responsible for handling the controller’s commands.

3.2.1 Initial MSU Assignment

When a DeDoS deployment is first started or a new application is launched, the controller finds an initial assignment of MSU instances to machines, such that the application’s SLA goals (throughput and end-to-end latency) are met. The assignment must be feasible: each machine must have sufficient resources (e.g., memory) to execute the MSU instances that are assigned to it.

Since the controller cannot predict the effects of future overload, the computed assignment only maintains the SLAs in the absence of such event. However, the controller monitors the system at runtime, using the statistics that are submitted by the agents. If it detects that the SLAs are being violated (e.g., due to an unexpected high load), it adjusts the assignment to mitigate the spikes, using the process we describe next.

3.2.2 Cloning and Merging

The controller supports customizable cloning and merging policies. The DeDoS native heuristics (i.e., default policies) operate by analyzing the collected metrics; forming a decision about whether to clone, merge, or do nothing; and executing the chosen action.
As native policies, the controller attempts to clone an MSU if its minimum reported queue length is greater than 0. The rationale here is that the system is provisioned such that the expected arrival rate is sustained, and in such conditions, no queue should be built for any of the MSU instances. In addition, if all runtimes are utilizing more than a configurable percentage of a resource (e.g., memory) and an MSU type accounts for a configurable percentage of a runtime’s utilization of that resource, the controller will begin to clone MSUs of that type. This latter policy targets the system’s bottlenecks by increasing parallelization.

Once the decision to clone is made, the controller picks a satisfying machine for the new instance, favoring locality with the clone’s neighbors in the dataflow graph. A local machine is best to minimize network communication. Once a machine has been elected, the controller picks the least loaded core which does not already host an instance of the same type, and contacts the machine’s agent to spawn the instance. The controller also updates all the relevant routing tables to enforce the load balancing policy in place for this MSU type.

An attempt to clone will fail if it is not possible to place the clone on any available runtime or if the same type of MSU has been recently cloned.

The controller removes cloned MSU instances when they are no longer needed (e.g., when the workload drops to normal). Two conditions must be met for an MSU to be removed. First, the last runtime where a clone has been placed must report a maximum queue length of 0 in the last monitoring interval (following the rationale described earlier); second, the MSU type must not be significantly contributing to more than a configurable percentage of a resource consumption on any runtime. An attempt to remove will fail if an attempt to clone an MSU of that type was made in the recent past (to protect against system oscillation), or if some configurable amount of time has passed since the last removal of that type.
3.3 Case Studies

We first introduce a Packet Processing application written in NDlog to demonstrate the feasibility of deploying NDlog to DeDoS. Then we describe a non-declarative Webserver application in DeDoS to show that DeDoS is not restricted to declarative language.

3.3.1 Packet Processing in NDlog

As an initial exploration, we implement an application that does routing (packet forwarding) written in NDlog. The application is a one rule NDlog program

\[ r1 \text{ packet}(\@Y,A,\text{Data}) :- \text{packet}(\@X,A,\text{Data}), \text{neighbor}(\@X,A,Y). \]

In this rule, all packets arriving at X are forwarded to neighbor Y based on some attribute A (e.g., the packet’s header data). \text{neighbor} is a local materialized table and \text{packet} is an event that triggers \( r1 \) execution.

Since all predicates in rule body are located at X, localization rewrite simply keeps the rule unmodified. ECA rewrite is also straightforward, due to the existence of the unmateri-
alized predicate \text{packet}. The generated ECA rule is

\[ r1\text{eca} \text{A\_SEND<packet}(\@Y,A,\text{Data})> :- \text{E\_RECV<packet}(\@X,A,\text{Data}>>,
\text{neighbor}(\@X,A,Y). \]

It specifies that in the event of \text{packet} receive, JOIN with \text{neighbor} and forward the \text{packet} to a new destination.

We wrote one \text{Network MSU}, one \text{Processing MSU}, and one \text{Destination MSU} for rule \( r1\text{eca} \). The \text{Network MSU} is responsible for taking \text{packet} off the network and enqueueing it to \text{Processing MSU}. \text{Processing MSU} first JOINs \text{packet} with \text{neighbor} to find where the packet should be forwarded. Then it PROJECTs the JOIN result to final output \text{packet}. And finally it enqueues the output \text{packet} to \text{Destination MSU}.
3.3.2 Web Server

DeDoS also supports non-declarative applications written in general C/C++ language. We implemented a simple web server constructed as five MSUs. The I/O MSU accepts incoming requests and steers them toward the Read MSU, which performs the TLS handshake, deciphers data, and relays plaintext to the HTTP MSU. The HTTP MSU implements NodeJS’s HTTP Parser [51]. Once the request is parsed, the HTTP MSU issues a call to the database tier to retrieve some object file, then enqueues the request to a Regex Parsing MSU, which uses the PCRE engine to parse it. The final HTTP response is sent to the Write MSU, which wraps it in a layer of TLS and sends it back to the client. We use OpenSSL version 1.0.1f for TLS support in both Read and Write MSUs.

Importantly, with the exception of the I/O MSU, all of the web server’s MSUs are event-driven and non-blocking. We favor non-blocking MSUs to augment the overall utilization of our machines. To avoid having to migrate socket states between machines, we configured the controller to enforce that the I/O, Read, and Write MSUs reside within the same DeDoS instance for a given client connection. We use HAProxy [86] as a front-end load balancer, allowing us to direct incoming client connections to any I/O MSU on any DeDoS instance. Our web server leverages DeDoS’ fine-grained modular architecture to mitigate unexpected high load.

3.4 Evaluation

In this section, we aim to answer two high-level questions (1) does running applications on DeDoS incur reasonable overheads, and (2) how well can DeDoS improve the performance under high load compared to whole-system replication? Our experimental testbed consists of a cluster of 8 machines connected via a 10 Gbps switch in a star topology. Each machine
Table 3.1: Network Datalog packet processing application overheads.

<table>
<thead>
<tr>
<th>Clients</th>
<th>DeDoS Throughput</th>
<th>Standalone Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>491.6 pkts/s</td>
<td>491.9 pkts/s</td>
</tr>
<tr>
<td>10</td>
<td>980.0 pkts/s</td>
<td>981.2 pkts/s</td>
</tr>
</tbody>
</table>

has 8 1.80 GHz cores (with hyperthreading and DVFS disabled), 64 GB of memory, and runs Linux kernel 4.4.0-62.

3.4.1 OVERHEADS

To measure the overhead introduced by the DeDoS runtime, we run the Packet Processing NDlog application described in §3.3 within and outside of DeDoS on a single server machine.

We use either five or ten clients that each send 100-byte packets at a rate of 100 packets per seconds. We compare a DeDoS-enabled application to a standalone version that runs the same code, but outside DeDoS. Table 3.1 shows our results: the throughput with DeDoS is only marginally lower (by less than 1%) than the throughput of the standalone system.

3.4.2 PERFORMANCE

We setup the Packet Processing application described in §3.3 and initialized it with a large in-memory neighbor table, rendering naïve replication too expensive in this case. Our workload consists of a varying number of clients that forward packets via our application. We increase the work load by using more clients to send more traffic. Figure 3.2 shows the throughput (pkts/s) that can be processed by (i) a standalone implementation, (ii) a DeDoS-enabled application with cloning disabled, and (iii) a normal DeDoS-enabled application.
The results show that standalone and DeDoS achieve comparable throughputs (which indicates low overhead), but that cloning enables DeDoS to handle roughly twice as many clients during high work load.

3.4.3 VARYING HOTSPOTS

We also show that when the overloaded hotspot migrates in the system, DeDoS is able to react to such movement by dynamically cloning and merging different types of MSUs.

We configure our testbed as follows: the webserver is deployed on three machines while three other machines run instances of an in-memory database. All web requests access the database, and HAProxy is used to distribute HTTP and database requests. The remaining two machines are used to generate traffics. To demonstrate DeDoS’ ability to handle varying system hotspots and reclaim resources, under dynamic patterns, the experiment runs for two hours and different components are being overloaded during different periods.
Figure 3.3 shows our main findings for different component overloading on the HTTP servers for a single run of the experiment that lasts 2 hours. The top figure shows response times for successful connections averaged every second, while the middle figure presents the connection success rate during this experiment. The bottom figure shows the number of MSU instances of a given type deployed on the system over time. High load occurs during the period colored in red. We compare DeDoS to two other approaches: (1) an approach that does not replicate at all (“standalone”), and (2) an approach that naively replicates an entire webserver to one of the database servers (“naïve”). Initially, the DeDoS’ webserver has 4 Read and 2 Regex MSUs on each of the three starting machines.

During the entire course of the experiment, DeDoS is auto-piloting without inputs from human users. We observe that DeDoS can accurately detect and react to the unexpected high load based on the resource allocation polices described in §3.2 without a-priori knowledge of the workload. DeDoS can consistently and automatically decide on an effective strategy for different types of loads. Figure 3.3 shows that DeDoS consistently outperforms standalone and naïve approaches, and sustains low latency and high response rate while standalone and naïve can only provide limited or sporadic services.
The experiment tries to overload three different components in the system: TLS, Regex and File Descriptor pool. TLS component gets overloaded when there are unexpected high number of incoming TLS requests, which consumes large amount of CPU resources. DeDoS gradually increases the number of READ MSUs to prevent from being overwhelmed. Regex component is also CPU hungry during high demand of complex regular expression matching. The DeDoS controller successfully detects such high load and allocates more Regex MSUs to support the system. To overload the File Descriptor (FD) pool, the clients set up long-lived connections to the server, so that the file descriptor pool for new incoming connections is exhausted. DeDoS is able to spawn new READ MSUs on machines with available FDs to sustain more connections.

3.5 Summary

DeDoS proposes a new approach to manage system resources in a fine-granularity. In DeDoS, software is built as a set of functional units called Minimal Splittable Units (MSUs) that can be replicated independently when under high load. DeDoS allows for more flexible allocation of resources and can efficiently dedicate more resources to MSUs that are more in need. Our evaluation shows that DeDoS runs with minimal overheads and is able to dynamically redistribute available system resources effectively.
Chapter 4

Network Resource Optimization

With the booming developments of application scenarios in distributed systems in recent years, such as Big Data and Artificial Intelligence, the amount of data that needs to be shipped across network for applications increases dramatically. Network communication thus is often considered as the bottleneck for such applications and systems. Since NDlog programs typically run in a distributed fashion, one particular challenge arises when the required input data for an MSU are scattered at different physical locations. The data need to be shipped to one location, namely where the MSU resides, for processing. It is crucial that the MSU is strategically placed such that we can minimize the communication cost. We present our proposed solution by reducing the optimal MSU placement problem to a min-cut problem on constructed query graphs that depict the execution logic of a given NDlog program.

In this chapter, we first describe the challenge and inefficiency in current systems, and then we propose our solution to minimize communication overhead.

4.1 Network Inefficiency

As briefly described in §2.1.1, the localization rewrite subroutine during NDlog compilation may result in unnecessary data movement across the network. We illustrate such scenarios with our running example to motivate a better operator placement strategy.

Current implementation of RAPIDNet uses a semi-naïve evaluation that does not consider scheduling strategies. One consequence is that when joining two tables that are located
at different places, the JOIN operator is placed at the location decided by the JOIN key. For example, rule $r_2$ tries to correlate link tuples stored on node S and path tuples stored on node Z. This is captured as a JOIN between the two tables. Here, we need to decide where we should perform the JOIN. We have two options as shown in Figure 4.1.

![Figure 4.1: Two options for evaluating $r_2$. (a) Ship link tuples from S to Z, join with path tuples at Z, then ship results back to S. (b) Ship path tuples from Z to S, join with link tuples at S, then insert results locally to S.](image)

![Figure 4.2: A Min-Cut based solution to minimize communication cost.](image)

In option A, we draw updates of link table from the DB at node S, and send them to node Z. They will JOIN with the path table to calculate longer paths, which need to be sent back to node S so that S learns these longer paths. Alternatively, we can send path tuples to node S as they become available, and perform the JOIN at node S. Between these two options, option B incurs less communication cost, as it only needs to send the path table, instead of both the link and path tables in option A.

More generally, we would prefer to move computation, more specifically MSUs, to where data locates, as the computation cost is much smaller than the cost of shipping data. We consider how we can minimize cross-node communication costs when we schedule the
MSUs. To solve the problem, we propose a min-cut based graph algorithm to minimize the communication cost for NDlog programs. The algorithm is inspired by a prior work that reduces network bandwidth consumption for multi-query evaluation [58].

We can abstractly construct the query graph that depicts the distributed JOIN of the link and path tables as in Figure 4.2. Note that it draws data from databases on both node S and node Z. The communication costs in option A and option B effectively correspond to “cuts” of the query graph, and we search for the minimal cut that corresponds to the minimal communication cost.

4.2 PROPOSED SOLUTION

Figure 4.3: Overview of proposed optimization.

We propose a solution that optimizes network communication overhead by utilizing a novel min-cut based algorithm on query graphs constructed from the NDlog rules.

Our proposed solution involves several steps shown in Figure 4.3. First, a Profiler collects cardinality statistics from all nodes in the system. These statistics, combined with the network topology and the NDlog program, are forwarded to an Optimizer. The Optimizer constructs two sets of graphs: Query graph and Communication graph, which we
will discuss shortly in §4.2.1. These two sets of graphs are then fed into a Min-cut based algorithm that decides the optimal MSU placement that minimizes potential network communications. The generated MSU placement overrides the original Localization rewrite. ECA rewrites remain unchanged.

Next, we describe the two sets of graphs constructed by Optimizer.

4.2.1 Query and Communication Graphs

Each NDlog rule can be considered as an execution of a query, where the rule head is the result and rule body is the query. For instance, rule $r_2$ specifies the following continuous distributed query

```
FROM link, path
WHERE link.Z = path.Z
```

Figure 4.4: Query graph and communication graph. (a) $G_{q_1}$ and $G_{q_2}$ for our example (b) Comm Graph $G_c$

The Optimizer first constructs a query graph $G_{q_i}$ for each rule. $G_{q_2}$ for rule $r_2$ is shown in Figure 4.4(a). The JOIN operator pulls link from S and path from Z, and sends results to S. Then the Optimizer constructs a communication graph $G_C$ from the network topology. In $G_C$, each vertex is labeled with node ID and each edge is optionally weighted by the link costs, as shown in Figure 4.4(b).
With the two sets of graphs, we next describe a Min-cut based algorithm that determines the optimal MSU placement to minimize network communications.

### 4.2.2 MIN-CUT BASED ALGORITHM

At high level, the Min-cut based algorithm involves three steps:

- Merge all Query Graph $G_q$s to construct a graph $G_h$.
- For each edge in $G_C$, determine which data are shipped across that edge by running an instance of Min-Cut.
- For each internal vertex in $G_q$, determine where the corresponding operator is placed in the network.

![Figure 4.5: Hyper graph and labeling.](image)

**Figure 4.5**: Hyper graph and labeling. (a) Graph $G_h$ (b) Node labeling for rule $r2$

Step 1 involves merging all $G_q$s into one graph $G_h$ by *fusing* same node across Query Graphs. The reason behind constructing such a graph is that a program has multiple rules, and we want to consider all those rules at the same time to find a global optimal (instead of local optimal for each rule). For instance, as shown in Figure 4.4(a), $G_{q_1}$ and $G_{q_2}$ share one same input source *link*. To capture this fact, we merge $G_{q_1}$ and $G_{q_2}$ by *fusing* *link* into one node. The result is the graph shown in Figure 4.5(a).
In Step 2, for each edge $(u, v)$ in the Communication Graph $G_c$, we label each node in $G_h$ with $u$ if the node belongs to the same connected component as $u$ after removing edge $(u, v)$ in $G_c$, and vice versa. In our example, for edge $(S, Z)$, the labeled graph is shown in Figure 4.5(b). For edge $(Z, D)$, all nodes are labeled with $Z$ since the two input tables are both on the “same side” of $(Z, D)$. Next we add vertices $u$ and $v$ into the labeled graph serving as source and sink of the Min-Cut operation. Before running Min-Cut, we connect $u$ to all vertices labeled as $u$ with infinite cost edges. And the same goes for $v$ and vertices labeled as $v$, as shown in Figure 4.6(a). Then we run Min-Cut on the graph and label remaining vertices with either $u$ or $v$ depending on which part it resides in. In our example, for edge $(S, Z)$, the constructed graph and Min-Cut results are shown in Figure 4.6.

The previous two steps have determined if a table or intermediate result should be sent across each edge in $G_c$. Step 3 combines the local results and makes the final global decision on where each intermediate result is placed. For each internal node in the graph $G_h$, we construct a directed graph $G_j$ by first adding all vertices from $G_c$, then for each edge $(u, v)$ in $G_c$, if the internal node was labeled $u$, add a directed edge from $v$ to $u$, and vice versa. This captures the movement of the intermediate result. When finished, graph $G_j$ will have one vertex that has 0 out degrees. It means that every iteration of Min-Cut agrees
on putting this intermediate result in this vertex. In other words, the operator that generates this intermediate result is placed at the vertex with 0 out degree.

Min-Cut on Hypergraph  In our example, when fusing two link nodes, since only one of the link node is involved in a JOIN operation, i.e., link for $G_{q2}$, the new fused link node only has one parent node in new graph, i.e., link $\bowtie$ path. If a node has multiple parents, then after fusing, the new edge would contain more than 2 vertices. In such case, $G_h$ would become a hypergraph. We briefly describe how to handle Min-Cut on hypergraphs in Figure 4.7.

A hypergraph contains vertices and hyperedges, where each hyperedge contains multiple vertices, as opposed to only 2 vertices in normal edge. Hyperedge costs are associated with the edge itself. To perform Min-Cut on a hypergraph, we first add two new vertices into the graph for each hyperedge, noted as $X$ and $Y$. And also create a directed edge from $X$ to $Y$. This edge has cost equal to the cost of the hyperedge. Then for all vertices $v$ in the hyperedge, create two directed edges $(v, X)$ and $(Y, v)$. These edges have cost of infinity. This transformation captures the equivalency between cutting the hyperedge and cutting edge $(X, Y)$, since we will never cut edges with infinity costs. Then we can run Min-Cut on the newly generated graph to obtain the set of edges belong to the min cut.
4.2.3 Definitions

We provide some formal definitions of the terms that we mentioned earlier.

- Communication Topology: A Communication Topology $G_C = (N, L)$ is an undirected weighted graph that represents the underlying physical network topology. We conventionally use node $n \in N$ and link $l \in L$ to denote the physical machines and network connections in $G_C$. Cost of sending 1 unit of data across link $l$ is $\text{cost}_l$, which is default to 1 if not explicitly defined.

- NDlog Program: An NDlog Program $P$ consists of a set of NDlog rules $r_i$, that represents the program execution logic. Each rule involves operating over one or more tables $t$.

- Size Estimate: For each materialized and intermediate table $t$, $\text{size}_t$ represents the estimated size of $t$. Sending $t$ across link $l$ incurs network cost of $\text{size}_t \times \text{cost}_l$.

- Query Graph: A Query Graph $G_{Q_i}^r = (V_i, E_i)$ is a directed acyclic graph that represents the execution of a rule $r_i$. We conventionally use vertex $v \in V_i$ to denote the base relations and intermediate results during rule execution, and edge $e \in E_i$ to denote the derivation between vertices in Query Graph. Each vertex $v \in V_i$ is associated with a label $b \in N$ that indicates where it is located in $G_C$.

- Hyper Graph: A Hyper Graph $G_H = (V_H, E_H)$ is a graph constructed by merging all $G_Q$’s, where $V_H = \bigcup_{v \in V_i} v$ and $E_H = \bigcup_{e \in E_i} e$.

- Label: Label $B : V_H \rightarrow N$ is a many-to-one mapping that maps vertices $V_H$ in $G_H$ to nodes $N$ in $G_C$.

- Labeled Hyper Graph: A Labeled Hyper Graph $G_L = (V_H, E_H, B)$ is a hyper graph that its vertices are labeled with corresponding physical location.
• Operator Placement: An Operator Placement $O : V_H \rightarrow N$ is a many-to-one mapping from $V_H$ of $G_H$ to $N$ of $G_C$ that determines the physical location of each $V_H$. $O(v) = n$ indicates that table represented by $v$ is located at $n$.

• Network Cost: Network Cost $C$ is the total data sent across network between nodes during $P$’s execution. $C = \sum_{l \in L} size_t \times cost_t$ where $l = (w, u), O(t) = w, O(t') = u$, and $(t, t') \in E_H$.

Optimal Operator Placement Problem We formally define Optimal Operator Placement problem. Given an NDlog program $P$, a Communication Topology $G_C$, and Size Estimates $t_c$’s, solving the Optimal Operator Placement problem yields an Operator Placement $O$ such that the Network Cost $C$ is minimized.

Min-Cut for Minimizing Communication Cost To solve the Optimal Operator Placement problem, we propose $MC2$, a Min Cut based algorithm to Minimize Communication Cost.

---

**Generate Query Graph**

1: function QueryGraph($r$)
2: $V \leftarrow \emptyset$, $E \leftarrow \emptyset$, $v_{int} \leftarrow None$
3: for rule head, $V \leftarrow V + v_h$
4: for each table $t \in r$ rule body do:
5: $V \leftarrow V + v_t$
6: if $v_{int}, v \leftarrow (v_{int} \bowtie v_t), V \leftarrow V + v$, $v_{int} = v$
7: else, $v \leftarrow v_t, V \leftarrow V + v$, $v_{int} = v$
8: $E \leftarrow E + (v_t, v_{int})$
9: $E \leftarrow E + (v_{int}, v_h)$
10: return $G_Q = (V, E)$

Figure 4.8: Generate query graph.

The general steps of $MC2$ algorithm is as follows:
Construct Hyper Graph

1: function HyperGraph(G_Q(QueryGraph(r_{r \in P}))
2: \( V \leftarrow \emptyset, E \leftarrow \emptyset \)
3: for each \( G_Q = (V_Q, E_Q) \in GQS \):
4: \( \text{for } v \in V_Q \)
5: \( \text{if } v \notin V, V \leftarrow V + v \)
6: \( \text{for } e = w, u \in E_Q \)
7: \( \text{if } w \in V \text{ and } u \in V, E \leftarrow E + e \)
8: return \( G_H = (V, E) \)

Figure 4.9: Construct hyper graph.

1. For each rule \( r_i \) in \( P \), construct a Query Graph \( G_Q^{r_i} \) using algorithm shown in Figure 4.8. We assume the join order of query execution is made available by some other query optimizer.

2. Construct Hyper Graph \( G_H \) by merging \( G_Q \)’s into one graph as shown in Figure 4.9.

3. For each link \( l = (u, v) \) in the Communication Topology \( G_C \), we construct a Meta Graph \( G_M = (V_H \cup \{u, v\}, V_E \cup \{(u, w) | w \in V_H(u)\} \cup \{(v, x) | x \in V_H(v)\}) \).

4. For each constructed Meta Graph \( G_M = (V_M, E_M) \), we run a subroutine \( mc \) that finds the min-cut in \( G_M \) such that \( u \in V_u, v \in V_v, V_u \cap V_v = \emptyset, \) and \( C = \sum c_{xy}(x \in V_u, y \in V_v, (x, y) \in E_M) \) is minimum.

5. Label each \( v' \in V_H \) according to the min-cut to yield the mapping \( v' \rightarrow v \) or \( v' \rightarrow u \) as shown in Figure 4.10.

4.2.4 Optimality

We argue that the proposed MC2 algorithm solves Optimal Operator Placement problem.
Label on Hyper Graph

| 1: function \( LHG(G_H, l = (w, u)) \) |
| 2: \( B : V_H \rightarrow \emptyset \) |
| 3: for \( v \in V_H \): |
| 4: \( \text{if } L(v) \in G^w_C \), \( B(v) = w \) |
| 5: \( \text{if } L(v) \in G^u_C \), \( B(v) = u \) |
| 6: \( V_H \leftarrow V_H + v_w, v_u \) |
| 7: for \( v \in V_H \): |
| 8: \( \text{if } B(v) = w, E_H \leftarrow E_H + (v_w, v) \) |
| 9: \( \text{if } B(v) = u, E_H \leftarrow E_H + (v_u, v) \) |
| 10: \( E_C \leftarrow mc(G_H) \) |
| 11: \( E_H \leftarrow E_H - E_C \) |
| 12: for \( v \in V_H^w \), \( B(v) = w \) |
| 13: for \( v \in V_H^u \), \( B(v) = u \) |
| 14: return \( B \) |

Figure 4.10: Label on hyper graph.

Given a min-cut solution such that \( MC = \sum c_{xy}(x \in V_u, y \in V_v, (x, y) \in E_M) \) is minimum, we have a label \( B \) that maps each relation \( t \) to a node \( N \). The incurred network cost during program execution is \( NC = \sum_{l \in L} \text{size}_t \times \text{cost}_l \), where \( l = (w, u), O(t) = w, O(t') = u \), and \( (t, t') \in E_H \).

Suppose there’s another label \( B' \) that maps relations differently than \( B \) and incurs network cost \( NC' < NC \). Assume \( B(t) = u \) and \( B'(t) = v \), where \( t \in V_H \) corresponds to a table.

The difference of network cost between \( B \) and \( B' \) \( D = NC - NC' \) is caused by shipping table \( t \) to node \( v \) rather than to \( u \). \((u, v)\) is a physical edge in Communication Topology \( G_c \). Since \( t \) is shipped to \( v \), all other tables that will join with \( t \) need to be present at \( v \) as well, either shipped to \( v \) or originally at \( v \). And the join results \( r \) may potentially need to be shipped back to \( u \). On the other hand, if the join results should ship to \( v \) at the first place, then it saves shipping \( r \) from \( u \) to \( v \). Let \( S_v \) and \( O_v \) denote tables that shipped
to \( v \) and originally located at \( v \) respectively. Let’s further assume the link cost between \( l = (u, v) \) is \( c_l \). Hence shipping \( S_v \) to \( v \) causes \( S_v \cdot c_l \) more network communication and avoid shipping \( O_v \) to \( u \) saves \( O_v \cdot c_l \) less bandwidth.

In the case where \( r \) should ship to \( u \), it causes more bandwidth by \( r \cdot c_l \). Thus \( NC - NC' = O_v \cdot c_l - (S_v \cdot c_l + r \cdot c_l) > 0 \). This corresponds to a cut in \( G_H \) for \((u, v)\) such that join happens at \( v \) rather than \( u \). If \( O_v \cdot c_l - (S_v \cdot c_l + r \cdot c_l) > 0 \) then it contradicts with our assumption that \( MC \) is a min-cut.

In the case where \( r \) should stay at \( v \), it saves bandwidth by \( r \cdot c_l \). Thus \( NC - NC' = (O_v \cdot c_l + r \cdot c_l) - S_v \cdot c_l > 0 \). It also contradicts with the assumption that \( MC \) is a min-cut.

4.2.5 Rule Rewrites Revisited

Once we have determined where each intermediate result and thus each operator is placed, we next describe how each rule is rewritten to reflect our new optimal placement decision. For rules where all predicates in the rule body are located at the same node (e.g. \( r1 \) in our example), the localization and ECA rewrites described in §2.1.1 remain unchanged.

For other rules that require tables to be sent around, we modify the localization rewrites to dictate that all tables to be shipped to the location where the intermediate result and the corresponding operator are placed. In our example, rule \( r2 \) will be rewritten into the following two rules:

\[
\begin{align*}
\text{r21 } & \text{rpath}(\@S, Z, D, P) :- \text{path}(\@Z, D, P), \text{placement}(\@Z, S). \\
\text{r22 } & \text{path}(\@S, D, P) :- \text{link}(\@S, Z), \text{rpath}(\@S, Z, D, P2), P=\text{concat}(S, P2).
\end{align*}
\]

Rule \( r21 \) will send \text{path} tuples at node \( Z \) to node \( S \), and rule \( r22 \) \text{JOINs} incoming \text{path} and \text{link} tuples at node \( S \). Note that we retrieve \( S \) from a table \text{placement}, which is the \( \text{path} \) tuple are sent as a \text{rpath} tuple, since they have different scheme, we use another tuple to trigger rule \( r22 \).
output of the Optimizer. It determines whether we should send path to S or not. Since the Optimizer has global view of \( G_c \), it is able to determine the destination of the tuple.

Now the JOIN results are locally available at S, so they can be directly inserted at S, without sending over the network as compared to the original rules. We would like to point out that rule \( r_{21} \) involves a local JOIN operation in order to determine the destination of the path tuple. This is a computation cost that we pay and it aligns with our assumption that communication cost is more often the bottleneck compared to computation.

Next we describe how localized rules are transformed during ECA rewrites. At a high level, if a rule contains unmaterialized predicate (or events), it will only generate one ECA rule; otherwise, for each materialized tuple, we generate two rules, one for insertion and the other for deletion. For instance, in our example, rule \( r_1 \) has one predicate that contains a materialized table, so it generates the following delta rule:

\[
\text{r1Eca0} \quad \Delta \text{path}(\$S, D, P) :- \Delta \text{link}(\$S, D), P=[S,D].
\]

This delta ECA rule indicates that in the event of link insertion (deletion), adding (invalidating) the corresponding path tuple.

For rules that have different head location and body location, a remote insert/delete event is generated to trigger corresponding operation. Rule \( r_{21} \) falls into this category. The following two rules act on incoming remote events.

\[
\text{r21Eca0Remote} \quad \Delta \text{tpath}(\$S, Z, D, P) :- \text{RECV}\Delta \text{r21tpath}(\$S, Z, D, P).
\]

Each event receive is then paired with potentially multiple action send, depends on the number of materialized predicates, similar to what happened to rule \( r_1 \). So the event \( \Delta r_{21tpath} \) is triggered by two other rules, one for \( \Delta \) of path, the other for \( \Delta \) of link, as follows:

\[
\text{r21Eca0} \quad \text{REMOTE_SEND}<\Delta r_{21tpath}(\$S, Z, D, P)> :- \Delta \text{path}(\$Z, D, P), \text{link}(\$Z, S).
\]

\[
\text{r21Ecal} \quad \text{REMOTE_SEND}<\Delta r_{21tpath}(\$S, Z, D, P)> :- \Delta \text{link}(\$Z, S),
\]
path(@Z, D, P).

For rules that have same head location and body location, there is no need for remote events. So depending on how many materialized predicates are in the rule body, multiple delta rules for rule head are generated. In our example, rule \( r^{22} \) falls into this category. Since there are two materialized predicates in the rule body, we have in 2 delta rules for \( r^{22} \), as following:

\[
\begin{align*}
\text{r22Eca0} & \quad \Delta \text{path}(@S, D, P) :- \Delta \text{link}(@S, Z), \text{tpath}(@S, Z, D, P2), \\
& \quad P=\text{concat}(S,P2). \\
\text{r22Eca1} & \quad \Delta \text{path}(@S, D, P) :- \Delta \text{tpath}(@S, Z, D, P2), \text{link}(@S, Z), \\
& \quad P=\text{concat}(S,P2).
\end{align*}
\]

Next we discuss how these rules shown above are translated to MSUs that can be deployed on DeDoS.
4.2.6 Compile ECA Rules to DeDoS MSUs

Figure 4.11 provides a bird’s eye view of MSU placements, flow graph, and a brief description of their operations. MSUs inside left rounded rectangle are located at S, and others located at Z. Arrows between MSUs represent the enqueue operation, where an upstream MSU enqueues a message into a downstream MSU’s message queue. A downstream MSU dequeues from its own message queue and process the input. Note that we added some MSUs to handle table insert/delete operations. This MSU workflow and placement graph corresponds to the optimal solution shown in Figure 4.2.

Each ECA rule is compiled to an MSU instead of a function. Each MSU consists of a receive function that implements the main functionality and an MSU struct definition that is required for the DeDoS system.

To compile the receive function, we made some modifications to the RAPIDNET compile process. We first change the function parameter and return type to adapt to the DeDoS runtime. Each receive function takes a pointer to itself and a pointer to the incoming MSU message. It’s now a static function and returns Int. We then add a function in the beginning that constructs an MSU message to a RAPIDNET tuple. Then the rest of the receive function stays roughly the same as generated from RAPIDNET compilation.

We also generate the MSU struct definition that defines DeDoS behaviors of the MSU, such as initialization, destroy, and most importantly, routing. For our new MSUs, we added a routing function that decides where to enqueue the message based on RAPIDNET tuple destination attributes. We specify the type and runtime of next hop MSU in the function, such that the messages are correctly enqueued to a location that reflects our intention to save network bandwidth.
All ECA MSUs could be cloned should it deem necessary. These MSUs are stateless which means they could replicate when overloaded. We discuss some types of MSUs that are not replicable in next section. They are unique and dedicated to each runtime.

4.3 IMPLEMENTATION AND EVALUATION

In this section we discuss some interesting aspects of our implementation and evaluation. We implement the optimization and compilation process in RAPIDNET compiler. The optimization process can also run as a standalone program for re-optimization and benchmarking.

4.3.1 OPTIMIZATION PROCESS

As shown in Figure 4.3, the optimizer takes as input the topology file and the NDlog program. For each physical link in the topology and for each distributed join in the program, it decides where the JOIN happens. Let’s consider the following NDlog PathVector program.

\[\begin{align*}
\text{r1} & \quad \text{path}(\text{@S}, \text{D}, \text{C}, \text{P}) : \text{link}(\text{@S}, \text{D}, \text{C}), \text{P}=[\text{S}, \text{D}]. \\
\text{r2} & \quad \text{path}(\text{@S}, \text{D}, \text{C}, \text{P}) : \text{link}(\text{@S}, \text{Z}, \text{C1}), \text{bestPath}(\text{@Z}, \text{D}, \text{C2}, \text{P2}), \\
& \quad \quad \quad \text{P}:=\text{concat}(\text{S}, \text{P2}), \text{C}:=\text{C1}+\text{C2}, \text{f}_\text{member}(@\text{S}, \text{P2})==0. \\
\text{r3} & \quad \text{bestPath}(\text{@S}, \text{D}, \text{a}_\text{MIN}<\text{C}>, \text{P}) := \text{path}(\text{@S}, \text{D}, \text{C}, \text{P}).
\end{align*}\]

For a physical link \((X, Y)\), for rule r2, we need to make two decisions: where does JOIN of \(X_{\text{link}}\) and \(Y_{\text{bestPath}}\) happen and where does JOIN of \(X_{\text{bestPath}}\) and \(Y_{\text{link}}\) happen.

For the first JOIN, the JOIN results are sent to \(X\). So the min-cut boils down to comparing shipping \(X_{\text{link}}\) to \(Y\) and results back to \(X\), versus shipping \(Y_{\text{bestPath}}\) to \(X\). The decision depends on several factors, including sizes of tables, selectivity of JOIN, and physical edge communication cost. Let \(S_{\text{Loc}}\text{-Table}\) denote the size of Table at Loc, \(N_{\text{Loc}}\) denote the
number of neighbors of Loc, $C_{X\rightarrow Y}$ denote the cost of sending one unit from $X$ to $Y$. Note that it could be different from $C_{Y\rightarrow X}$. We further assume there are $V$ nodes in the topology.

We use $N - 1$ as an estimate for bestPath table size for both nodes, and use $N_X$ and $N_Y$ as estimates for link table size. For JOIN selectivity, we choose $\frac{(N_Y - 1)}{N_Y}$ since out of $Y$’s neighbors, $X$ will use $N_Y - 1$ best paths to reach other nodes. Thus if we choose to JOIN at $Y$, the network cost will be

$$N_X \cdot C_{X\rightarrow Y} + N_X \cdot (V - 1) \cdot \frac{(N_Y - 1)}{N_Y} \cdot C_{Y\rightarrow X}$$

Otherwise the network cost will be

$$(V - 1) \cdot C_{Y\rightarrow X}$$

We could further include the size estimate of each tuple. Let $S_{Table}$ denote the average tuple size of Table. $S_{link}$ is a fixed number since all link tuples are of same length. For bestPath, since it varies by the path length, we could choose half of the topology diameter as an estimate. Hence the network costs will become

$$N_X \cdot C_{X\rightarrow Y} \cdot S_{link} + N_X \cdot (V - 1) \cdot \frac{(N_Y - 1)}{N_Y} \cdot C_{Y\rightarrow X} \cdot S_{bestPath}$$

and

$$(V - 1) \cdot C_{Y\rightarrow X} \cdot S_{bestpath}$$

respectively. The optimizer chooses the smaller decision to reduce network cost. The other JOIN decision could be made with similar logic.

4.3.2 ADDITIONAL MSUs

In order to enable MSUs to communicate, we additionally add three types of MSUs: Database MSU, Demux MSU, and NDlog Socket MSU.
**Database MSU**  The MSUs generated from ECA rules execute functions for the rule, but do not maintain local data. Since database operations will trigger further actions, we choose to have a centralized database maintained by a dedicated Database MSU. The Database MSU reads local placement files into its tables on initialization, and receives messages from ECA MSUs that contains DB operations. After operations are finished, the Database MSU sends triggers to Demux MSU that distributes to corresponding ECA MSUs.

**Demux MSU**  We also implement a Demux MSU to distribute incoming messages of a runtime to intended local ECA MSUs. Demux MSU implements the demux function in RAPIDNET applications. It reads the incoming tuple’s action and type, and enqueues to the correct next MSU in the flow.

**NDlog Socket MSU**  For a client to be able to send base tuples and start program execution, i.e. sending link tuple to nodes, we implement an NDlog Socket MSU that handles incoming network packets on a dedicated port. It receives the network message and construct a corresponding tuple and then enqueues the tuple to Database MSU for insertion. NDlog Socket MSU enables NDlog applications in DeDoS to communicate with the outside world. The communication handled by NDlog Socket MSU is different from inter-runtime network communications which are used for MSU messaging.

Figure 4.12 shows the three additional MSUs and the enqueue flow for inter and intra runtime. NDlog Socket MSU only enqueues to Database MSU, which only enqueues to Demux MSU. Demux MSU enqueues to ECA MSUs and ECA MSUs enqueue to Database MSU and Demux MSUs, both locally and remotely. The dashed arrows represent inter-runtime network communications.
4.3.3 Evaluation

We evaluate the Efficiency and Effectiveness of our optimization for network communication cost. During the optimization, we assume unit physical network edge cost and tuple size for simplicity. As mentioned §4.3, it is straightforward to include different network edge cost and tuple sizes.

We randomly generate network topologies with the GT-ITM topology generator [44] using Transit-Stub type networks. The topology sizes range from 50 to 500. On average, each stub domain has 3 nodes, each transit node connects 8 stubs, and each transit domain has 2 nodes. We vary the number of transit domains to change the topology size.

We first show in Figure 4.13 that our optimizer is efficient. The standalone optimizer is able to decide the JOIN locations for PathVector program on a 500 node topology within 1 second on average. This enables the possibility of realtime and dynamic re-optimizations.
Figure 4.13: Optimization latency.

during execution. The latency also exhibits a linear increase with topology sizes, which demonstrates the scalability of our optimizer. To be more precise, the optimization latency increases with the number of physical edges in the topology. Since we use transit-stub networks for our topology, the number of edges in the graph grows linearly with topology sizes. The complexity of the application also affects the optimization latency. For each physical edge, we construct a query graph and perform min-cut. The complexity of the query graphs is application dependent. *PathVector* has a nice property where its *link* table is only joined with other tables that are located on its direct neighbors. While this is a property we would find in many applications, it is not universally applicable.
Next we examine the effectiveness of our optimization. Figure 4.14 shows the number of tuples saved from shipping across the network, compared with RAPIDNET’s naïve JOIN location decisions. We could see that the savings increase exponentially with topology sizes. Combined with previous results, it shows that our optimizer could achieve exponential reduction in network communication cost with linear time latencies.

To further illustrate how much network communication is reduced relatively to RAPIDNET’s naïve decisions, we plot number of tuples to be sent in two scenarios along with saved tuples. Figure 4.15 shows that the number of saved tuples are more than 50% of total number of tuples to be sent with RAPIDNET’s naïve decisions. We would also
like to mention that the savings shown for PathVector may not 100% apply to other applications or scenarios. In some cases, the naïve decisions may already be very efficient in network consumption. But our optimal placement will always perform at least as good as naïve placement. It is provably and guaranteed to be optimal in reducing network communication cost.
In this chapter, we describe a novel Min-Cut based algorithm to strategically place MSUs in the DeDoS platform. The optimal placement in the network provably reduces network communication overheads and saves valuable bandwidth to better serve the applications. We implement additional MSUs to accommodate NDlog applications running in DeDoS. Our evaluation shows that the optimization is both efficient and effective. For our sample NDlog application, the optimization can be finished in 1 second for a transit-stub topology with 500 nodes. The optimized placement sends less than half of number of tuples during program execution compared to naïve placement. Given that the optimization runs sub-second and can be done online, we identify that measuring the performance of NDlog applications deployed on DeDoS in an online environment with dynamic re-optimization is a promising future direction. Measuring the sensitivity of the efficiency and effectiveness of MC2 algorithm with other NDlog applications is also an interesting direction to explore.
In this chapter, we first discuss challenges in preserving privacy for distributed debugging with network provenance. Then we propose a novel technique that enables engineers and administrators to diagnose distributed systems with privacy guarantees. We also formally prove our scheme is secure under our privacy model. And finally we describe our prototype implementation and show the applicability and overheads of the prototype in evaluation.

5.1 OVERVIEW AND GOALS

Although the use of provenance has been suggested for a variety of settings, we posit that the lack of confidentiality in existing provenance systems severely limits its practical use since data cannot be shared securely across administrative boundaries.

5.1.1 SECURITY GOALS

Privacy Preserving Network Provenance (PPNP [99]) enables support for privacy-preserving provenance maintenance and querying. Our high-level goal is to enable operators of separately administered systems to share provenance data in a more controlled manner such that sensitive data are revealed only to authorized parties.

We conceptualize privacy-preserving network provenance by considering a coloring of the provenance graph. Let $C$ be a set of colors, which can be roughly viewed as credentials. In our privacy-preserving provenance model, each data owner has one or more colors, and assigns exactly one of its colors to each of its tuples.
Conceptually, the coloring of nodes defines the access control policy for the provenance graph.

Dismissed: simple encryption of tuples. At first blush, it may seem sufficient to simply encrypt the values of tuples stored in the provenance graph to provide privacy protections. In our initial exploration of privacy-preserving provenance, we considered using attribute-based encryption (ABE) [80] where colors represented attributes and tuples were encrypted to enforce access controls.

However, encrypting only the vertices in the provenance graph is insufficient to provide meaningful security since the structure of a provenance graph can also be revealing. For example, in the case of inter-domain routing, the structure of a provenance subgraph belonging to an autonomous system (AS) may reveal the size of that AS, its connections, and its business relationships (i.e., peers). More generally, merely obfuscating the “labels” in a graph has been dismissed by the security community as a poor method of information hiding [74].

We require a stronger security model in which any party, honest or malicious, learns only the portions of a given provenance graph for which it possesses the corresponding colors. We express our desired security properties more concretely next, and present and analyze a scheme based on Searchable Symmetric Encryption [25] (SSE) and more generally Structured Encryption (SE) [18] in Section 5.2. We view part of our contribution as identifying a novel application of SSE and SE to network provenance, which is also much lighter-weight compared to ABE.\footnote{One important difference between SE and ABE is that SE does not provide “collusion-resistance,” meaning different users might be able to combine their keys to decrypt data neither could decrypt on their own. However, for us credentials will be atomic, i.e., not Boolean formulae on attributes, so this does not cause an issue.}
Security goals and threat model. Let $G = (V, E)$ be a simple directed graph\(^2\) and $L : V \rightarrow 2^C$ be a labeling function of vertices (tuples) to colors. In a privacy-preserving network provenance system, in response to a query, a (honest or malicious) querier $P$ with colors $C_P \subseteq C$ learns only a subgraph $G' \subseteq G$ such that $G' = (V', E')$ where $V' = \{v \in V | L(v) \in C_P\}$ and $E' = \{(v_i, v_j) | v_i, v_j \in V \text{ and } \{v_i, v_j\} \cap V' \neq \emptyset\}$.

We note that the subgraph may also contain half-edges in which only one of its adjacent vertices is in the subgraph; for the “invisible” node in half-edges, $P$ learns only that there is a corresponding node in the full provenance graph.

We will use standard cryptographic primitives such as blockciphers and hash functions, and make standard assumptions about their security.

In Section 5.2, we formally define and analyze security properties of PPNP.

Additional goals. A privacy-preserving network provenance scheme should provide the following properties:

- **Completeness**: Upon receiving a query, PPNP should return all vertices and edges that the requester is allowed to view.

- **Portability**: Provenance queries can be answered efficiently, and the results should remain secure and complete even when the provenance information is communicated to a remote third-party node (e.g., for caching purposes or to delegate storage to a cloud-based infrastructure) that may not have full access to plaintext provenance.

- **Performance**: The protocol should have reasonably low overhead, especially during provenance maintenance time. This is particularly important for long running distributed systems that have complex logic.

\(^2\)A simple directed graph is a directed graph with no self-loops or multiple edges.
Importantly, PPNP does not identify misbehaving nodes. While there do exist techniques for discovering nodes in provenance systems that misbehave or equivocate when answering provenance queries [103], such secure network provenance systems depend on having unrestricted access to provenance information. We suspect—although we do not investigate it here—that secure provenance systems can be straightforwardly deployed alongside PPNP within administrative domains.

We leave the study of the integration of secure network provenance with privacy-preserving network provenance as an interesting future research direction.

5.2 STRUCTURED ENCRYPTION FOR COLORED SUBGRAPHS

PPNP relies on a new structured encryption scheme [18] for “colored subgraph queries.” Intuitively, such a scheme allows a user to encrypt a colored graph such that the encryptor can issue a per-color token that allows decrypting only the subgraph whose vertices have that color. The scheme leverages searchable symmetric encryption (SSE) [25] which allows data to still be efficiently searchable by a third-party even when it remains encrypted (Section 5.2.3).

Briefly, PPNP operates by constructing a dictionary in which the colors are the keys and each corresponding value contains the vertices with that color and the edges between them. It also contains a secret share of the corresponding cross-color edges, where the other secret share is in the value for the other color (Section 5.2.4). From the retrieved query result, an unauthorized user might be able to infer an approximation of graph size, but she will not know the exact size (due to padding) or details of individual edges as they are protected by encryption and/or secret sharing. We further describe in Section 5.2.5 fragmentation, a performance optimization, and present the complete leakage profile.
5.2.1 Notation and Conventions

All algorithms, including adversaries, are required to be efficient and may be randomized unless otherwise specified. If $A$ is a randomized algorithm, then $x \leftarrow s A(\ldots)$ denotes running $A$ on the elided inputs and fresh random coins, and assigning the output to $x$. If $A$ is deterministic, we drop the dollar-sign next to the arrow. We denote by $\Pr [ A(\ldots) \Rightarrow x ]$ the probability that $A$ outputs $x$ when run on the elided inputs and fresh random coins. If $S$ is a finite set, then $s \leftarrow s S$ denotes sampling a uniformly random element from $S$ and assigning it to $s$. We abbreviate $s_1 \leftarrow s S ; s_2 \leftarrow s S$ by $s_1, s_2 \leftarrow s S$. Strings are binary. If $s$ is a string, then $s_i$ denotes its $i$-th bit. If $s_1, s_2$ are strings then $s_1 \parallel s_2$ denotes an encoding of $s_1, s_2$ from which they are uniquely recoverable. We often use other objects such as graphs as arguments to algorithms or with string notation, with the understanding that they are first implicitly encoded as strings in some canonical way.

5.2.2 Formal Model and Security

Formal Model. Let $G = (V, E)$ be a simple directed provenance graph, $C$ be a set of colors, and $L : V \rightarrow 2^C$ be a labeling function. A structured encryption scheme for privacy preserving network provenance is a tuple of four algorithms:

$$GE = (\text{KeyGen}_G, \text{Enc}_G, \text{TokenGen}_G, \text{Search}_G)$$

The key generation algorithm $\text{KeyGen}_G$ outputs a key $K$. The encryption algorithm $\text{Enc}_G$ takes as inputs a key $K$, a provenance graph $G$, and a labeling function $L$, and outputs an encrypted graph $\gamma$. The deterministic token generation algorithm $\text{TokenGen}_G$ takes as inputs a key $K$, and a color $c$, and outputs a token $\tau_c$. The deterministic query-answering algorithm $\text{Search}_G$ takes as input a list of tokens $(\tau_{c_1}, \ldots, \tau_{c_q})$ and an encrypted graph $\gamma$, and outputs a graph $G'$. 

58
Correctness. We say that GE is correct if for all $G, C, L$ as above, all subsets $C' \subseteq C$, all $K$ output by $\text{KEYGEN}_G$, all $\gamma$ output by $\text{ENC}_G(K, G, L)$, all $c' \in C'$ and all $\tau_{c'}$ output by $\text{TOKENGEN}_G(K, c')$, it holds that $\text{SEARCH}_G((\tau_{c'}), c' \in C', \gamma)$ outputs the subgraph $G' = (V', E')$ of $G$ such that $V' = \{v \in V | L(v) \in C'\}$ and $E' = \{(v_1, v_2) \in E | v_1, v_2 \in V'\}$.

**Remark 5.2.1** The structured encryption schemes we define are “response-revealing” in the terminology of [18]. Roughly, this means that there is not a separate decryption algorithm, and the search algorithm reveals the result of the query in the clear. This is because in our application the querier and the entity executing the $\text{SEARCH}_G$ are actually the same. This will also be the case for the searchable symmetric encryption schemes we define later.

Security. We consider a non-adaptive, simulation-based notion. Here “non-adaptive” refers to the fact that we only consider an adversary that chooses its queries before seeing the encrypted graph, and “simulation-based” captures the intuition that the adversary only learns some associated leakage of the graph. Namely, given a leakage function $L_K$, consider the following experiments with an adversary $A$ against GE and simulator $S$. (Here and in similar experiments below, $A$ and $S$ maintain state that we omit for simplicity.) In the real experiment $\text{REAL}_{GE}(A)$:

- The adversary $A$ first outputs a graph $G$, set of colors $C$, labeling function $L$, and colors $c_1, \ldots, c_q \in C$.
- The experiment next generates $K \leftarrow s \text{KEYGEN}_G$, $\gamma \leftarrow s \text{ENC}_G(K, G, L)$, and $\tau_{c_i} \leftarrow \text{TOKENGEN}(K, c_i)$ for all $i = 1$ to $q$. It then passes $(\gamma, \tau_{c_1}, \ldots, \tau_{c_q})$ to $A$.
- Finally, the adversary $A$ outputs a bit $b$, which is output by the experiment.

In the ideal experiment $\text{IDEAL}_{L,K,GE}(A, S)$:
• The adversary $A$ first outputs a graph $G$, set of colors $C$, labeling function $L$, and colors $c_1, \ldots, c_q \in C$.

• The experiment then generates $\ell \leftarrow \text{LK}(G, C, L, c_1, \ldots, c_q)$ and passes $\ell$ to the simulator $S$.

• Finally, the simulator $S$ outputs a bit $b$, which is output by the experiment.

We say that GE is **Lk-Secure** if for every adversary $A$ there is a simulator $S$ such that

$$\Pr[\text{REAL}_{GE}(A) \Rightarrow 1] - \Pr[\text{IDEAL}_{Lk,GE}(A, S) \Rightarrow 1]$$

is sufficiently small (say $2^{-128}$).

### 5.2.3 Searchable Symmetric Encryption

We provide some background on searchable symmetric encryption (SSE), first introduced by Curtmola et al. [25], which we will use in our main construction below.

Let $W$ be a set of keywords (also called labels). A dictionary is a list of keyword-data pairs $I = ((w_1, d_1), \ldots, (w_n, d_n))$, where $w_1, \ldots, w_n \in W$ are unique. We write $w \in I$ to mean that $w = w_i$ for some $1 \leq i \leq n$. For $w \in I$ we denote by $I(w)$ the corresponding data $d_i$.

**Syntax.** An SSE scheme is a tuple of four algorithms

$$\text{SSE} = (\text{KEYGEN}_S, \text{ENC}_S, \text{TRAPGEN}_S, \text{SEARCH}_S)$$

defined as follows. The key-generation algorithm $\text{KEYGEN}_S$ outputs a key $K$. The encryption algorithm $\text{ENC}_S$ takes as inputs a key $K$ and a dictionary $I$ and outputs an encrypted dictionary $\tilde{I}$. The deterministic trapdoor generation algorithm $\text{TRAPGEN}_S$ takes as inputs a key $K$ and a keyword $w$, and outputs a trapdoor $T_w$. The deterministic searching algorithm
SEARCH$_S$ takes as inputs a trapdoor $T_w$ and an encrypted dictionary $\tilde{I}$, and outputs a string $d$.

Correctness. We say that SSE is **correct** if for all $n \in \mathbb{N}$, all dictionaries $I = ((w_1, d_1), \ldots, (w_n, d_n))$, all keywords $w \in W$, all $K$ output by KEY$_{GEN}_S$, all $\tilde{I}$ output by ENC$_S(K, I)$, and all $T_w$ output by TRAP$_{GEN}_S(K, w)$, it holds that SEARCH$_S(T_w, \tilde{I})$ outputs $I(w)$ if $w \in I$ and $\perp$ otherwise.

Security. As above, we consider a non-adaptive, simulation-based notion. Given a leakage function $L_K$, consider the following experiments with an adversary $A$ against SSE and simulator $S$. In the *real* experiment REAL$_{SSE}(A)$:

- The adversary $A$ first outputs a dictionary $I$ as above and keywords $w_1, \ldots, w_q \in W$.

- The experiment next generates $K \leftarrow$ KEY$_{GEN}_S$, $\tilde{I} \leftarrow$ ENC$_S(K, I)$, and $T_{w_i} \leftarrow$ TRAP$_{GEN}_S(K, w_i)$ for all $i = 1$ to $q$. It then passes $(\tilde{I}, T_{w_1}, \ldots, T_{w_q})$ to $A$.

- Finally, the adversary $A$ outputs a bit $b$, which is output by the experiment.

In the *ideal* experiment IDEAL$_{L_K, SSE}(A, S)$:

- The adversary $A$ first outputs a database $I$ as above and keywords $w_1, \ldots, w_q \in W$.

- The experiment then generates $\ell \leftarrow L_K(I, w_1, \ldots, w_q)$ and passes $\ell$ to the simulator $S$.

- Finally, the simulator $S$ outputs a bit $b$, which is output by the experiment.

We say that SSE is **Lk-Secure** if for every adversary $A$ there is a simulator $S$ such that

$$\Pr[\text{REAL$_{SSE}(A) \Rightarrow 1$}] - \Pr[\text{IDEAL$_{L_K, SSE}(A, S) \Rightarrow 1$}]$$

is sufficiently small.
5.2.4 MAIN CONSTRUCTION

We give a construction of structured encryption scheme for colored subgraphs from an SSE scheme. The techniques are inspired by the structured encryption scheme for labeled graphs by Chase and Kamara [18], but the details of our construction differ.

Scheme. Let $\text{SSE} = (\text{KEYGEN}, \text{ENC}, \text{TRAPGEN}, \text{SEARCH})$ be an SSE scheme. We assume graphs have vertex labels of some fixed length. Our structured encryption scheme for colored subgraphs with nonce-length $k \in \mathbb{N}$ is defined as

$$\text{GE}_1 = (\text{KEYGEN}_1, \text{ENC}_1, \text{TOKENGEN}_1, \text{ANSWER}_1)$$

in Figure 5.1. Correctness follows from taking the nonce-length to be sufficiently large to avoid collisions.

Intuitively, the scheme simply uses the underlying SSE scheme to store the data for each color. The data for each color contains the vertices with that color and the edges between them. It also contains a secret share of the corresponding cross-color edges, where the other secret share is in the data for the other color. We explain our algorithm in more details in Section 5.3, with a driving example that contains a concrete provenance graph.

Remark 5.2.2 In the implementation described in Section 5.4, we do not store all the graph data in the dictionary as defined in the above scheme, but rather the data stored in the dictionary serves as pointers to encrypted vertices or edges, which can then be decrypted by the entity processing the query. This is done only for performance and does not affect security.

Security Analysis. For the security analysis, we define leakage profile $\mathcal{L}_k$ in terms of a given leakage profile $\mathcal{L}_k$ for the underlying SSE scheme, plus some additional leakage on
cross-color edges. Namely, we define $L_k_1(G, C, L, c_1, \ldots, c_q)$ as $L_k(I, c_1, \ldots, c_q)$, for $I$ defined as in Figure 5.1, plus $\mathcal{H}$ and $N_H$, where

$$
\mathcal{H} = \{(u, v) \mid L(u) = c_i \neq c_j = L(v), c_i, c_j \in (c_1, \ldots, c_q)\}
$$

and

$$
N_H = \{|(x, y)| \mid |(L(x), L(y)) \cap (c_1, \ldots, c_q)| = 1\}.
$$

Note that $N_H$ is the number of cross-color edges from(to) vertices within $(c_1, \ldots, c_q)$ to(from) vertices outside $(c_1, \ldots, c_1)$; the actual edges remain secret.

For the SSE schemes we consider, the data corresponding to the keywords for which trapdoors are requested is output by the leakage profile (as well as some additional leakage discussed later). This means $GE_1$ leaks not only the subgraph corresponding to the colors for which tokens are requested, but also those cross-color edges whose two colors are in $(c_1, \ldots, c_q)$, and the total number of cross-color edges to or from this subgraph to vertices with other colors that are not in $(c_1, \ldots, c_q)$. Leaking the total number of cross-color edges could be eliminated by using padding and a message authentication code, but we omit this from our main construction for bandwidth efficiency.

Formally, we prove the following theorem:

**Theorem 5.2.3** If SSE is Lk-secure, then GE is Lk-secure.

Proof Sketch. Given an adversary $A_1$ against GE, there is a corresponding adversary $A$ against SSE defined in the natural way. Hence, by Lk-security of SSE, there is a simulator $S$ such that

$$
\Pr[\text{REAL}_{SSE}(A) \Rightarrow 1] - \Pr[\text{REAL}_{Lk, SSE}(A, S) \Rightarrow 1]
$$

is sufficiently small. We use $S$ to construct a simulator $S_1$ for $A_1$ as follows. Given $Lk_1(G, C, L, (c_1, \ldots, c_q)) = (Lk(I, (c_1, \ldots, c_q)), \mathcal{H}, N_H)$ as defined above, $S_1$ runs $S$ on
input $L(K(I, (c_1, \ldots, c_q)))$. Let $\ell$ be the output of $S$. Then $S_1$ runs $A_1$ on $\ell$ and the rest of its input prepared appropriately using $H$ and $N_H$, and returns its output. By the construction of $A$ it follows that

$$\Pr[\text{REAL}_{LK, GE_1}(A_1) \Rightarrow 1] - \Pr[\text{IDEAL}_{LK, GE_1}(A_1, S_1) \Rightarrow 1]$$

is also sufficiently small. Hence $GE_1$ is $LK_1$-secure.

1: function $\text{KEYGEN}_1$
2: \hspace{1em} return $K \leftarrow s$ $\text{KEYGEN}_S$
3: function $\text{ENC}_1(K, G, L)$
4: \hspace{1em} **Construct Dictionary:**
5: \hspace{2em} For each $(u, v) \in E$ do: $p_{uv}, q_{uv} \leftarrow s \{0, 1\}^{|u|}$, $n_{uv} \leftarrow s \{0, 1\}^k$
6: \hspace{2em} For each $c \in C$ do:
7: \hspace{3em} $l_cV \leftarrow \{v \mid L(v) = c, v \in V\}$
8: \hspace{3em} $l_cE \leftarrow \{(u, v) \mid L(u) = L(v) = c, (u, v) \in E\}$
9: \hspace{3em} $l_cH_1 \leftarrow \{(u \oplus p_{uv}, q_{uv}, n_{uv}) \mid L(v) \neq L(u) = c, (u, v) \in E\}$
10: \hspace{3em} $l_cH_2 \leftarrow \{(p_{xy} \oplus q_{xy}, n_{xy}) \mid L(x) \neq L(y) = c, (x, y) \in E\}$
11: \hspace{3em} $l_c \leftarrow (l_cV || l_cE, || l_cH_1 || l_cH_2)$
12: \hspace{1em} $I \leftarrow (c, l_c)_{c \in C}$
13: \hspace{1em} **Encrypt Dictionary:**
14: \hspace{2em} return $\gamma \leftarrow \text{ENC}_S(K, I)$
15: function $\text{TOKENGEN}_1(K, c)$
16: \hspace{1em} return $\tau \leftarrow \text{TRAPGEN}_S(K, c)$
17: function $\text{SEARCH}_1((\tau_{c_1}, \ldots, \tau_{c_q}), \gamma)$
18: \hspace{1em} **Search Encrypted Dictionary:**
19: \hspace{2em} For each $c_i, 1 \leq i \leq q$, do: $l_{c_i} \leftarrow \text{SEARCH}_S(\gamma, \tau_{c_i})$
20: \hspace{1em} **Reconstruct Graph:**
21: \hspace{2em} For each $c_i, 1 \leq i \leq q$:
22: \hspace{3em} Parse $l_{c_i}$ as $(l_{c_i}V || l_{c_i}E || l_{c_i}H_1 || l_{c_i}H_2)$
23: \hspace{3em} For each $v \in l_{c_i}V$, $V' \leftarrow V' \cup v$
24: \hspace{3em} For each $(u, v) \in l_{c_i}E$, $E' \leftarrow E' \cup (u, v)$
25: \hspace{3em} For each $n$ in $(u' \oplus p, q, n)$ and $(p, v' \oplus q, n)$:
26: \hspace{4em} Compute $(u', v')$ and $E' \leftarrow E' \cup (u', v')$
27: \hspace{1em} return $G' = (V', E')$

Figure 5.1: Structured encryption scheme for colored subgraphs $GE_1$. 64
5.2.5 SSE with Fragmentation

Finally, we describe a dictionary-based SSE scheme with data fragmentation based on the \( \pi_{\text{pack}} \) scheme from Cash et al. [16], which we optimize for bandwidth efficiency. We use this SSE scheme in our structured encryption scheme for colored subgraph queries above. Intuitively, the SSE scheme “chops up” the data for each keyword into equal-sized fragments of some length parameter, after padding to a multiple of the fragment length. It is described as “pack”ing multiple items into one fixed length chunk in the original \( \pi_{\text{pack}} \) scheme.

More formally, to define the SSE scheme, let \( \text{PRF} : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^* \) be a function family. We also use a dictionary data structure, which can be formalized in standard ways. The scheme \( \text{SSE}_F \) with fragmentation-length \( f \in \mathbb{N} \) and label-length \( k \in \mathbb{N} \) is defined as

\[
\text{SSE}_F = (\text{KEYGEN}_F, \text{ENC}_F, \text{TRAPGEN}_F, \text{SEARCH}_F)
\]

in Figure 5.2. Correctness follows from taking the label-length to be sufficiently large to avoid collisions.

Security Analysis. For the security analysis, we adopt the query pattern leakage as defined by Chase and Kamara [18]. Namely, define \( \text{QP}(w_1, \ldots, w_q) \) to be the \( q \times q \) binary matrix with a 1 at position \( i, j \) if \( w_i = w_j \) and 0 otherwise. Then we define the leakage function

\[
\text{L}_{K_F}(\mathcal{I}, w_1, \ldots, w_q) = \left( \{\mathcal{I}(w_i)\}_{i \in [q]}, \sum_{i=1}^{n} \left[ |\mathcal{I}(w_i)|/f \right], \text{QP}(w_1, \ldots, w_q) \right).
\]

The corresponding security theorem follows from the analysis of the \( \pi_{\text{pack}} \) scheme in Cash et al. [16].
Intuitively, it hides the length of each data item (while avoiding padding each data item to the maximum length) for which the adversary does not have the token for the corresponding keyword. Interestingly, the fact that these lengths are hidden does not seem to depend on $f$. However, as $f$ increases, the adversary gets a better upper-bound on the number of keywords, since $\sum_{i=1}^{n} \lceil |I(w_i)|/f \rceil$ is closer to $n$. In terms of bandwidth efficiency, as $f$ decreases, more pseudorandom labels need to be stored in the dictionary.

Hence, there is a tradeoff: bandwidth depends on how close the length of each data item is to $f$, due to padding. If $f$ is large and these lengths are not close to $f$, then poor bandwidth efficiency is achieved. The optimal choice of $f$ for bandwidth efficiency is therefore data-dependent.

Let $\{c_1, \ldots, c_n\}$ be the color set, $k$ be the length of pseudorandom labels and $I = ((c_1, d_1), \ldots, (c_n, d_n))$ be the constructed dictionary. The optimal fragmentation size $F$ can be derived by

$$\min_F \sum_{i=1}^{n} \left( \left\lceil \frac{|d_i|}{F} \right\rceil \times (F + k) \right).$$

5.3 Privacy Preserving Network Provenance Design

In this section, we introduce PPNP, a novel privacy-preserving network provenance system. PPNP uses the structured encryption scheme presented in Section 5.2 as a cryptographic primitive to enforce access control policies.

The access control policies are specified as the colors of the vertices and edges in the provenance graph, as determined by the provenance data owners. For simplicity, we assume that the color of an edge is determined by the colors of its two end vertices. (The proposed technique can be extended to support the case where edges are labeled in isolation with vertex colors.)
Figure 5.2: Searchable Symmetric Encryption scheme with fragmentation.

Figure 5.4 shows an overview of PPNP’s design through a simplified example. Here, a node Q intends to query a provenance entry from node X, who already has a cached encrypted provenance graph from node Y, where the complete provenance graph is as is depicted in Figure 5.3.

- Upon receiving a provenance query for $V_i$ from Q, X sends its own encrypted provenance graph to Q, along with the cached encrypted provenance graph from Y.
Table 5.1: Provenance dictionaries before fragmentation.

<table>
<thead>
<tr>
<th>term(c)</th>
<th>vertices(lcY)</th>
<th>edges(lcE)</th>
<th>half-edges(lcH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>@X:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>red</td>
<td>V1, V4</td>
<td>∅</td>
<td>(p21, V1 ⊕ q21, n21), (V4 ⊕ p42, q42, n42)</td>
</tr>
<tr>
<td>green</td>
<td>V2, V5</td>
<td>V5V2</td>
<td>(V2 ⊕ p21, q21, n21), (p42, V2 ⊕ q42, n42), (p32, V2 ⊕ q32, n32)</td>
</tr>
<tr>
<td>@Y:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blue</td>
<td>V3</td>
<td>∅</td>
<td>(p63, V3 ⊕ q63, n63), (V3 ⊕ p32, q32, n32)</td>
</tr>
<tr>
<td>green</td>
<td>V6</td>
<td>∅</td>
<td>(V6 ⊕ p63, q63, n63)</td>
</tr>
</tbody>
</table>

Figure 5.3: An example provenance graph.

- Having received both X’s and Y’s encrypted provenance graph, Q sends out *token requests* to X and Y, asking for tokens to access their respective authorized colors.
- X and Y send back to Q the tokens—that is, the corresponding keys for the colors that Q are authorized to access: green from X, and blue and green from Y.
- Once Q receives the tokens, it can then extract a sub-provenance graph for V₁ that Q is allowed to view.

To illustrate differential access control in PPNP, we show the view from another node P which intends to query the same graph as Q in Figure 5.5. P is authorized to access red and green from X, and blue from Y, while Q is authorized to access green from X, and blue and green from Y.
We also show components, i.e. nodes and (half)edges, of Q’s view in Figure 5.6. Q decrypts ciphertext-blocks from X and Y separately. From X, Q successfully decrypts two vertices (V2 and V5), one full edge ((5,2)), and three halfedges ((2,1), (4,2), (3,2)). Similarly, from Y, Q gets two vertices (V3 and V6), and three halfedges ((3,2), (6,3), (6,3)). For halfedges, we use “cut shapes” to denote matching halfedges visually. The actual matching is done by the nonce associated with each halfedge. Note that for full edge, it is not “cut”. Also, in order to recover the full edge, a node must possess both halfedges. Thus Q would match the two wavy halfedges to recover edge (6,3), and two sawtooth halfedges to recover edge (3,2). Halfedges (2,1) and (4,2) is not matched, and Q can only infer that these two halfedges have one green vertices.

5.3.1 Answering Provenance Queries

Upon receiving a provenance query, a node first prepares its provenance graph as a dictionary, and then replies with the encrypted graph as obtained using the $\text{ENC}_1$ function.

Assume at node X, the provenance graph is denoted by $G = (E, V)$, where $V$ is set of vertices and $E$ is set of edges in $G$. Additionally, for each $v \in V$, we associate a color $c$ with $v$. To construct the data structure for SSE, we create the dictionary as follows: (For simplicity, and in order to achieve a solution using only light-weight symmetric-key cryptographic primitives, we impose a constraint that a vertex is associated with exactly one color. Extending our scheme to allow vertices with multiple colors, or more generally Boolean formulae on the colors, is an interesting direction of future work.)

The $\text{ENC}_1$ function (shown in Figure 5.1) produces an encrypted dictionary. It begins by constructing a list $l_{cV}$ that contains all vertices that are labeled with the color $c$, for each color $c$ in the color set $C$. It then creates a list $l_{cE}$ that contains all edges whose ends are labeled as $c$. Then, a list $l_{cH}$ is constructed such that all half-edges of $c$ are covered, where the half-edges of a color $c$ are defined as the edges $e = (u, v)$ that have either $u$ or $v$ labeled...
Figure 5.4: Answering provenance query. (a) At earlier time, X caches Y’s encrypted provenance graph. (b) Upon receiving query from Q, X sends Q both its own encrypted and cached provenance graphs. (c) Q asks X and Y for authorized tokens. (d) X and Y send back tokens for colors that Q are authorized to access. (e) Q decrypts provenance (sub)graphs that it is allowed to view. Note that the dashed edge was recovered by cross-node handling mechanism.

as $c$, but not both. Finally, ENC1 concatenates the lists $l_{cV}$, $l_{cE}$, and $l_{cH}$ into a single list, $l_c$.

The function constructs $l_c$ for each $c \in C$.

For a half-edge $e = (u, v)$ where $u$ is labeled as $c_u$ and $v$ is labeled as $c_v$ ($c_u \neq c_v$), we randomly generate $p$, $q$, and $n$ such that $|u| = |v| = |p| = |q| = |n|$. In $l_{c_uH}$, we include a tuple $(u \oplus p, q, n)$ as an element in $l_{c_uH}$, and its corresponding tuple $(p, v \oplus q, n)$ as an element in $l_{c_vH}$.
After building the dictionary, each \((c, l_c)\) pair is encrypted by a secret key \(K^X_c\), where \(c\) is the color and \(X\) is the node where \(G\) is located. \(\text{ENC}_1\) returns \(\text{ENC}_S(K, \mathcal{I})\), the encrypted dictionary.

As an example, Table 5.1 provides the constructed dictionary for the provenance graph depicted in Figure 5.3. As one can see from the Table, the values of different keys in the dictionary may have different lengths. Unless otherwise masked (see below), this may leak additional information about the underlying provenance graph. For instance, when the vertices are labeled in an unbalanced fashion—e.g., a majority of vertices share one same label and a small minority of vertices share another label—an adversary would be able to distinguish the two labels from the lengths of document lists.
A typical approach to eliminating this information leakage is to perform padding to the dictionary such that all keys have same-lengthed values. However, such an approach introduces significant storage and communication overhead. As described in Section 5.2.5, we instead adopt an SSE scheme that incorporates data fragmentation to mitigate the information leakage without incurring these large costs. Table 5.2 shows the final fragmented dictionary, with fragment size \( f = 5 \times |c| \) and \(|pad| = |c|\).

Table 5.2: Fragmentation of the inverted-indices (dictionaries) from Table 5.1. \( F \) is a PRF. Colors are abbreviated as \( r, g, b \). To improve readability, parentheses and commas have been omitted.

<table>
<thead>
<tr>
<th>term((e))</th>
<th>document-list((l_e))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( @X: )</td>
<td></td>
</tr>
<tr>
<td>( F(K^X_r, 1) )</td>
<td>( V_1 ) ( V_4 ) ( p_{21} ) ( V_1 \oplus q_{21} ) ( n_{21} )</td>
</tr>
<tr>
<td>( F(K^X_r, 2) )</td>
<td>( V_4 \oplus p_{42} ) ( q_{42} ) ( n_{42} ) ( pad ) ( pad )</td>
</tr>
<tr>
<td>( F(K^Y_g, 1) )</td>
<td>( V_2 ) ( V_5 ) ( V_5 ) ( V_2 ) ( V_2 \oplus p_{21} )</td>
</tr>
<tr>
<td>( F(K^Y_g, 2) )</td>
<td>( q_{21} ) ( n_{21} ) ( p_{42} ) ( V_2 \oplus q_{42} ) ( n_{42} )</td>
</tr>
<tr>
<td>( F(K^Y_g, 3) )</td>
<td>( p_{32} ) ( V_2 \oplus q_{32} ) ( n_{32} ) ( pad ) ( pad )</td>
</tr>
<tr>
<td>( @Y: )</td>
<td></td>
</tr>
<tr>
<td>( F(K^Y_b, 1) )</td>
<td>( V_3 ) ( p_{63} ) ( V_3 \oplus q_{63} ) ( n_{63} ) ( V_3 \oplus p_{32} )</td>
</tr>
<tr>
<td>( F(K^Y_b, 2) )</td>
<td>( q_{32} ) ( n_{32} ) ( pad ) ( pad ) ( pad )</td>
</tr>
<tr>
<td>( F(K^Y_b, 1) )</td>
<td>( V_6 ) ( V_6 \oplus p_{63} ) ( q_{63} ) ( n_{63} ) ( pad )</td>
</tr>
</tbody>
</table>

### 5.3.2 Handling Cross-node Communications

Consider an edge \( e = (u, v) \) that crosses two physical nodes: \( u \) is located on node \( Y \) and has label \( c_u \); \( v \) is located on node \( X \) and has label \( c_v \). It should be noted that a querier who has access to vertex \( v \) should not automatically be granted access to any information of \( u \) (i.e., the ID and label of \( u \)). Unless explicitly granted the access to \( u \), the querier should only be aware that \( v \) is connected to some unknown vertex located on node \( Y \).

Since each node encrypts its own provenance graph by its own secret key, it is non-trivial for two nodes to share information of such a cross-node edge without compromising
the aforementioned property. To achieve this, we resort to an approach similar to how we treated half-edges in Section 5.3.1.

When Y prepares its provenance graph, it randomly generates $p$, $q$, and $n$ such that $|u| = |v| = |p| = |q| = |n|$. Similar to half-edges, it includes tuple $(u \oplus p, q, n)$ in $l_{vH}$, and adds an auxiliary element, $(p, q, n)$, to the final inverted-index. Note that the auxiliary elements of an inverted-index are not encrypted.

When X receives the encrypted inverted-index which includes the auxiliary element) from Y, it parses $p, q, n$ and adds tuple $(p, v \oplus q, n)$ in $l_{vH}$ of its inverted-index. Table 5.1 provides an example of the generated inverted-indices for Figure 5.3, where cross-node communications are involved.

5.3.3 RECONSTRUCT PROVENANCE GRAPH

After the query node receives the final answer to its query and has been granted all tokens, it starts solving the provenance graph jigsaw puzzle by assembling the separated pieces together.

For each puzzle piece, it decrypts the dictionary with its corresponding tokens row-by-row. In more detail, from document list $l_{cV}$ and $l_{cE}$ for label $c$, the vertices and edges with label $c$ can be recovered, as detailed in the SEARCH function listed in Figure 5.1. For half edges in $l_{cH}$, a hash map is maintained, where $n$ in tuple $(p, q, n)$ is the search key. The purpose of this hash map is to match every half edge with its other half using the cuts of the jigsaw.

Recall that each half edge $e = (u, v)$ with $c_u \neq c_v$ appears twice in the dictionary: once in the document list for $c_u$ in the form of $(u \oplus p, q, n)$, and once for $c_v$ in the form of $(p, v \oplus q, n)$. With the two tuples matched by $n$, the original edge $(u, v)$ can be recovered by $(u \oplus p, q) \oplus (p, v \oplus q)$. 
The same matching method applies to cross-node edges as well. Each cross-node edge \( e = (u, v) \) with \( u \in Y \) and \( v \in X \) also appears twice: once in the dictionary reported by \( Y \) in the form of \((u \oplus p, q, n)\), and once in the dictionary reported by \( X \) in the form of \((p, v \oplus q, n)\). From here, the provenance graph can be recovered.

Hidden Information For a query node, after it reconstructs the provenance graph, it is able to see a portion of the subgraph that it is authorized to access. It may also be interested to learn how much information is hidden from it, i.e. what percentage of the original graph it is allowed to access. Our PPNP scheme could easily provide such information without additional cost. The received ciphertext size is a fair estimate of the size of the original provenance graph. Since the ciphertext is in an inverted-index format, the number of rows corresponds to number of colors and each row size corresponds to number of vertices of that color. Thus a querier could trivially compare its reconstructed subgraph with the received ciphertext to get a percentage of how much of the original provenance graph it is authorized to access.

5.3.4 Application of PPNP

While we focus on the application of PPNP to network provenance; it is not difficult to apply PPNP to other graph-based provenance models.

Applying PPNP to PROV. For concreteness, we briefly describe the application of PPNP to the PROV provenance model. PROV adopts a graph-based provenance model where a vertex in the graph can be one of three types: (a) entities, such as physical or digital items; (b) activities, such as actions or processes that derives entities; or (c) agents, such as persons, organizations or software that are partly responsible for activities. Edges in the graph identify direct relationships between the vertices. For example, an edge between
an entity and an activity can describe that the entity was generated by the activity. The exact relationship that an edge identifies is encoded as an annotation of the edge (e.g., a “wasGeneratedBy” annotation for the previous example).

Assume that a set of provenance stores collectively maintain a distributed PROV provenance graph, and that the PROV provenance graph has been “colored” based on an imposed access control policy by the owners. The PROV provenance graph can be constructed as a colored graph with annotations on the edges and vertices. The SSE-based structured encryption scheme presented in Section 5.2 would still be applicable with a minor extension to further incorporate the annotations: we additionally include annotations to their corresponding vertices, edges and half-edges in the constructed dictionary. The retrieved partial view of the provenance will then naturally contain the annotations of the vertices and edges that the querier is permitted to access.

Customizing PPNP for deployment. We also note that there are several tunable knobs that can be customized based on the privacy requirement of individual applications.

For distributed system debugging [20, 94] and IoT tracking [84] that require strong confidentiality, it is necessary to apply GE₁ from Figure 5.1 with the fragmentation-aware SSEₖ scheme in Figure 5.2. For confidentiality, any querier who does not hold the authorized token for a color c would not be able to access the vertices and edges with that color, even if it holds the ciphertext for the entire graph. With fragmentation, the querier cannot deduce from the ciphertext sensitive information such as the names and the number of colors/labels, or the provenance graph size (due to padding).

In provenance-aware healthcare multi-agent systems (HC-MAS) [55], one can simply apply GE₁ with any underlying SSE scheme, as it is unnecessary to apply fragmentation for hiding the names and the number of colors/labels in the provenance graph. In this case, the colors correspond to medical procedures which are publicly available information.
5.4 Evaluation

We constructed PrivProv in RAPIDNET, a declarative NDlog engine built upon the ns-3 network simulator. In order to compare PrivProv with previous network provenance systems (e.g., ExSPAN [102]), we modified it to support PrivProv’s privacy-preserving features. Specifically, we added functionality for constructing inverted-indices from provenance graphs, fragmenting and encrypting the inverted-indices, decrypting ciphertexts, and reconstructing provenance graphs from inverted-indices. In addition, we modified the RAPIDNET compiler to include labels for provenance graphs, and implemented user-defined functions in RAPIDNET to interact with SSE. Cryptographic functions used in our PrivProv implementation were imported from the Crypto++ library [24].

By default, ExSPAN uses “string polynomials” to send digests of provenance graphs. For example, the provenance graph in Figure 5.3 would be transmitted as $V_1(V_2(V_4 + V_5 + V_3(V_5)))$. To ensure a fair comparison to PrivProv, we modified user-defined functions in ExSPAN to include “full” provenance graphs in which a full list of vertices and edges are transmitted to represent a graph.

5.4.1 Evaluation Settings

To understand the relative overheads of building the dictionary, encryption, and performing a token query in PrivProv, we evaluated the following four configurations:

- **ExSPAN** serves as our baseline, which returns results of provenance queries in plaintext. It does not construct or communicate the secure dictionaries used for supporting SSE.

- **ExSPAN+Dictionary** returns provenance queries in the format of a dictionary that maps colors to their corresponding vertices and edges in the provenance graph. For
Table 5.3: Experiment configurations.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Dictionary</th>
<th>Fragment Encrypt</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExSPAN</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ExSPAN+Dictionary</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PrivProv w/o Token</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PrivProv w/ Token</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

an edge with different colored ends, the edge will be included in the entries of both colors.

- **PrivProv w/o token** additionally applies cryptography to construct the privacy preserving query result.

- **PrivProv w/ token** further includes the token query that retrieves, from the original owner, the token required for reconstructing the provenance graph.

Table 5.3 presents a summary of the four configurations.

**Applications.** We evaluated PrivProv with two representative applications: mincost and the Chord [85] distributed hash table (DHT) which provides the capability of inserting and retrieving key value pairs in a distributed fashion. For Chord DHT, we focused on the provenance of the lookup process that, upon a user’s request, identifies the node responsible for hosting a provided key. This process can be succinctly implemented in NDlog using the following rules [62]:

11 lookupResults(@R,K,S,SI,E):- lookup (@NI,K,R,E),
    node (@NI,N), bestSucc (@NI,S,SI), K in (N,S).

12 bestLookupDist (@NI,K,R,E,a_MIN<D>):- node (@NI,N),
    lookup (@NI,K,R,E), finger (@NI,I,B,BI),
    D := K-B-1, B in (N,K).
13 forwardLookup(@NI,a_MIN<BI>,K,R,E):- node(@NI,N),
   bestLookupDist(@NI,K,R,E,D),
   finger(@NI,I,B,BI), D == K-B-1, B in (N,K).
   f_typeOf(BI) != null.

![Figure 5.7: Bandwidth utilization of provenance queries.](image)

**Experimental setup.** We performed the experiments on a Dell OptiPlex 7040 desktop running Ubuntu 14.04 LTS kernel version 3.19.0 with an Intel Core i7-6700 processor, 32GB 2133MHz DDR4 RAM, and a M.2 256GB SATA SSD.
Our experiments used ns-3’s simulation functionality, which allows evaluations of PrivProv in a sizable network topology. Network topologies in all experiments are generated by the GT-ITM topology generator [44] using Transit-Stub type networks. On average, each transit node is connected to eight stub domains; each stub domain contains on average three nodes, while each transit domain contains four nodes. Thus, on average, each transit domain contains 50 nodes. We increase the number of transit domains to vary the size of network topologies. The links between transits are configured with 1Gbps of bandwidth and 20ms latency, while links within a transit have 100Mbps bandwidth and 50ms latency.

Unless explicitly specified, our experiments ran on a 100-node topology generated with the settings described above.

**Workflow.** After applications reach a fixed point—that is, all pair-wise shortest paths are computed (for mincost), and the Chord ring and finger tables are stabilized (for Chord). 1000 provenance queries are issued against randomly chosen min-cost paths (for mincost) and lookup results (for Chord).

PrivProv supports a general graph coloring scheme. In this paper, we adopted a simple scheme in which provenance vertices are colored based on the IP address of the vertices’ host machine. This corresponds to a straightforward role-based access policy. For simplicity, we use the SHA-1 digest of IP addresses as vertex colors.

We set the fragment size to 300, except for the evaluation of fragmentation optimization, which uses a varying fragment size.

5.4.2 Evaluation Results

PrivProv does not introduce any modification to provenance maintenance. (We confirm this below.) We thus focus on evaluating the performance of provenance querying.
Figure 5.8: Latency distribution. The cumulative distribution of provenance query latencies (figure a), a breakdown of provenance query latencies (figure b), and the query latency composition of 100 node topology running 1000 uniformly random queries in Chord (figure c).

COMMUNICATION OVERHEAD

Figure 5.7 presents the aggregate bandwidth utilization over time during the processing of 1000 random queries.

We observed that PrivProv incurs roughly a fourfold communication overhead compared with ExSPAN, where 50% of the overhead comes from the construction of the inverted-index. This is expected since almost every edge is a cross-color edge (recall that vertices are colored based on the hosting machine), each edge would appear exactly twice in the inverted-index. For applications that require less confidentiality of cross-color edges—for example, if it is acceptable to reveal cross-color edges even when a user has tokens for only one end of the edges—the communication overhead would be significantly reduced, as shown by ExSPAN+Dict(Half) in the figure, where a cross-color edge is revealed under edge-tail’s token. The average bandwidth for ExSPAN and ExSPAN+Dict(Half)
are 28.91KB/s and 37.85KB/s respectively, compared to 65.27KB/s for ExSPAN+Dict. Other factors that contribute to the overhead include fragmentation, padding that ensures the inverted list of all colors have the same length, and auxiliary information for handling cross-machine edges. The overhead of token queries is negligible.

**QUERY LATENCY**

Figure 5.8 (a) presents the Cumulative Distribution Function (CDF) of the query latencies of 1000 random provenance queries. The figure shows that PrivProv w/o token does not introduce much latency. In fact, it almost overlaps with ExSPAN. Token queries, however, require an additional round trip time, which slows the provenance query by 20%. We remark that tokens can be cached and hence the token query overhead can be amortized over multiple provenance queries.
Figure 5.10: Communication overhead with varying fragmentation sizes.

Figure 5.8 (b) presents the breakdown of the query latencies. “CPU time” counts the time spent in retrieving plaintext provenance from maintained provenance tables; “provenance time” counts the time spent in constructing and assembling query results based on the retrieved plaintext provenance; “network time” counts simulation time spent in transmitting provenance query results; and “token network time” counts simulation time spent in network delay and transmission delay for token queries.

Since the experiments were run in simulation mode, we calculated query latencies in two parts: we tracked CPU time and provenance time by recording the actual wall clock time during the execution; for network time, we considered ns-3’s simulated network transmission time.

We observed that PrivProv does not incur much computation overhead (as indicated by the low provenance time) for preparing and parsing the secure dictionary of provenance graphs. PrivProv introduces a small increase in network time due to increased size of mes-
sages. The significant delay comes from processing the token queries, which introduces one additional round trip time. We note that these findings align with the latency results of Figure 5.8(a).

One counterintuitive point is that CPU time of PrivProv is slightly lower than that of ExSPAN. Upon further investigation, we found that the additional CPU time in ExSPAN is caused by the implementation of RAPIDNET where objects are deep-copied. In contrast, PrivProv uses a flat encoding for the encrypted secure dictionary and thus has a less “structured” encoding of provenance objects as compared to ExSPAN.
Scalability

Figure 5.9 presents the communication costs for 1000 random provenance queries for various sized network topologies. We make two observations: (i) The communication overhead of provenance queries increases sublinearly with network size. This is expected since for \texttt{mincost} the provenance complexity is directly related to the network diameter, the latter of which also increases sublinearly with network size. (ii) Second, the relative overhead of constructing the dictionary (ExSPAN+Dictionary), creating the secure dictionary (PrivProv w/o token), and processing token queries (PrivProv w/ token) remains unchanged, which reconfirms our observation from Section 5.4.2.

Figure 5.9 shows that the overhead plateaus after 200 nodes. This is due to the fact that the network diameter remains the same. In our experiment setting, we increased network size by adding more transit domains in the network, which did not increase network diameter after four transit domains. Thus, the depth of the resulted provenance graphs did not grow further. The scalability relies on the underlying protocol. PrivProv should not affect the scalability of provenance maintenance and querying since it adds a constant factor of overhead, in both computation and communication.

The Effect of Fragmentation

Figure 5.10 presents the effect of various fragmentation size on communication overhead. We performed the experiments on three network topologies with 100, 200, and 300 nodes respectively. As we can observe clearly from the figure that the “V” shaped lines indicate an optimal fragmentation size which minimizes network communication overhead. The intuition is that with small fragmentation size, the overhead of maintaining meta-data such as random labels is significant, analogous to sending small payload TCP segments, while with large fragmentation size, padding becomes a significant contributing factor to
the communication overhead. As discussed in Section 5.2, the optimal fragmentation size
is determined by the length distribution of $d_i$. The evaluation assumes a coloring scheme
where provenance vertices are colored based on IP address of the vertices’ host machine
(Section 5.4.1). As a result, each color contains provenance vertices and edges related to
cost and mincost tuples on one physical node. Although the network size increases, the
distribution of $d_i$ remains roughly unchanged, and so is the optimal fragmentation size.

EVALUATION OF CHORD DHT

Figure 5.11 presents bandwidth consumption over time for processing 1000 random prove-
nance queries when the underlying protocol is the Chord DHT. We observed that PrivProv
incurs 2-3x overhead in bandwidth consumption. For Chord, the majority of the edges
(around 80%) in a provenance graph have the same color for both ends; the relative cost
for constructing the dictionary is thus significantly lower than that of mincost. The relative
cost for constructing the secure dictionary, on the other hand, remains similar to mincost,
as expected.

Figure 5.8 (c) presents the breakdown of query latency for Chord. The figure shows that
CPU time dominates query latency, due to the increase in program complexity of Chord,
compared with mincost. Again, the increases in provenance time and network time are
nearly negligible.

Token network time delay adds one more round trip time. The increase in token network
time delay compared with mincost is due to the increased number of physical nodes and
hops involved in one provenance query. For mincost, the number of physical nodes and
hops involved in one query is capped by the network diameter. For Chord, however, since it
is an overlay application, each hop corresponds to multiple physical hops in the underlying
network.

85
5.5 SUMMARY

In this chapter, we present PPNP, a novel distributed privacy-preserving network provenance scheme that supports the richness of network provenance while providing strong privacy guarantees over confidential data. We formally prove that our proposed cryptographic-based PPNP scheme is secure, and show how PPNP can be applied to existing provenance systems that require heterogeneous privacy preferences. We develop a prototype implementation, PrivProv, and evaluate its performance on two distinctly different provenance applications. Our evaluation results demonstrate that PrivProv incurs negligible increase in latency and a reasonable bandwidth overhead, making it practical for large, distributed deployments.
In this chapter, we first summarize the dissertation, and then we briefly discuss some interesting and promising future directions related to the dissertation.

6.1 SUMMARY

In this dissertation, we first introduce the DeDoS platform that proposes a finer modularity in network application development. DeDoS aims to optimize resource allocation for distributed systems by making local resources globally available, such that other overloaded MSUs could utilize spare resources to improve throughput. We evaluate our prototype implementation and the results demonstrate that DeDoS significantly outperforms monolithic applications and incurs minimal overheads.

Then with the DeDoS platform, we propose a novel Min-Cut based algorithm that strategically places MSUs in the system such that the inter-machine network communication is minimized. The algorithm globally decides where each MSU should be placed and what data should be sent to them. We also implement a compiler that automatically translates NDlog programs into DeDoS applications for scalable deployment.

Finally, we introduce a novel SSE based encryption scheme to enable privacy preserving network debugging with provenance. We propose a security model to and prove that our encryption scheme provides strong guarantees on privacy of provenance. We implement a prototype and empirically show that our proposed scheme only incurs minimal latency.
increase during provenance querying and reasonable network overhead during provenance maintenance.

6.2 Future Directions

In this section, we identify some interesting and promising future directions that are related to this dissertation. More specifically, we first discuss one future direction that relates migration of MSUs with local data. Then we introduce an interesting provenance model that incorporates probability for debugging Machine Learning and Data Mining systems.

Replicate MSUs with Data Migration In Chapter 3 we described how to clone and merge MSUs when the workload of some MSUs fluctuates. There we mainly focused on CPU or memory intensive tasks. One particular question arises: if the cloned MSU has some local data that need to be persistent, for instance intermediate computing results, how should we merge the MSU without losing data. One naïve approach is to discard any local data when the MSU is merged. In this way, any intermediate results are lost, and need to be recomputed, which may affect system performance. Another approach would be transfer any local data to another MSU of the same type. Although there is no need to recompute the intermediate results, the routing table for upstream MSUs needs to be updated timely to reflect this migration. Also the MSU cannot be merged until all its data are migrated into other MSUs. This may lead to increased latency in reacting to fast changing workloads. It is worth investigation of an approach that only migrates most important local data before merging, i.e., intermediate results that consumed many resources, recomputing would incur high overhead. The first step involves an algorithm that keeps track of how much resources have been committed to local data. Provenance is a natural fit for tracking such information. Then some modifications in the global controller are also required. During merging, the
controller and local agent will need to engage to a more complicated protocol to ensure data migration and timely merging.

Probabilistic Provenance Model  We further discuss a novel probabilistic provenance model for debugging machine learning and data mining applications. MLDM (machine learning and data mining) techniques are increasingly used in data-driven decision processes across different disciplines. However, debugging these applications remains a significant challenge, especially when one tries to understand the interactions between different learning algorithms used in ensembles. At its crux, a majority of MLDM algorithms may generate tuples that are not only probabilistic, but whose dependency relationships with inputs may themselves be probabilistic. This is a feature that cannot be easily captured using existing provenance models. It is worth pursuing of a novel provenance model that combines tuple uncertainties with probabilistic dependencies.

We expect the model to assist MLDM developers by providing multiple options of analysis and debugging, not only explaining why and how results are generated. It is also necessary to explain the absence of an event: why a tuple was not returned as expected. These problems are also called why-not queries with increasing attention in the database community. Wu et al. [94] present a concept called negative provenance to handle why-not queries in Software-Defined Networks. We are going to study if this idea can be applied to our model in MLDM field.

Probabilistic Provenance Application  The probabilistic provenance model mentioned before is not restricted to MLDM applications. Exploring a wide range of other use cases is a promising future direction. Use cases in distributed systems are particular interesting, where lossy links can add uncertainty. Also machine learning is widely used in network
intrusion detection as well. Protocols that are probabilistic in nature are also worthy pur-
suing, for example, gossip distributed protocols and congestion control protocols with
randomized backoffs.
BIBLIOGRAPHY


of ACM SIGMOD International Conference on Management of Data (SIGMOD), 2010.


