UNIVERSITY OF CALIFORNIA
SANTA CRUZ

OPTIMIZING MEMORY-EFFICIENT ALGORITHMS FOR LOW-COST AND SCALABLE NETWORK SERVICES

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Minmei Wang

June 2022

The Dissertation of Minmei Wang
is approved:

Prof. Chen Qian, Chair

Prof. Faisal Nawab

Prof. Katia Obraczka

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Optimizing Memory-efficient Algorithms for Low-cost and Scalable Network Services

by

Minmei Wang

Today, with emerging technology such as Wi-Fi, 4G, and 5G, more heterogeneous mobile devices, including low-power IoT devices, are connected to the Internet. Those billions of connected devices generate a large volume of data and require accessing network services with low latency, making us optimize memory-efficient algorithms for low-cost and scalable network services. Ideally, we expect network services are memory-efficiency and computation-efficiency to meet objectives of high throughput and low latency. However, existing methods and systems face the following challenges: 1) high memory and resource requirements, hard to be deployed on devices with limited resources, 2) processing data slowly to meet objectives. To address these challenges, we propose designing and utilizing space-efficient data structures to redesign various network devices. Specifically, we first design a new memory-efficient data structure called Vacuum filters. Then my thesis focuses on two types of widely-used network services.

The first type of service is security and privacy service for end devices. Mainly, we focus on two fundamental problems. 1) speeding up the certificate validation process, which is the basic and important step before a secure channel is built between end devices. 2) privacy protection when end devices communicate with remote cloud servers. We utilize an efficient data structure called Othello to develop the Collaborative Certificate Validation protocol (CCV), which provides a fast certificate validation for IoT devices. In addition, we design the LOIS framework based on Vacuum filters to protect users' privacy when end devices communicate with remote cloud servers.

The second type of service is the design of efficient network functions (NFs). NFs are essential services on the network support layer. They also rely on efficient data structures and algorithms to achieve memory and computation efficiency. We observed that multiple, co-existed NFs on the same device have become common in today's programmable networks. Thus, we design a tool called HyperMerger which can automatically generate resource-efficient consolidated algorithms to support co-existed NFs simultaneously for both hardware and software platforms.

Acknowledgments

Many people offered me valuable help during my whole Ph.D. career. I am very honored to have had this wonderful experience in my life.

First, I would like to give my sincere thanks to Prof. Chen Qian, my advisor who led me to the research area of computer networks, for his guidance and consistent support to do research. He taught me how to find research problems, conduct solid research with impacts, and present my work to audiences with different backgrounds. The systematic training as a Ph.D. student benefits me a lot, enabling me to find a faculty job to continue my journey to explore exciting and meaningful research problems after my graduation.

I would also like to thank members of my reading committee, Prof. Katia Obraczka, and Prof. Faisal Nawab, for their insightful suggestions for my dissertation.

I've been fortunate to work with many intelligent collaborators and benefit a lot from the collaborations for my thesis. I especially thank Xin Li for his effective advice in the design of CCV, Mingxun Zhou for the great help on the design of Vacuum filters, Shouqian Shi for the insightful discussion of the motivations of LOIS, Xiaoxue Zhang for her help on the analysis of IoT traffic for LOIS, Xiaofeng Shi for his excellent help in the design of HyperMerger. I am grateful for the advice of Prof. Song Han to improve LOIS. I also thank Prof. Zaoxing Liu for his insightful discussions and suggestions on the design for HyperMerger. In addition, I benefit a lot when studying in the lab with Ge Wang, Haofan Cai, Huazhe Wang, Junjie Xie, and Minghao Xie. Thanks for walking the journey of life together.

I want to thank my family for their support and love. They give me the power to do the unknown and challenging exploration. I also thank my friends for making my life more colorful.

I also want to express my sincere thanks to my husband, Xiaofeng Shi, for his sharing of life with me. We both pursue our Ph.D. degrees at the University of California, Santa Cruz (UCSC). We are peers in our lives and on the road to our careers. He is patient and full of life wisdom. It's very pleasant to live with him. I am very grateful for his support and companionship.

Last, I had a satisfying doctoral career at UCSC. UCSC is a beautiful and peaceful place for life and education. I enjoy hiking on the trail road of the campus and seeing many animals, such as squirrels, deers, and raccoons. I like to run around the playground, having a fantastic view of the sea. Thanks to UCSC for giving me a perfect journey in my life.

Chapter 1

Introduction

1.1 Heterogeneous Mobile Devices and Their requirements for Network Services

In recent years, heterogeneous mobile devices, including mobile phones and Internet of Things (IoT) devices, have been widely connected to the Internet via Wi-Fi, 4G, and 5G technology. The number of connected network devices is expected to reach 29 billion by 2023, as predicted by Cisco [23]. And IoT devices will occupy half of the globally connected devices. Mobile devices are pervasive - devices are being deployed in nearly every conceivable environment, such as industrial automation, smart devices, vehicular communication, smart cities, and smart homes [31] [152] [112], which has brought great convenience for human lives.

Heterogeneous mobile devices are connected to the Internet, requesting network services across different layers. We divide network services into four layers: perception layer, network layer, network support layer, and application layer. The perception layer consists of various types of end devices, such as different IoT devices and mobile phones. Its basic purpose is to build a bridge to connect the physical world with the Internet. We summarize network services directly accessed by end devices as services on the perception layer, including data interface services to collect data, security, and privacy services when end devices communicate with other devices or remote servers. The network layer is to transmit data collected or generated by end devices from the perception layer via an existing communication network [79]. Examples of network services on the network layer are various network topologies and routing algorithms. The upper layer is the application layer, which provides different types of applications for end devices, including data storage and processing, data analysis, security and privacy services, etc. The network support layer is an additional layer between the network and application layers. The network support layer aims to collect network data, e.g., network measurement data and data derived from network traffic, to take automated actions based on collected data or transmit the collected data to the application layer to improve the performance of different applications. The network support layer consists of two main modules: the network function (NF) and the service data modules. The network function module consists of various types of network functions, such as load balancing [65, 99, 129] and packet measurements [70, 93, 161]. And the service data module is to collect data related to applications, then uses the data to achieve a more reliable and better performance of the upper-layer applications. For example, when end devices connect to the Internet via cellular networks, the service data module can collect connection information to specify the time and base station for each end device and save them as logs. When there is a connection failure, those logs can be used to analyze the reason for the connection failure.

Currently, 5G technology changes our network by utilizing a high-band spectrum for high speed and low latency communications. On the one hand, 5G boasts the capability to connect billions of end devices to the Internet due to its high bandwidth [135]. And the huge volume of connected devices generates a large amount of data. On the other hand, the 5G network deeply integrates Software Defined Networking (S-DN), enabling an flexible and automatic network at the edge [116]. And new technology such as programmable switches based on the P4 language [37] emerged to handle traffic with a high traffic rate. Hence, more services (e.g., various network functions) are being moved to programmable switches to adapt to the changing network environment. After viewing changes brought into the network, we need to rethink designs for network services to meet network requirements.

1.2 Challenges and Requirements of Designing Network Services

Billions of connected devices build a more convenient human life. At the same time, we need to face serious challenges when designing network services with better performance.

Huge amount of devices and data. Facing billions of devices and lots of data is common for the design of network services. Thus, how to design an efficient network service that has a high throughput and can be scaled to many devices is a challenge.

Devices with limited resources. Network services are finally deployed on heterogeneous network devices. Some of them have limited resources. For example, IoT devices have limited computational and storage resources, and a programmable hardware switch has limited SRAM resources. So, designing a resource-efficient network service that can be deployed on specific devices with limited resources is a challenge.

Considering those challenges, designs of network services should meet the following requirements.

- **Low-cost.** Network services with low computational and memory costs are desirable to be deployed on devices with limited resources.
- **Scalability.** Since there are billions of connected devices and the number of connected devices continuously increases, designed network services need to be scaled to handle the huge amount of devices and their generated data.

1.3 Design Principles for Network Services

This thesis introduces my research on designing network services on different layers. There are two fundamental design principles to guide the design of various network services and meet the above requirements.

Utilizing and designing new space-efficient data structures. We found that space-efficient data structures are useful tools for building a network service that meets requirements. Especially, those data structures are widely used to handle a large volume of data. Examples of those data structures are various types of approximate membership query (AMQ) data structures [39, 59, 142], hash tables [148], various sketches [52, 70, 93, 94]. By carefully analyzing components of network services, utilizing those space-efficient data structures is a promising way to implement efficient network services.

Comprehensive analysis of network services. An essential step is to analyze network services and choose the proper device and platform to deploy the services. For example, more network functions are moved to programmable switches with a high packet processing rate due to the scenarios of high traffic data and low latency requirements. Based on the selected platform and used technology, we can further design algorithms for different network services considering the features and resources of different platforms.

1.4 Optimizing Memory-efficient Algorithms for Low-cost and Scalable Network Services

In this thesis, I introduce my work on providing low-cost and scalable network services by optimizing memory-efficient algorithms. My goal is to enhance user experience. Specifically, my work consists of three parts: 1) design of space-efficient data structures, 2) security and privacy service for IoT devices, 3) design of efficient network functions for network management.

1.4.1 Part I: Design of Space-efficient Data Structures

Approximate membership queries (AMQs) rely on space-efficient data structures to decide whether a queried item is in a large dataset efficiently with false positives.

These data structures (called the AMQ data structures) are essential components of network services, such as data management [84] and firewalls [77]. The goals for the design of AMQ structures are to make a good trade-off between memory cost and false positive rate as well as to provide high insertion/lookup/deletion throughput. There are many existing AMQ data structures, such as Bloom filters [39], the most well-known AMQ data structures, and Cuckoo filters [59], another efficient AMQ data structures that can support deletions. We argue that there still have improvements in the trade-off between memory cost and false-positive rate. Especially, Cuckoo filters can achieve their claimed memory efficiency only in ideal situations with a proper number of items in the dataset. Thus, we designed a space-efficient AMQ data structure called Vacuum filters [142], which can cost the smallest space among all known methods when the false positive rate is $\epsilon < 3\%$. We use experiments to show that Vacuum filters can be used in certificate revocation status checking applications and can achieve a better performance compared to Cuckoo filters and Bloom filters. In addition, Vacuum filters can be used in other applications, such as storage systems [41], Bitcoin [18].

1.4.2 Part II: Enhancing the Perception Layer Services: Security and Privacy Service for IoT Devices

More connected IoT devices have raised security and privacy issues. Many literature reviews discussed these issues for the IoT environment. Mendez [102] pointed out that IoT devices demand the following four main sets of security requirements to be considered as secure: 1) secure authentication; 2) secure bootstrapping and transmission of data; 3) security of IoT data; 4) secure access to data by authorized users. Lopez [95] analyzed the privacy issues in detail, which classified privacy problems into two main categories according to the entity whose privacy is being threatened. The first category is user-centric privacy, which comes from the ability of sensors to capture sensitive information about people. The second one is network-centric privacy, which is related to the privacy information about the network itself or the elements being monitored by the network.

Among the security issues, securely authenticating two communication entities and then building secure communication channels are basic and important tasks. A great amount of data, including both public and private information, will be generated, processed, and transmitted by IoT devices. Devices should be authenticated through key management systems [78]. Efficiently secure communication between IoT devices based on cryptographic mechanisms needs to be practically designed in order to securely transmit or share data. In addition, we also need to pay attention to the privacy issues when IoT devices communicate with remote servers since most of the sensing data contain sensitive information about the users. Thus, a privacy protection framework needs to be designed for IoT devices.

Observing the demand to enhance the perception layer services, I focus on two critical research problems: 1) secure communication between IoT devices; 2) data privacy for IoT devices when they communicate with remote cloud servers. Specifically, I designed a **C**ollaborative **C**ertificate **V**alidation protocol (CCV) to perform a fast certificate validation process, which is a basic step to build secure communication. Besides, I designed a system for **L**ightweight **O**blivious **I**oT **S**ervices called LOIS to protect user privacy when devices communicate with remote servers.

CCV [141]: A Fast Certificate Validation Protocol for IoT Devices. Many current IoT devices rely on a central platform to share data [124]. However, emerging and future IoT devices, such as personal health monitors, unmanned aerial vehicles, robots, and self-driving cars, become multi-functional, self-organized, and interactive. Hence due to scalability and autonomy problems, there may not be a central platform to interconnect all these devices. IoT devices may directly communicate with each other to share data. Public key cryptography (PKC) enables fundamental security protocols for IoT data communication based on a well-functioning public key infrastructure (PKI). The authentication process in protocols relies on PKC to establish a secure channel between two end devices or one device and a server. For example, in an IoT-based healthcare system, wearable sensors that collect human-related data need to securely communicate with other sensors, caregivers, and doctors [106]. Existing approaches modify traditional end-to-end IP security protocols to adapt to IoT environments, such as DTLS [122] and HIP DEX [108], which rely on PKC for handshaking. **Certificate validation is an essential step for the PKC-based security protocols**. Although certificate validation can be completed relatively easily on an ordinary computer, it incurs non-trivial overhead on resource-constraint IoT devices. For example, using an optimized method that requires only one signature verification, certificate validation still costs 1.9 seconds, and certificate-based public-key operations demand 95% of the overall processing time of handshaking on the WisMote platform [92], as reported in [72]. Thus, the efficiency of the certificate validation task is required to be improved to enable efficient data communication. We proposed to utilize the power of distributed caching and explore the feasibility of using the cache spaces on all IoT devices as a large pool to store validated certificates, which can be accessed by any internal device. We designed the CCV protocol based on the idea, which adopts the cooperation strategy in an extensive IoT network and utilizes the overall computation power and storage resources. When one device d needs to validate a certificate that has been validated and cached by another device h in the network, d can request a collaborative certificate validation from h to confirm that the requested certificate matches the cached one. CCV can dramatically save computation resources, which is suitable for IoT devices with limited resources.

LOIS: A Low-cost Packet Header Protection for IoT Devices. Many IoT devices collect data from surroundings or capture users' activities and then transmit data to remote servers. Even the payload of packets that carries the data is encrypted with TLS/SSL, packet headers and traffic patterns can still leak sensitive information such as the device identity or the corresponding users' activities. Hence, it's necessary to protect sensitive packet headers and traffic patterns to protect user privacy. An intuitive solution for protecting packet headers is to use virtual private networks (VPNs) [28] to wrap all traffic through the channel by encrypting the packets, including packet headers. However, the VPN method incurs a considerable cost, reducing the performance. We proposed a system called LOIS to protect packet headers efficiently. The LOIS system is built upon the fact that some major cloud providers, such as Amazon and Google, host a large number of IoT applications for end devices - either by themselves or by their customers. Therefore, each such cloud can offer a unified IP address for all the supported services, which serves as the destination IP address for all the packets to the cloud. Different services can be distinguished by a specific service ID. I utilize the stream

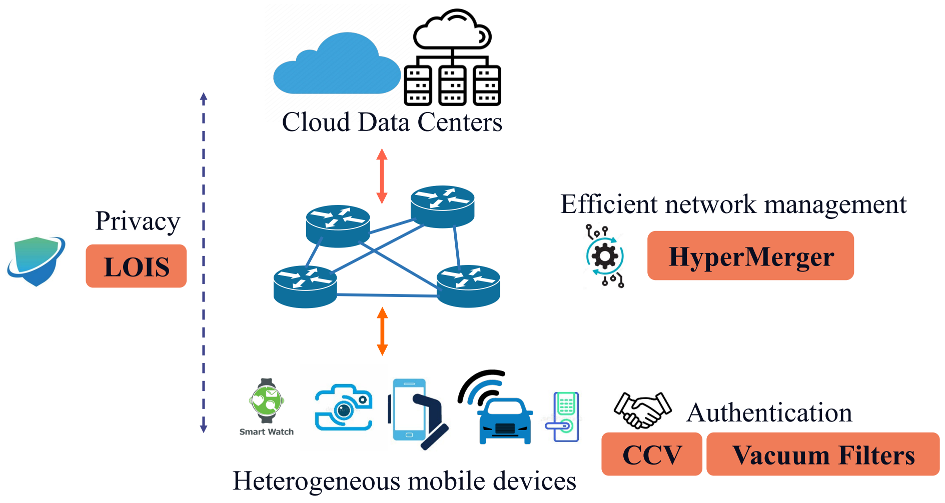


Figure 1.1: Combining three parts in the network

cipher to encrypt sensitive fields in packet headers to protect users' privacy. To agree on the same stream cipher for any packet between senders and receivers, I built a list of one-time keystreams for each service and shared it between senders and receivers. Then LOIS used an identifier to specify which keystream is used for the packet. The identifier is attached to a packet, and attackers cannot infer the used keystream by viewing the identifier. LOIS can protect users' privacy with much less overhead compared to VPNs.

1.4.3 Part III: Design of Efficient Network Functions

Network functions (NFs) such as packet forwarding, load balancers, and packet measurements play an important role in the network support layer. A line of research applies space-compact data structures for individual NF. However, it's common that a network device should support multiple co-existed NFs simultaneously. Hence, in my thesis, I focus on how to design efficient consolidated algorithms to support co-existed NFs. Since it's not practical to manually design a consolidated algorithm for every combination of different co-existed NFs, I designed HyperMerger, the first automatic tool that takes multiple NF algorithms as inputs and generates a consolidated and resource-efficient data plane algorithm to serve multiple NFs. HyperMerger utilizes the P4 platform as a bridge to automatically generate required programs for NFs. The current implementation of HyperMerger takes NF algorithms represented in P4 programs as inputs and outputs P4 and C codes that can be run on hardware and software network devices. But our proposed methods in HyperMerger can be extended with data plane algorithms written in languages other than P4. Fig. 1.1 shows the combination of these works to build an efficient environment for end devices when they rely on network for services.

The remainder of this thesis is structured as follows. Chapter 2 introduces our designed Vacuum filters and uses experimental experiments to show the applicability of Vacuum filters to real applications. Chapter 3 shows our CCV to provide fast certificate validations for IoT devices. Chapter 4 presents a detailed design of LOIS to protect privacy when IoT devices communicate with remote servers. Chapter 5 shows our work of HyperMerger to automatically generate consolidated algorithms based on space-efficient data structures to support co-existed NFs. Finally, Chapter 6 concludes the contributions of my thesis work and discusses the future work.

Chapter 2

Vacuum Filters: A More Space-Efficient and Faster AMQ structure

In this chapter, we introduce the detailed design of our proposed AMQ structure called Vacuum filters and discuss the applications of Vacuum filters.

AMQ structures, such as the well-known Bloom filters [35], are essential components of numerous practical computer software and systems, such as Google Bigtable [41], Apache Cassandra [84], Google Chrome, the content distribution network Akamai [96], Bitcoin [18], and Ethereum [22]. The most attractive feature of the AMQ structures is their **memory efficiency**, with the trade-off on allowing few false positives. Compared to an error-free representation of a set of items, such as a hash table, an AMQ structure can work on devices with limited memory resource (network routers,

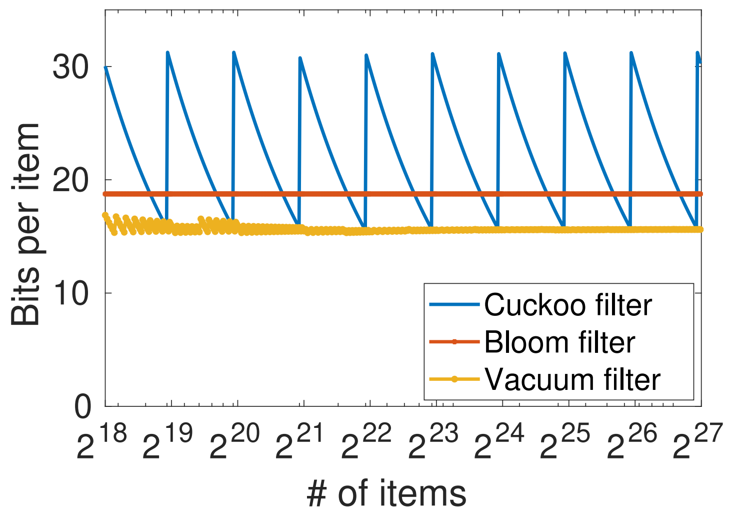


Figure 2.1: Bits per item vs. # of items

switches, and IoT devices), or in a higher level of the memory hierarchy (cache vs main memory, or main memory vs disk). For example, Bloom filters have been extensively used in reducing disk I/O [35,41], avoiding unnecessary remote content lookups [60,96], network functions [54, 88, 132, 146, 151], services on mobile and IoT devices [86], and many data management applications including distributed joins and semi-joins [109], indexing [30], auxiliary metadata [41, 53], and query processing problems [85]. The recently proposed cuckoo filters [59] improves Bloom filters in ideal-case space-efficiency and enabling deletions.

We argue that there are still improvements to design AMQ structures to better trade-off between space cost and false-positive rates. Thus, we design *vacuum filters*, a type of AMQ data structures that cost the **smallest space among all known methods**, i.e., more memory-efficient than Bloom filters, cuckoo filters, and other AMQ structures, when the false positive rate $\epsilon < 3\%$. The name is from vacuum packing which uses the least space to pack items. To understand the space and false positive rate trade-offs of AMQ data structures, we show the empirical results to better illustrate our idea and contributions. Fig. 2.1 shows the memory cost of the most representative data structures for AMQs (in bits per item), including the Bloom filters [35], Cuckoo filters [59], and vacuum filters (this work). The false positive ϵ is set to 0.01%, a common requirement of many applications [38, 59]. Compared to Bloom, vacuum filters can reduce the memory cost by more than 3 bits per item, resulting in $\sim 15\%$ space saving. Compared to a cuckoo filter, a vacuum filter only costs 50% space in the worst cases and saves 25% space on average. Vacuum filters cost less memory than Bloom filters and other AMQ structures when $\epsilon < 3\%$. The advantage of space-efficient becomes larger when the target ϵ is smaller. Although Bloom filters cost less memory when $\epsilon > 3\%$, 3% is considered too much for most AMQ designs. It is common that applications require $\epsilon < 0.1\%$ or even $< 0.01\%$, such as the examples in [33, 38, 59, 86, 88, 146]. In addition, vacuum filters and cuckoo filters provides similar lookup throughput and both are much faster than other AMQ structures. We show a brief comparison among many popular AMQ structures in both space and lookup throughput in Table 2.1. Vacuum filters show the space and throughput advantages compared to all other methods.

Since AMQ data structures have been so widely used for many fields of computing technologies, we argue that $> 20\%$ space reduction compared to Bloom and cuckoo filters (average case) is a **significant contribution**. Considering that AMQ data structures are used in fast memory (SRAM, TCAM, cache) that are expensive and power-hungry, such memory efficiency becomes especially important to reduce device cost and avoid unnecessary device updates. In addition, fast memory is usually shared

Table 2.1: Comparison of AMQ structures in space, throughput, and supporting dynamics. D: deletions; I: duplicate insertions; R: set resizing. Results are based on the experiments when false positive rate ϵ from 0.01% to 0.1%.

Data structure	Space	Thrpt	Dynamic
Bloom F. [35]	1x	1x	I
Dele. BF [123]	~1.13x	~1.95x	D,I
Blocked BF [119]	~1.12x	~5.8x	I
Count. BF [60]	4x	<1x	D, I
Count. Quotient F. [115]	0.87x to 1.45x	~4.7x	D, I, R
Cuckoo F. [59]	0.84x to 1.5x	~10x	D
Morton F. [38]	0.88x to 1.1x	~6x	D
Vacuum F.	0.84x to 0.91x	~11x	D, I, R

by multiple applications. For example, the SRAM on network switches and routers needs to supports network functions including forwarding tables [146, 148], multicast [88], traffic measurement [147, 151], packet caching [27], and load balancing [65, 99]. Reducing the AMQ cost benefits a variety of functions.

In addition, many practical applications require AMQ data structures to support dynamics, including insertions, deletions and set resizing. A well-known limitation of Bloom filters is that they cannot support deletions. To allow deletions, cuckoo filters store duplicate fingerprints. As a result, a cuckoo filter may crash due to table overflow when inserting duplicate items, which frequently happens in practical applications, as explained in Sec. 2.1. Moreover, neither Bloom nor cuckoo filters allow set resizing. The only AMQ method known for set resizing is quotient filters [33], which cost more space than Bloom and provide lower throughput than cuckoo.

Our important observation is that an AMQ data structure cannot support both deletions and duplicate insertions unless it allows reconstruction from the complete set. The major problem of current AMQ reconstruction is that it has to be executed in a large but slow memory because a reconstruction needs to access the complete item set. During that time, the AMQ structure running in the fast memory is unable to answer queries in order to achieve consistency. We resolve this problem by proposing a new update framework for vacuum filters called IUPR (Instant Updates and Periodical Reconstructions) to support deletions, duplicate insertions, and set resizing. IUPR is a good fit for legacy memory hierarchy and network architectures. For example, the vacuum filter can be run in the main memory of a query server, while the construction can be conducted on a backend storage server to access the set of items. In the widely adopted software defined networking (SDN) paradigm [5, 111], the vacuum filter is used in network switches with limited memory, the construction program can be run in the SDN controller on a server, and the vacuum filter updates are achieved via standard APIs such as P4 [37].

In a nutshell, the vacuum filter has unique advantages: **1)** its memory cost is the smallest among existing methods; **2)** its query throughput is higher than most other solutions, only slightly lower than that of cuckoo in very few cases; and **3)** it supports practical dynamics using the memory hierarchy in practice. No existing method can achieve all of them. Since AMQ data structures have been widely adopted and memory efficiency is their most essential feature, the $> 20\%$ space reduction is **fundamental improvement rather than a small increment**.

2.1 Related Work

This section introduces existing AMQ data structures.

Bloom Filter. Bloom filters (BFs) [35] are the most well-known AMQ data structures. A Bloom filter represents a set of n items $S = x_1, x_2, ..., x_n$ by an array of m bits. Each item is mapped to k bits in the array uses k independent hash functions $h_1, h_2, ..., h_k$ and every mapped bit at location $h_i(x)$ is set to 1. To lookup whether an item x_i is in the set, the Bloom filter checks the values in the $h_i(x)$-th bit. If all bits are 1, the Bloom filter reports `true`. Otherwise, it reports `false`. A Bloom filter yields *false positives*. The false positive rate is $\epsilon = (1 - e^{-kn/m})^k = (1 - p)^k$. A Blocked Bloom filter (BBF) [119] divides a Bloom filter into multiple small blocks, each block fits into one cache-line. A BBF is cache-efficient because it only needs one cache miss for every query. One limitation of Bloom filter is that it cannot support deletions. Counting Bloom filters [60] allow deletions, which replace every bit by a counter to store the numbers of setting these bits to 1. Introducing counters significantly increases memory cost. The deletable Bloom filter (DIBF) [123] supports deletion by adding and maintaining a collision bitmap. Items can be deleted with a probability.

Quotient Filters. A quotient filters (QF) [33] uses a hash table to store the fingerprints of the inserted items. The hash table contains 2^q continuous entries. Every entry comprises one slot to store the fingerprint of an item and some extra flag bits to handle hash collision. The space cost of QF is larger than that of BFs. Counting Quotient Filter [115](CQF) improves QF from throughput and memory usage perspectives. It also supports deletions of the inserted items. However, both QF and

CQF's hash tables should have the number of entries to be a power of two, which hurts the memory efficiency.

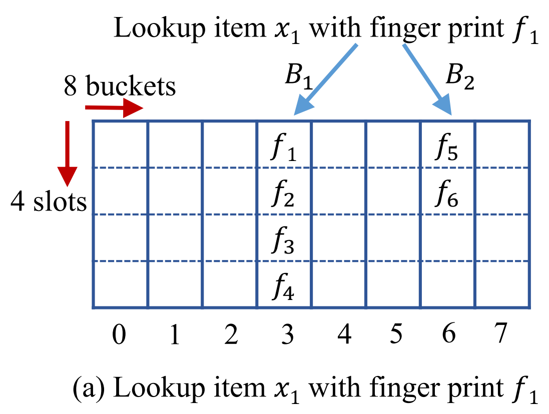
Cuckoo Filters. The recently proposed cuckoo filters (CFs) [59] improves Bloom filters in two aspects. First, in ideal cases, cuckoo filters cost smaller memory than the space-optimized Bloom filter when the target false positive rate $\epsilon < 3\%$. Second, cuckoo filters support deletion operations without extra memory overhead. A cuckoo filter is a table of m *buckets*, each of which contains 4 *slots*. Every slot stores an l-bit *fingerprint* f_x of an item x. For every item x, the cuckoo filter stores its fingerprint f_x in one of two candidate buckets with indices $B_1(x)$ and $B_2(x)$:

$$B_1(x) = H(x) \mod m$$

$$B_2(x) = \texttt{Alt}(B_1(x), f_x)$$

where H is a uniform hash function and function $\text{Alt}(B, f) = B \oplus H'(f)$, where H' is another uniform hash function. It is easy to prove: $B_1(x) = \texttt{Alt}(B_2(x), f_x)$ which means, using f_x and one of the two bucket indices $B_1(x)$ and $B_2(x)$, we are able to compute the other index. To lookup an item x, we check whether the fingerprint f_x is stored in two buckets $B_1(x)$ and $B_2(x)$ as shown in Fig. 2.2(a).

Cuckoo filter insertion. For each item x, the cuckoo filter stores its fingerprint f_x in an empty slot in Bucket $B_1(x)$ or $B_2(x)$ if there is an empty slot, as in Fig. 2.2(a). If neither $B_1(x)$ nor $B_2(x)$ has an empty slot, the cuckoo filter performs the **Eviction** process. It randomly chooses a non-empty slot in bucket B (B is one of $B_1(x)$ and $B_2(x)$). The fingerprint f' stored in the slot will be removed and replaced by f_x. Then f' will be placed to a slot of the alternate bucket $\texttt{Alt}(B, f')$ of f', as shown


Lookup item x_1 with finger print f_1
8 buckets
4 slots
B_1
B_2
f_1
f_2
f_3
f_4
f_5
f_6
0 1 2 3 4 5 6 7
(a) Lookup item x_1 with finger print f_1

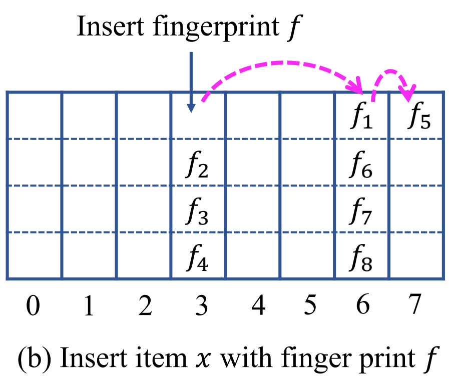


Figure 2.2: Example of a cuckoo filter

in Fig. 2.2(b). If the alternate bucket is also full, the cuckoo filter recursively evicts an existing fingerprint f'' in Bucket $\texttt{Alt}(B, f')$ to place f', and looks for an alternate slot for f''. When the number of recursive evictions reaches a threshold, this insertion is failed and a reconstruction of the whole filter is required.

Though there have been continuous studies of the variants of cuckoo filters [38, 104, 144], we observe that **two fundamental limitations** prevent cuckoo filters from being widely used as a replacement of Bloom filters.

1. The claimed advantage of cuckoo filters in memory-efficiency can only be achieved in ideal situations, i.e., the number of items is around 3.8×2^x for an integer x. In a generalized case where the number of items could be arbitrary, cuckoo filters may need as much as $\sim 50\%$ extra memory in the worst case and $\sim 25\%$ extra memory in average.

2. To support deletions, cuckoo filters will store duplicate fingerprints and may crash due to table overflow when inserting duplicate items. In practical applications, inserting duplicate items are ubiquitous and cannot be detected by cuckoo filters.

3. Cuckoo filters do not support incremental expansion of the item set, a requirement by many applications.

Dynamic cuckoo filters. A dynamic cuckoo filter (DCF) [46] uses a number of linked homogeneous CFs, can support extension of the key set. Since a lookup needs to check all linked CFs, a DCF has lower throughput and higher false positive rate compared to a CF.

Morton filters. Morton Filters (MFs) [38] are variants of CFs. The main design goal of MF is to provides higher throughput for special hierarchical memory systems. MFs introduce virtual buckets and divide logical buckets into memory-aligned blocks. To support high throughput, MF contains extra bits - overflow flags and bucket counters in every block. MFs are claimed faster than CFs on the ARM architecture. MFs only support certain lengths of fingerprints (hence certain false positive rates), which significantly restricts its application range. Besides, Morton Filters cannot use the semi-sorting optimization in CFs [59].

Other variants of cuckoo filter. The adaptive cuckoo filter [104] reduces the false positive rate by maintain a cuckoo hash table in a slow memory. It changes a stored fingerprint when a false positive is detected. The D-ary cuckoo filter [144] aims to provide higher space utilization by increasing the number of candidate buckets for each key. However, it increases the time cost of insertions and lookups.

2.2 Design of Vacuum Filters

2.2.1 Problem statement

A vacuum filter is an AMQ data structure for a set of items, which supports **insertion**, **lookup**, and **deletion** operations. The construction of a vacuum filter can be implemented as serial insertions of all items in the given set. When executing the lookup operation for a queried item x, the vacuum filter should return either `positive`, indicating that x is in the set, or `negative`, indicating that x is not in the set. Similar to most other AMQ data structures [33,35,59], a vacuum filter may report false positive results, but never report false negative results. A vacuum filter utilizes a table-based structure to store fingerprints, similar to those used in quotient filters [33], cuckoo filters [59], and Morton filters [38]. Each fingerprint is a brief representation of the key of an item. If the fingerprint of a queried item is found in certain buckets of the table, the AMQ structure returns `positive`. Compared to cuckoo filters, the typical table-based AMQ structures, vacuum filters have the following main advantages by resolving several challenges that are not addressed in prior methods. 1) Vacuum filters are more space-efficient than cuckoo, Bloom, quotient, and Morton filters. Compared to cuckoo filters, vacuum filters reduce the space cost by $> 25\%$ on average. 2) Vacuum filters provide higher lookup and insertion throughput, due to better data locality. 3) Vacuum filters can replace cuckoo filters in most applications.

2.2.2 Where to gain extra space-efficiency and throughput

Both vacuum and cuckoo filters use the table structure: A table has m buckets and each bucket has 4 slots to store fingerprints, as shown in Fig. 2.2. When a new fingerprint is inserted and its both buckets are full, there should be a way to evict an existing fingerprint to its alternate bucket by using the function $\texttt{Alt}()$. If the alternate bucket is again full, another fingerprint in the alternate bucket will be recursively e-victed. The most important feature of $\texttt{Alt}()$ is to ensure that $B_2 = \texttt{Alt}(B_1, f) \mod m$ and $B_1 = \texttt{Alt}(B_2, f) \mod m$, for a fingerprint f and its alternate buckets B_1 and B_2. In addition, compared to the function $\texttt{Alt}()$ used in cuckoo, we want to achieve **two desired properties** of $\texttt{Alt}()$ in the new design.

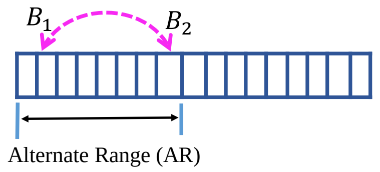
1. The fingerprints should be spread evenly over all buckets by $\texttt{Alt}()$ to achieve high table *load factor* without any restriction on the number of buckets. The load factor is the proportion of filled slots in a table – an important metric for memory efficiency.
2. The two alternate buckets should be stored with a certain level of *locality*: they could co-locate in a same cacheline, page, or big page to reduce the memory access cost.

Note cuckoo filters use $\texttt{Alt}(B, f) = B \oplus H'(f)$, which relies on the assumption that m must be a power of two. It may cause significant space waste. In a case when we need 1025 buckets, we have to use $m = 2048$, almost 50% space waste. Simply replacing $\oplus$ with $+$ does not resolve this problem. It can work with an arbitrary m but insertions may easily fail with a low load factor of the table when m is not a power of two. Also there is very little locality using this `Alt`() function. Two buckets can be arbitrarily far away across the table. Hence each insertion operation may access the memory in different cachelines and pages, resulting in many cacheline and translation lookaside buffer (TLB) missings. Each lookup or deletion operation needs to access two buckets in different cachelines and pages. Assuming the structure runs in the main memory, these cache missings are considered the bottleneck operations of an AMQ structure. We run a set of experiments with $m = 2^{28}$ and every bucket occupies 8 bytes. The results show the two alternate buckets of cuckoo filters never fit into the same 64-byte cache line or the same 4K-byte page.

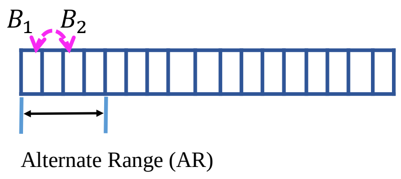
The alternate function of vacuum filters achieves these two goals. We firstly model the problem as a "Balls into Bins" problem [120] and introduce the concept of *alternate ranges* to balance the tradeoff between the load factor and data locality. Based on these results, we propose an optimized *multi-range alternate function* that achieves both good data locality and high load factor.

2.2.3 Alternate Ranges

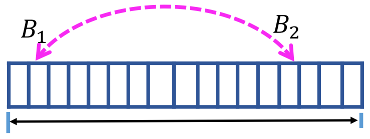
To maintain a certain level of locality, we may divide the whole table into multiple equal-size chunks, each of which includes L consecutive buckets and L is a power of two. Hence m is a multiple of L, instead of being restricted to a power of two as in cuckoo filters. The two candidate buckets of each item should be in the same chunk. For each item x, we compute the indices of the two alternate buckets using



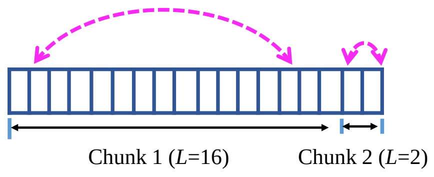
(a) Example of alternate range



(b) Small AR: good locality, good flexibility, but low load factor



(c) Large AR: high load factor, but bad flexibility and bad locality



(d) Chunked design. Large chunks have bad locality. Small chunks cause low load factor

Figure 2.3: Selecting a proper AR is non-trivial

$B_1(x) = H(x) \mod m$ and:

$$B_2(x) = \texttt{Alt}(B_1(x), f) = B_1(x) \oplus (H'(f) \mod L) \tag{2.1}$$

Note we change the alternate function slightly. It is easy to prove that $\lfloor B_1(x)/L \rfloor = \lfloor B_2(x)/L \rfloor$, which means the two buckets fall into the same chunk. This method is denoted as chunked-CF. Hence by knowing the fingerprint f and one of the alternate buckets, we can always compute the other alternate bucket using Equation 2.1, because we also have $B_1(x) = \texttt{Alt}(B_2(x), f)$. For a better illustration, we call the length of chunk, L, as the *alternate range* (AR), shown in Fig. 2.3(a).

Determining a proper size L of the alternate range is challenging. If the AR is small, as shown in Fig. 2.3(b), the filter provides good data locality since the two alternate buckets are very close to each other and likely to be in the same cache line or page (called *co-located*). However, a small AR can cause *fingerprint gathering*. Fingerprint gathering means all alternate buckets of many fingerprints are in a small range

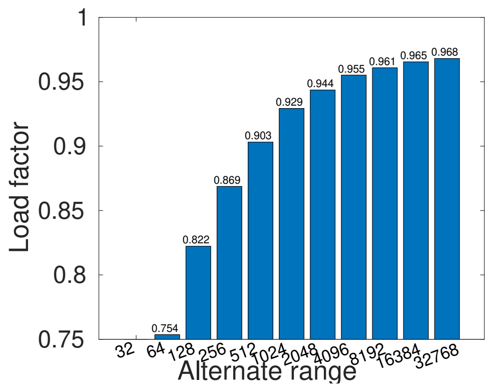


Figure 2.4: Load factor vs. ARs

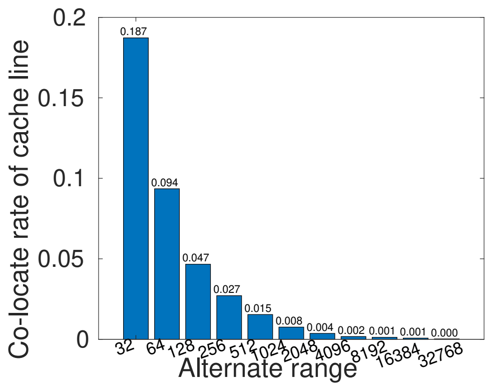


Figure 2.5: Rate of two alternate buckets in a same cacheline

of buckets. The search space of the eviction process is limited, hence eviction loops are likely to happen and the insertions can easily fail. A large AR (Fig. 2.3(c)) can avoid fingerprint gathering and provide high load factor, but its locality becomes bad. Besides, the flexibility of the table is limited, because the number of buckets should a multiple of L – extra buckets may be used and the space cost increases.

We show the experimental results of the load factor in Fig. 2.4, the rate of two alternate buckets in a same cacheline in Fig. 2.5, and the rate of two alternate buckets in a same page in Fig. 2.6, by varying the AR size L. The dilemma is obvious from the results: small ARs cause low load factors and large ARs cause bad locality.

An existing work of chunked hash table [98] fixes the number of chunks to 256 empirically. We improve this design by calculating the minimum AR size based on the number of items n and the target load factor α. The first algorithm `LoadFactorTest`(n, α, r, L) is to test whether a specific alternate range, L, can achieve the target load factor α given the number of items n. r is a parameter that shows the ratio of inserted items in the total number of items. Given n, the target load factor α, and the number of slots per bucket b (4 in our case), we can calculate the number of buckets $m = \lceil n/4\alpha L \rceil L$. The whole table of buckets is separated into $c = m/L$ chunks. The items are randomly distributed to the chunks. We can model this problem as the "Balls into Bins" problem [120]. The goal is that all these chunks will not be overwhelmed by the inserted items. The "Balls into Bins" model provides an estimation of the maximum load and we use a loose estimation from [120] to calculate the maximum load with high probability:

$$\texttt{EstimatedMaxLoad}(n, c) = \frac{n}{c} + \frac{3}{2}\sqrt{2\frac{n}{c} \cdot \log(c)} \tag{2.2}$$

If the estimated maximum load is smaller than the capacity of each chunk P, we consider this L is good to use and the algorithm returns 'Pass'. Otherwise it returns 'Fail'.

Then the second algorithm selects the minimum AR size that can pass the load factor test to achieve good locality, shown in Algorithm 1. For example, when $m = 2^{25}$ and expected load factor $\alpha = 0.95$, the algorithm will select $L = 32768$ as the AR size L.

2.2.4 Multi-Range Alternate Function

Algorithm 1: `RangeSelection`(n, α, r)

```
$L = 1$ ;
while LoadFactorTest$(n, \alpha, r, L) \neq Pass$ do
    $L \leftarrow L \times 2$
end
return $L$
```

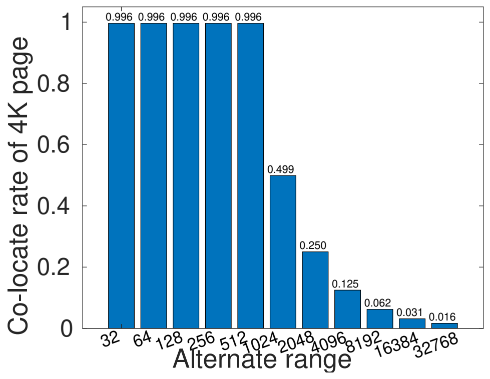


Figure 2.6: Rate of two alternate buckets in a same page

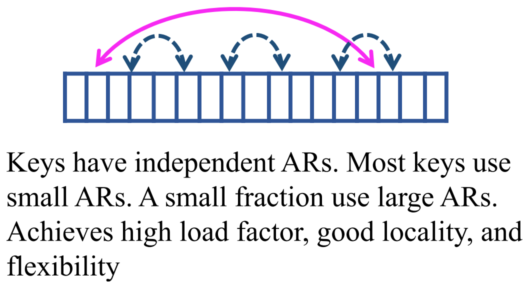

Keys have independent ARs. Most keys use small ARs. A small fraction use large ARs. Achieves high load factor, good locality, and flexibility

Figure 2.7: Vacuum design for alternate ranges

We shall not stop with a fixed AR size as we showed that small ARs cause low load factors and large ARs cause bad locality. To achieve the best of both worlds, we propose a multi-range alternate function. Our idea is inspired by the road network, which usually consists of a large portion of short local roads and a small portion of long highways. We allow every item to have an *independent alternate range*. Also, most items have small alternate ranges to achieve locality and a small number of items have large alternate ranges to avoid low load factor, as shown in Fig. 2.7.

For example, given the target load factor $\alpha = 95\%$, we may allow 25% items to use a large AR and 75% items to use a much smaller AR. Denote the large AR as L_0 and small AR as L_1. The AR size calculated by Algorithm 1 can be considered as the upper bound of L_0 to achieve α. Hence we set $L_0 = \texttt{RangeSelection}(n, \alpha, 1)$. Next, we consider the small AR L_1. Note that a large chunk with size L_0 and a small chunk with size L_1 may overlap as shown in Fig. 2.7. Hence the items with AR L_0 may exist in a small chunk but will not always take the space of the small chunk – it can be evicted to a bucket outside the small chunk. We can re-use the range selection algorithm for these

Table 2.2: Determine the # of AR to balance the load factor and locality. 100 random tests for 2^{27} items. One failed insertion means the final load factor is lower than 95% in a run.

Metric \ # of ARs	1	2	4	8
Average Load Factor	96.6%	96.3%	96.1%	95.3%
Insertion Fail Rate ($\alpha = 95\%$)	0%	0%	1%	7%
Co-locate in a 64B cacheline	0.00%	4.69%	7.03%	9.86%
Co-locate in a 4KB page	0.83%	50.61%	75.00%	81.03%
Co-locate in a 4MB page	95.32%	98.44%	99.22%	99.61%

75% of items of AR L_1 to compute $L_1 = \texttt{RangeSelection}(n, \alpha, 0.75)$.

Moreover, we can use more than two ARs. For example, if we want to have K different ARs, we can set those ARs by the calculation method above. Denote those ARs as L_0 to L_{K-1}. Let $L_0 \geq L_1 \geq \ldots L_{K-1}$. For the $i-$th AR, $\alpha * (1 - i/K)$ of items should have their ARs smaller than or equal to L_i. So we set $L_i = \texttt{RangeSelection}(n, \alpha, 1 - i/K)$.

How many ARs should we use eventually? From the implementation perspective, we set the number of ARs as a power of two, so we can assign ARs to the items by their least significant bits. We test different numbers of ARs to find the best configuration and show the results in Table 2.2. The result shows that when we use one AR, it cannot achieve good data locality (co-locate rates). With more ARs, the fail cases happen more frequently. To balance both the load factor and locality, we use 4 different ARs in the final design of vacuum filters.

The final design of the alternate function `Alt()` is presented in Algorithm 2. All items will be divided into four (roughly) equal-size groups and each group uses a AR size determined by the calculation of `RangeSelection()`.We double the smallest alternate range $L[3]$ to avoid fingerprint gathering that will cause insertion failures. Due to the loose maximum load estimation, it causes relatively bad performance when there is a small number of keys. Thus, we design another alternate function that is shown as Algorithm 3 when the number of keys is smaller than 2^{18}.

Algorithm 2: Vacuum Filter : `Alt`(B, f)

```
if $L$ *is not initialized* then
    for $i = 0; i < 4; i{+}{+}$ do
        $L[i] = \texttt{RangeSelection}(n, 0.95, 1 - i/4)$
    end
    $L[3] = L[3] \times 2$ // Enlarge $L[3]$ to avoid failures ;
end
$l = L[f \mod 4]$ // Current alternate range ;
$\Delta = H'(f) \mod l$ ;
return $B \oplus \Delta$ ;
```

Algorithm 3: Vacuum Filter : `Alt`(B, f)

```
$\Delta' = H'(f) \mod m$;
$B' = (B - \Delta') \mod m$ ;
$B' = (m - 1 - B' + \Delta') \mod m$;
return $B'$;
```

2.2.5 Optimization for insertion

The insertion algorithm is slightly different from the recursive eviction process introduced in cuckoo filters [59]. To further optimize space utilization and throughput, we introduce optimization in the insertion process.

The proposed recursive eviction is different from that in cuckoo hashing [114]. The hash table can be viewed as an undirected graph, where each bucket is a vertex and each item can be considered as an edge that connects the two alternate buckets of the item. Cuckoo hashing uses a “random eviction” scheme to look for an empty slot, which can be considered as a random depth-first search (DFS) of the graph. The random DFS scheme is easy to implement. However, as table occupancy grows, the random DFS scheme needs to examine more buckets and thus require more eviction steps to find an empty slot. A breadth-first-search (BFS) scheme for cuckoo hashing is proposed in [89]. For a full bucket of fingerprints, evicting each fingerprint will start a possible eviction path. The BFS scheme has a broader searching space and thus may reduce the number of evictions to find an empty slot. However, it needs to maintain an extra queue to find the path. In vacuum filters, we combine the advantages of both BFS and DFS. When searching for an empty slot, we look ahead one step. Specifically, when we have to evict an existing fingerprint from the two full buckets of the inserting item, we traverse all alternate buckets of the 8 fingerprints. If there is an empty slot among these buckets, then we can evict the corresponding fingerprint to the empty slot and finish the eviction process. This optimization increases the success rate of insertion and hence improve both the load factor and insertion throughput. In addition we do not need to maintain an extra queue. Algorithm 4 shows the vacuum filter insertion algorithm. Besides, semi-sorting is a technique to reduce the space cost of storing fingerprints, introduced in an early work [36]. Vacuum filters provide an option of using semi-sorting, similar to cuckoo filters [59].

2.2.6 Lookup and deletion

To lookup an item x, it first computes two candidate buckets $B_1(x) = H(x)$ and $B_2(x) = \texttt{Alt}(B_1(x), f)$. If the fingerprint f_x matches one fingerprint stored in the two buckets, the algorithm returns `positive`. Otherwise it returns `negative`. When we generate the first candidate bucket for an item, we need a modulo operation to map the 32-bit hash values of the bucket indices to $[0, m-1]$. In cuckoo filters, when m is a power of two, the modulo can be replaced by a simple bit-wise `AND` operation to increase the speed of calculation – and hence improve the lookup and insertion throughput. However, when m is not a power of two as in vacuum filters, the bit-wise `AND` operation fails, which forces us to use another method.

What we need is a map function, which maps 32-bit hash value to $[0, m-1]$ uniformly to avoid the collision. We adopt the method from [1]: $\text{map}(x, m) = (x \cdot m) \gg 32$. If x is uniformly distributed on $[0, 2^{32} - 1]$, then the first multiplication scale it to a distribution over $[0, (2^{32} - 1) * m]$. Then the right shift operation compress the interval to $[0, m-1]$. This method generates a roughly uniform distribution with only two light instruction, which is comparably fast to the `AND` operation.

The deletion algorithm for item x is simple. If there is a fingerprint equal to f_x in the two candidate buckets of x, the vacuum filter removes this fingerprint.

Algorithm 4: Vacuum Filter : Insert(x)

```
$f = H'(x)$ // $H'$ is the fingerprint function;
$B_1 = H(x)$, $B_2 = \texttt{Alt}(B_1, f)$;
if $B_1$ or $B_2$ has an empty slot then
    put $f$ into the empty slot ;
    return Success;
end
Randomly select a bucket $B$ from $B_1$ and $B_2$ ;
for $i = 0$; $i < MaxEvicts$; $i$++ do
    // Extend Search Scope
    foreach fingerprint $f'$ in $B$ do
        if Bucket $\texttt{Alt}(B, f')$ has an empty slot then
            put $f$ to the original slot of $f'$ ;
            put $f'$ to the empty slot ;
            return Success ;
        end
    end
    Randomly select a slot $s$ from bucket $B$ ;
    Swap $f$ and the fingerprint stored in the slot $s$ ;
    $B = \texttt{Alt}(B, f)$ ;
end
return Fail ;
```

2.3 Analysis results

We present the analysis results on the load factor, false positive rate, space cost and time cost.

2.3.1 Load factor

We give the theoretical analysis for static ARs. The analysis for the multi-range alternate function is more complicated and we will leave it as future work. In the static range case, items will be randomly distributed into different chunks. Each chunk can be viewed as a single sub-table and all evictions of an insertion happen in a single chunk. Therefore, we need two steps of analysis - we first calculate the expected load factor of each chunk, and then check whether the load factor of each chunk can satisfy a certain value of overall load factor of the entire vacuum filter.

Upper bound of the number of items in each chunk. Let the AR be L and the number of buckets is m. There are $c = m/L$ chunks and n items to insert. We utilize the results from the well-studied "Balls into Bins" model [120], in which n balls are uniformly randomly distributed to c bins. In [120], the author gives an upper bound about the number of balls in any bin.

Theorem 1 *Let M be the random variable that counts the maximum number of balls in any bin. Then $\texttt{Pr}[M > k_a] = o(1)$, where $a > 1$ and $k_a = \frac{n}{c} + a\sqrt{2\frac{n}{c}\log c}$, if $n \gg c \log c$.*

In our case, n and c satisfy $n \gg c \log c$. Let $a = \frac{3}{2}$. We have the maximum number of items in any chunk will be smaller than $\frac{n}{c} + \frac{3}{2}\sqrt{2\frac{n}{c}\log c}$ with high probability.

Expected load factor of vacuum filters. In [57], the author provides the analysis of the load factor of cuckoo filters. However, it requires the number of slots per bucket to be 10.916, which does not fit our design. **To our knowledge, there is no theoretical result of the load factor that fits the four-slot table design.**

Hence we use experiments to test the maximum load factor for those chunks. The experiment shows that the load factor of a single chunk can achieve 97% with probability greater than 99%. Hence in the `RangeSelection` algorithm, we fix the overall load factor as 95% and select the AR value such that the upper bound of the number of items does not exceed 97% of the chunk capacity, which ensures that even if the number of items of a single chunk reaches the upper bound, the insertions will not fail with $> 99\%$ probability. As a result, the final design of vacuum filters can achieve high load factor (95%) with $> 99\%$ probability.

2.3.2 False positive rate and space cost

Two factors influence the false positive rate of a vacuum filter: 1) the length of fingerprint l; and 2) the number of slots b in each bucket (usually set to 4). We call an item 'alien item', if it is not in the target item set. In a vacuum filter, the probability that a query of an alien item matches one stored fingerprint (a false-positive match) is at most $1/2^l$. The probability of false positive rate can be computed after $2b$ comparisons. We also need to consider the load factor α of the table of the vacuum filter. Then the expected number of comparisons is $2b\alpha$. The probability of no false hit is $(1-1/2^l)^{2b\alpha}$.

Thus, the upper bound of the total probability of false positive rate is

$$\epsilon = 1 - (1 - 1/2^l)^{2b\alpha} \approx 2b\alpha/2^l \tag{2.3}$$

We can derive the necessary fingerprint length for a given target false positive ϵ:

$$l \geq \lceil \log_2(2b\alpha/\epsilon) \rceil \tag{2.4}$$

For a given number of items n, the whole memory consumption M_V is

$$M_V = (n/\alpha)\lceil \log_2(2b\alpha/\epsilon) \rceil \tag{2.5}$$

where the unit is bit.

The theoretical number of bits per item for vacuum filters (VFs) is $\frac{\log_2(2b\alpha)+\log_2(1/\epsilon)}{\alpha}$. Since the semi-sorting technology use one bit less per item, the space cost per item for vacuum filters with semi-sorting (VFs-ss) is $\frac{\log_2(2b\alpha)-1+\log_2(1/\epsilon)}{\alpha}$.

Given $\alpha = 0.95$ and $b = 4$, the space cost per item for VFs and VFs-ss are $3.07 + 1.05\log_2(1/\epsilon)$ and $2.07 + 1.05\log_2(1/\epsilon)$. Cuckoo filters (CFs) have the same space cost in their best cases, and their average and worst cases need 25% and 50% more space respectively. For counting quotient filters (CQFs) [17], at the same load factor, the space cost per item is $2.24 + 1.05\log_2(1/\epsilon)$. Similar to CFs, CQFs require the number of buckets to be a power of two. Hence the space cost in the average and worse cases is higher than this value. For Bloom filters, the space cost per item is $1.44\log_2(1/\epsilon)$. When $\epsilon < 3\%$, VFs have the lowest space cost compared to other AMQ structures.

2.3.3 Time cost

The time cost for each lookup or deletion of VFs is constant – either of them only needs at most 2 memory accesses.

The analysis of the time cost for each insertion is more complicated. Since all fingerprints are uniformly distributed among the buckets, the probability that a bucket of b slots is full is α^b. Assume the searching process in the insertion algorithm follows the Bernoulli distribution with a successful rate $1-\alpha^b$. Let p be the number of traversed buckets of an insertion. Then the expectation of p under the load factor α, $E(p_\alpha)$ can be estimated by the equation

$$E(p_\alpha) = (1-\alpha^b) + \alpha^b(E(p_\alpha)+1) \tag{2.6}$$

We have $E(p_\alpha) = 1/(1-\alpha^b)$. For the average insertion cost E for serial insertions from load factor 0 to α, we can integrate $E(p_x)$ from 0 to α,

$$E = \int_0^\alpha E(p_x)dx = \int_0^\alpha 1/(1-x^b)dx \tag{2.7}$$

With $b = 4$ and $\alpha = 0.95$, we have $E = 1.3$. Our experimental results show that the average traversed number of buckets is about 1.58, which is close to the theoretical result.

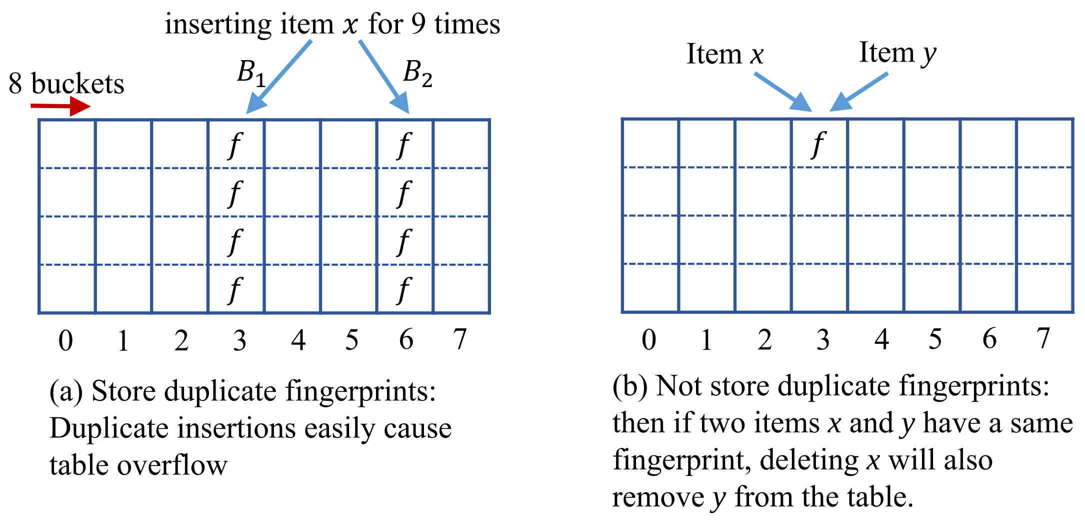

inserting item x for 9 times
8 buckets
B_1
B_2
f
0 1 2 3 4 5 6 7
(a) Store duplicate fingerprints: Duplicate insertions easily cause table overflow
Item x
Item y
(b) Not store duplicate fingerprints: then if two items x and y have a same fingerprint, deleting x will also remove y from the table.

Figure 2.8: Duplicate insertions are difficult to handle

2.4 Vacuum Filters Under Dynamics

2.4.1 Problem statement

Dynamics of the item set include item insertions, deletions, and set resizing. Note if the item insertion rate is approximately equal to the deletion rate, set resizing will not happen. However, if the insertion and deletion rates are not equal in some applications, the set size may change after a period of time. For example, by keeping inserting items to the set, the set may become too larger to fit the current table at a certain time. Bloom filters can deal with insertions but not deletions. Cuckoo filters can handle insertions and deletions when there is no *duplicate insertion*. Duplicate insertions mean a same item may be inserted to the data structure for multiple times. If set resizing or duplicate insertions happen, cuckoo filters may fail or at least become sub-optimal.

Duplicate insertions exist in many practical applications. For example, a proxy server in a content distribution network may use an AMQ structure to represent the current set of cached content in the local network [39,96]. When a proxy server caches a content, it notifies all other servers about the cached content. Obviously, a same content may be cached in different servers by multiple times, hence there will be duplicate insertions. We show the dilemma in dealing with duplicate insertions in Fig. 2.8. As shown in Fig. 2.8(a), if we allow to store duplicate fingerprints, then after 9 times of inserting the same item x, the table will overflow and fail. As shown in Fig. 2.8(b), if we do not allow to store duplicate fingerprints, then two items x and y may have a same fingerprint. Deleting x will also remove y from the table, resulting in future false negatives!

In addition to duplicate insertions, neither cuckoo filters nor counting Bloom filters can deal with set resizing. In fact, no existing AMQ structures work well under duplicate insertions or set resizing.

2.4.2 Instant updates and periodical reconstructions

We plan to solve these problems in a practical model. Most applications make use of a memory hierarchy to construct the AMQ data structures, in which the full data set is stored in a slow and large memory and the AMQ is running in a fast and small memory. For example, the slow memory can be disks and the fast memory can be main memory [35]; the slow memory can be a server and the fast memory can be network devices that use ASICs [39, 149]. The AMQ structure (vacuum filter in our case) is used to response to membership lookups that should be processed in a fast speed. In addition, the vacuum filter should *instantly* update itself when there is any insertion or deletion. However, after a period of time and a number of updates, the vacuum filter may be sub-optimal, e.g., its size does not fit the current item size. At this time, a new version of vacuum filter is constructed by another process (or another machine) using the slow memory. The old vacuum filter running in the fast memory will be replaced with the new version. This method is called IUPR (Instant Updates and Periodical Reconstructions). For every time of reconstruction, duplicate insertions and set resizing are resolved. Hence the vacuum filter can be recovered from the sub-optimal condition.

2.4.3 Instant updates of vacuum filters

One requirement for the self updates on the vacuum is a fast speed to avoid interruption of responding AMQ queries. Our goal is to provide fast updates and the update operations include insertions and deletions. For deletion operations, the requirement is to delete the items which must have been previously inserted [59]. Insertion operations are more complicated to deal with compared with deletion operations. It is because a full vacuum filter needs to be extended to insert more items.

To achieve this goal, we design the Dynamic Vacuum Filter (DVF) inspired by the Dynamic cuckoo filter (DCF) [46] for fast self updates of IUPR. A DCF uses a linked chain of cuckoo filters for some extended space.

However, directly applying DCFs to vacuum filters faces some problems. 1) A DCF is based on the standard cuckoo filter, the load factor of each cuckoo filter is low when inserting a certain number of items, which is not memory-efficient. 2) The design of a DCF incurs high cost for the lookup process. Assuming there are s chained cuckoo filters, the lookup process needs $2s$ memory accesses, which increases the cost when s grows big. 3) The false positive rate increases to $2bs/2^l$ compared to $2b/2^l$ of the original cuckoo filter. 4) A DCF requires that every linked cuckoo filter has the same size of buckets. Thus the number of buckets in each cuckoo filter is decided by the first cuckoo filter. If the initial number of buckets is small, this design could incur many small cuckoo filters when more items come, significantly increasing the cost for the operations. However, in many applications it is hard to pre-determine a proper number of buckets at the system initialization. Therefore DCFs cannot be directly used for vacuum filters.

A DVF uses a linked chain of vacuum filters. A vacuum filter achieves about 95% memory utilization rate without any constraint on the number of inserting items. Thus, the DVF can resolve the first problem of DCFs. Another feature of the DVF is that we do not require all the linked vacuum filters have the same number of buckets, which makes the DVF more flexible in dynamic environments. The number of buckets in each vacuum filter is independently decided.

Insertions. The status of a vacuum filter may be full or not when there are new items to be inserted. Two conditions are used to decide whether the filter is full. 1) The load factor of the filter reaches a defined threshold, e.g., 96%; 2) The current insertion fails. If the filter is not considered full, we can easily insert the item. When the filter is considered full, we create another table and order all tables in a sequence with ascending order of the time of creation. Then the future items will be inserted into the newly created table, called the tail table. If there are multiple tables in the list, during item insertions, we check the status of the tail table instead of traversing all tables in the list to save time. Obviously the lookup and insertion performance will downgrade by chaining more tables. However, IUPR includes the periodical reconstruction phase, which will recover the chained tables to a unified vacuum filter table including the fingerprints of all current items. The process benefits from the property of a vacuum filter that it can be in any size without restricting the value of m.

Lookups. To lookup of an item x, the vacuum filter needs to traverse the tables in the chain. If no table has a matched fingerprint, the vacuum filter reports `negative`.

Deletions. The deletion of an item x is simple. The DVF performs a membership query of x in the whole vacuum filter list. If a corresponding fingerprint of x is found, we delete the fingerprint. One situation caused by deletion operations is that some vacuum filters in the list may get low load factors after deleting a number of items. The sub-optimal tables will be resolved by each reconstruction.

During each update operation, the system cannot support the membership query. Since each of these updates can be finished with in $O(1)$ time in average similar to those in cuckoo hashing [114]. Thus lookup operations will not be significantly interrupted. On the other hand, a reconstruction to create a new version of the vacuum filter may take a longer time. Hence we propose to let a different machine or process to run the reconstruction program *concurrently* with the lookup process.

2.4.4 Periodical reconstruction

To resolve the performance downgrading problem caused by fast updates, we introduce the periodical reconstruction process that creates new versions of the vacuum filter.

Each reconstruction can be triggered in two ways. First, after a time interval T, the construction process reconstructs the vacuum filter from the up-to-date data set. Second, it can be triggered by special events, e.g., the lookup throughput of the vacuum filter is lower than a pre-defined threshold, or the number of updates exceeds a pre-defined threshold. During the reconstruction process, the current vacuum filter running in the fast memory still accepts membership queries and performs fast updates. When a new version of the vacuum filter has been constructed, the construction process will first apply the update operations that happened during the reconstruction time. It is because these operations are not included in the past reconstruction. Then the construction process transmits the new version to the lookup process, and the lookup process replaces the old filter with the new version. Fig. 2.9 is an example of such parallel reconstruction. It shows the timelines of both lookup and construction processes. At time t_1, a reconstruction is triggered and the new vaccum filter should reflect the original item set and the recent updates #1-4. When this reconstruction finishes, there are three more updates happened during this period, namely updates #5-7. Hence the construction process applies instant updates to the new version for 5-7 and sends the new version to the lookup process. The lookup process has self-updated the updates #1-7 and may experience a downgraded performance. It replaces the old vacuum filter with the new version and is recovered to a better performance. Under these dynamics *the downgraded performance of the vacuum filter can always be recovered* after a period of time. The experimental results is shown in Section 2.5.3.

In addition, reconstructions help to keep a very small number of chained tables in the vacuum filter. In DCF that the chain could be arbitrarily long. A long chain

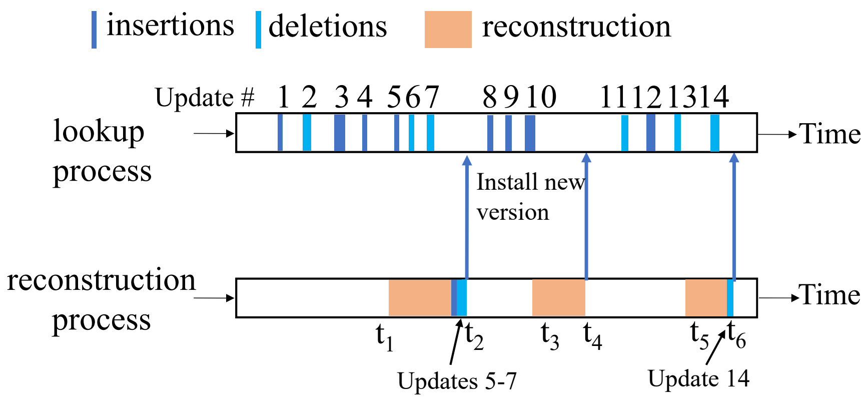


Figure 2.9: Handling concurrency in IUPR

of table will significantly decrease the lookup and update throughput. Two parameters are taken into consideration to achieve a good tradeoff for the dynamic design. The first one is the time interval of triggering the reconstruction process. The second one is the number of buckets for the newly chained table during fast updates. The reconstruction time interval is suggested to be determined in a per-application basis. If we decide the reconstruction time interval T and predict the insertion rate as r_i and deletion rate as r_d, we set the size of the newly chained table to contain $2T(r_i - r_d)$ items, using 2x as a safety margin.

2.4.5 Summary of IUPR

The design of IUPR solves the performance downgrading problem caused by duplicate insertions and set resizing. These problems frequently happen in practical applications but has not been well-addressed by existing methods. IUPR is an extra component to further strengthen vacuum filters that work on practical memory hierarchies. *Note even if IUPR is not used, vacuum filters provide better performance than*

Bloom or cuckoo filters in most concerned metrics including the space cost, throughput, and false positive rate.

2.5 Performance Evaluation

2.5.1 Evaluation methodology

We implement a complete software prototype of vacuum filters including the basic algorithms and the protocols in IUPR. There are two design options of vacuum filters in implementation. One option is that for different lengths of fingerprints, whether we should add padding bits so that no bucket will cross two cachelines. The other option is whether the implementation includes semi-sorting. The implementation of vacuum filters including semi-sorting but no padding is denoted as **VF-ss (No Padding)**. The implementation including padding but no semi-sorting is denoted as **VF**. The implementation including both padding and semi-sorting is denoted as **VF-ss**. We use the implementation of cuckoo filters provided by their authors [16]. The corresponding versions of cuckoo filter are denoted as **CF** and **CF-ss**. For Morton filter (MF), we use the implementation provided by the authors [20]. The implementation of Bloom filters is standardized, we set the number of hash functions to $\lfloor \ln 2(m/n) \rfloor$ in order to achieve the lowest false positive rate [39], where n is the number of items, m is the array size. We also implement the deletable Bloom filters (DlBFs) [123] and blocked Bloom filters (BBFs) [119]. We use open source code for Counting Quotient Filters (CQFs) [17]. The code for Vacuum Filters is available in [21].

Unless otherwise mentioned, the items used for experiments are pre-generated

64-bit distinct integers from random number generators. All experiments are running on a DELL work station with Intel E5-2687W CPU, 3.00 GHz, and 30 MB L3 Cache. The hard-disk is SK-hynix-SC311-S 1TB SSD.

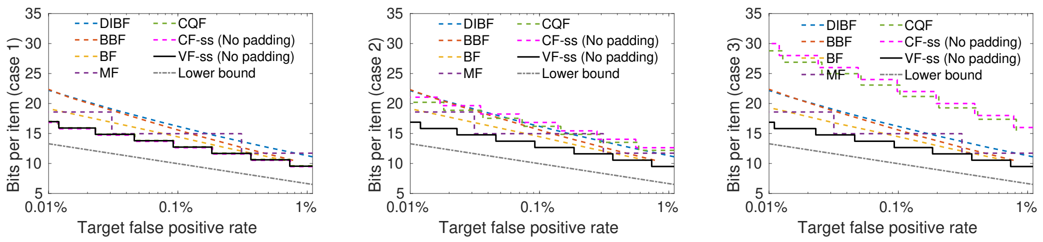
Metrics: We evaluate the following metrics:

- *False positive rate*: measured by querying a filter with non-existing (alien) items and then calculating the fraction of returned positive results. The false positive rate is usually a target value that needs to be achieve by adjusting other parameters.
- *Bits per item*: this metric reflects the memory cost. We count the average number of bits used per item in an AMQ structure to achieve a target false positive rate.
- *Load factor*: measured by the number of bits used to store fingerprints, over the total size of the data structure.
- *Lookup, insertion, and deletion throughput*: measured by the number of lookup, insertion, or deletion operations a data structure can process per second.

Unless otherwise mentioned, for every result shown in this section, we conduct 10 production runs and compute the average.

For the evaluation under dynamic environments, we compare the performance between IUPR-VF method and a modified dynamic vacuum filter (linked-VF) method without reconstruction, which will be illustrated in Section 2.5.3. We also compare IUPR-VF with IUPR-CF.

In addition, we evaluate the gain of replacing Bloom and cuckoo filters with vacuum filters in a real application: checking the revocation status of digital certificates.



(a) Case 1 (b) Case 2 (c) Case 3

Figure 2.10: Memory cost versus false positives

2.5.2 Evaluation of data structures

We compare the performance of vacuum filters (VFs), cuckoo filters (CFs), Bloom filters (BFs), Blocked Bloom filters (BBFs), Deletable Bloom filters (DIBFs), Morton filters (MFs) and Counting Quotient Filter (CQFs). For VFs and CFs, we also compare the performance of their semi-sorting implementation (VF-ss and CF-ss). Note counting Bloom filters (CBFs) have higher memory cost and lower throughput than BFs by the design. Hence we do not need to compare to CBFs.

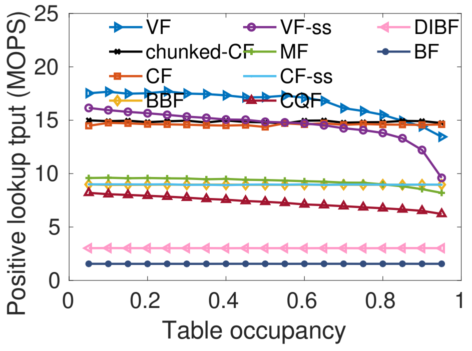
Memory cost and false positives. We evaluate the memory usage with target false positive rates for VF-ss, CF-ss, BFs, BBFs, DIBFs, MFs and CQFs. We show the results of three cases. In Case 1 the number of items $n = 0.95 \times 2^{25}$. In Case 2, $n = 0.75 \times 2^{25}$. In Case 3, $n = 2^{25}$. Note in all three cases, the bits per item of VFs have no clear difference. However for CFs and MFs, Case 1 is considered the best case, Case 2 is considered the average case, and Case 3 is considered the worst case. The selected numbers of items and the results of three cases are the representative ones.

Fig. 2.10 shows the results of the memory cost (in the number of bits per item) versus the target false positive rate. We also show the theoretical lower bound in the figures for comparison, which does not exist in practice. We find that VFs achieve **the**

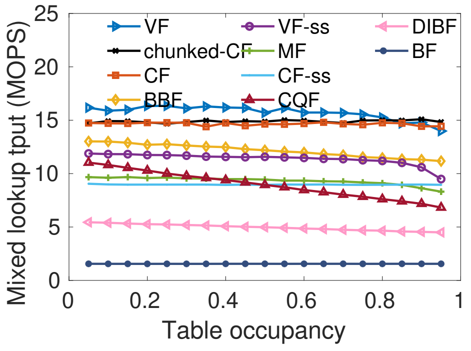
lowest memory cost in all cases, providing 2-5 bits saving per item compared to other methods in the average case under a same false positive rate. The plots of memory cost for CFs, MFs, CQFs, and VFs are all in stair-steps, because they need to use the fingerprints whose length is an integer. When the key numbers are near powers of two (such as Case 3), CFs and CQFs have to create a much bigger hash table to store the fingerprints. As a result, the load factor of a CF will be close 50% and the memory cost is high. In the average case (Case 2), VFs still show an advantages of ~5 fewer bits per item compared to CFs. BFs and the variants DIBFs and BBFs show worse memory and the false positive tradeoff than the VFs in all cases. MFs need more memory than VFs in all cases.

Operation throughput. We fix the fingerprint length for VFs, CFs, MFs, and CQFs to 12 bits. For VF-ss and CF-ss, we select 13 bits per fingerprints for a better alignment. The item number is $2^{27} * 0.95$, and the memory consumption are all fixed to 192MB, which is much bigger than the L3 cache. **Note this setting is a best-case situation for CFs and CQFs. We give advantages to them to show that even under these cases, VFs still outperform these methods. VFs do not rely on these situations. In other situations, VFs may provide more obvious performance gains.**

Lookup throughput. Fig. 2.11 shows the lookup throughput for two cases - 100% of existing items, and 50%-50% mixed of existing and alien items. Our results show that VF-ss is about 1.1x to 1.8x faster than CF-ss for positive lookups. VFs provide higher throughput than all other methods when the table occupancy rate is lower than 90%. In the implementation, VFs may access fewer than two cachelines, and

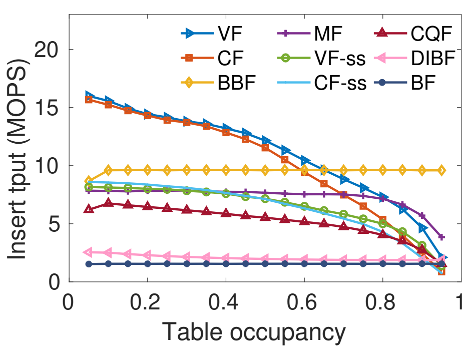


(a) Lookups for existing items

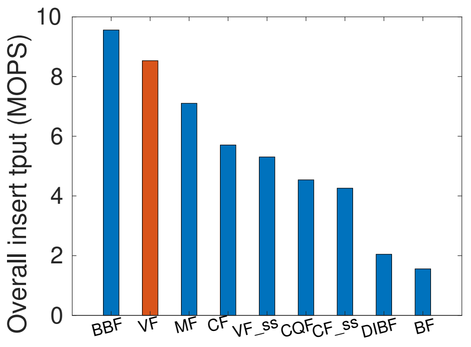


(b) Lookups for mixed items

Figure 2.11: Performance of lookup throughput



(a) Varying occupancy



(b) Overall throughput

Figure 2.12: Performance of insertion throughput

CFs will always access two cachelines. Intuitively, CFs' throughput will be much slower than VFs'. However, since The Intel Xeon micro-architecture has multiple memory accessing units, the two memory accesses may be done concurrently. As a result, the lookup throughput of CFs will be slightly better than VFs when the occupancy rate is high. The number of chunks for chunked-CF is 256. We can find that chunked-CF has almost the same performance as CF. The lookup throughput of BFs and DBFs is always the lowest.

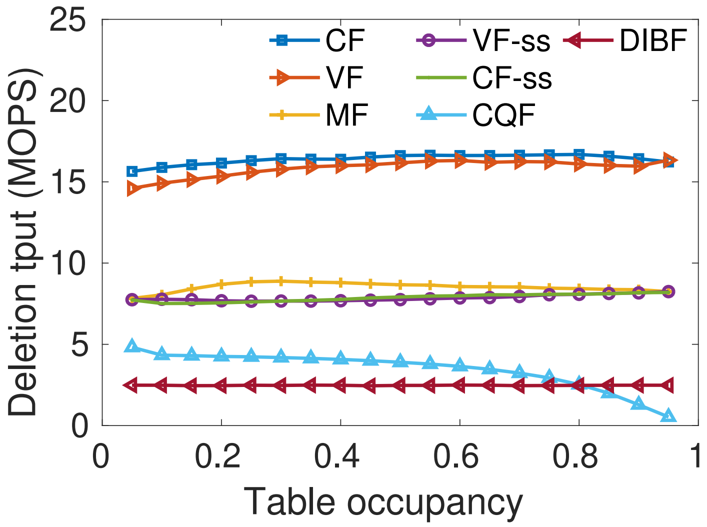
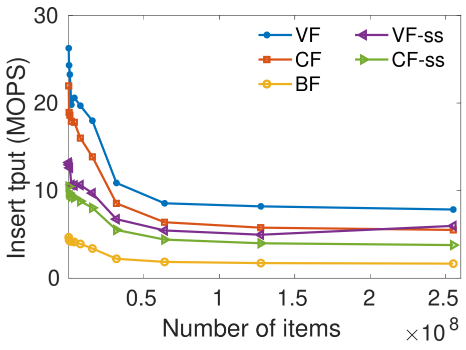


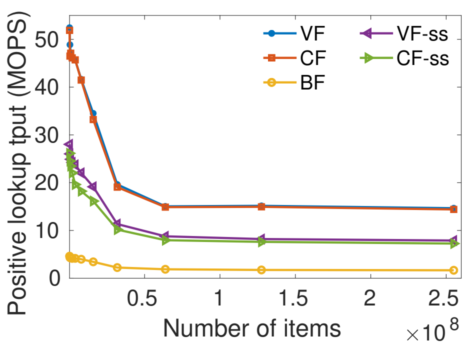
Figure 2.13: Deletion throughput

Deletion throughput. The deletion throughput is shown in Fig. 2.13. We can find that VFs and CFs have similar deletion throughput, higher than most other methods.

Insertion throughput. Fig. 2.12(a) shows the insertion throughput with different table occupancies. VFs have the highest throughput compared to other AMQ structures when the table occupancy $< 65\%$. The insertion throughput of all methods except Bloom filters and their variants decreases with higher table occupancy. When the occupancy is higher than 60%, both CFs and VFs require more bucket accesses to find an empty slot. In these cases, the data locality can significantly affect the throughput. Since the semi-sorting optimization requires more complicated operations, the memory access speed may not be the bottleneck. Hence, VF-ss shows slightly better throughput than CF-ss. BFs need about 9 random memory accesses, which have the slowest insertion speed among all methods. However BBFs are fast in insertions because



(a) Insertion



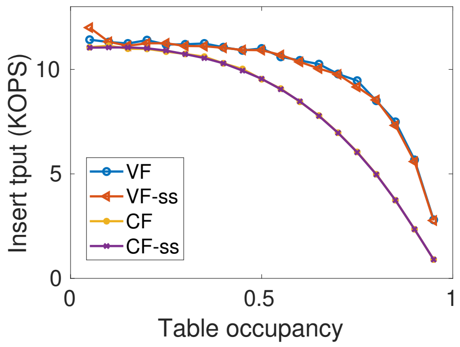
(b) Lookups of existing items

Figure 2.14: Throughput versus num. of items.

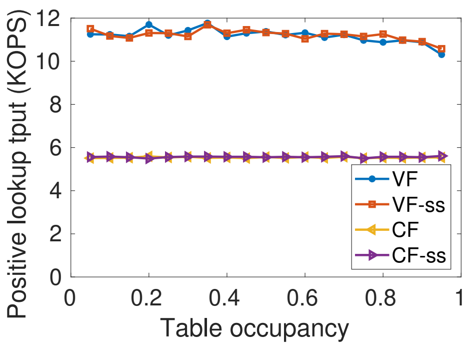
they cause one cache miss at the most. After inserting all items into the data structures, we compute the overall performance of insertion throughput, shown in Fig. 2.12(b). We can find that VF achieves best overall insertion throughput except for BBF. The overall insertion throughput of VFs and VF-ss is higher than that of CFs and CF-ss.

Throughput versus the number of items. We evaluate the throughput for different numbers of items to demonstrate the scalability and stability of the methods. Figs. 2.14(a) and 2.14(b) shows the results. VFs are slightly better than CFs in all cases.

Throughput in SSD. We evaluate the throughput of different methods in the SSD hard disk, where the memory access speed is much slower. And we limit the memory to 100MB. The results are shown in Fig 2.15. The results show that the lookup throughput of VF and VF-ss is around 1.86x to 2.13x faster than those of CF and CF-ss. The insertion experiments also show VFs provides higher throughput than CFs regardless of the occupancy.



(a) Insertion throughput



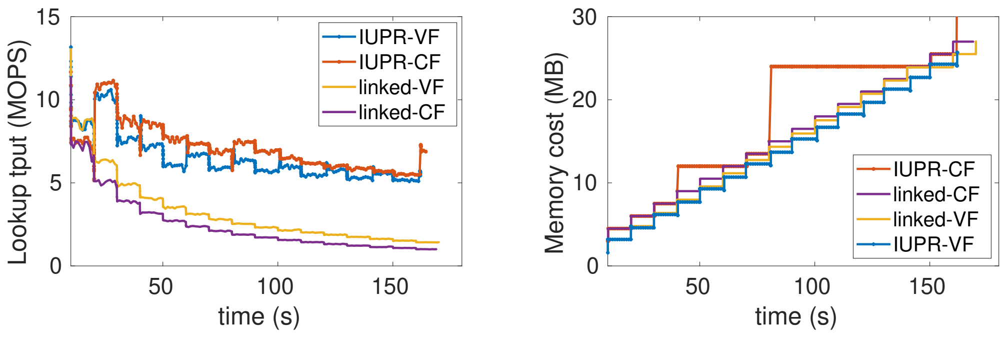
(b) Positive lookup throu

Figure 2.15: Performance of throughput in SSD.

2.5.3 Evaluation under dynamics

In this section, we evaluate the performance of vacuum filters in dynamic environments with frequent item set updates. For the comparison purpose, we build two systems. The **Linked-VF** system keeps creating new tables with items updates and links the tables in a chain. The **IUPR-VF** system uses the proposed IUPR update and reconstruction methods. We also replace vacuum filters by standard cuckoo filters to make a comparison between them. We apply the semi-sorting versions of VFs and CFs to save memory cost.We set length of fingerprints to be 13 bits. The vacuum filters and cuckoo filters are empty at the beginning. Updates happen with a stable frequency. The arrival time of the update events follows the Poisson distribution. The parameter λ denotes the average number of updates happening in the given time interval.

We evaluate two metrics: lookup throughput and memory cost. For the memory cost, we only consider the memory consumption of the AMQ structures running in the fast memory. We consider two parameters that influence performance. 1) Re-



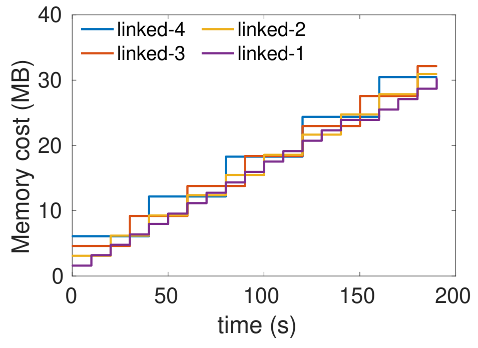
(a) Lookup throughput

(b) Memory cost

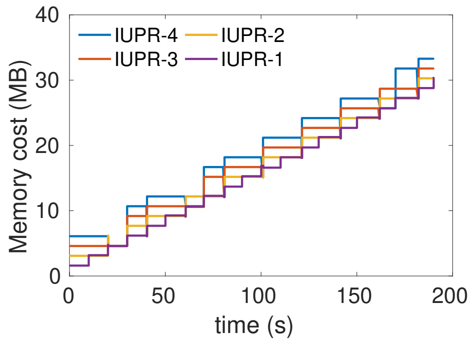
Figure 2.16: IUPR-VF/CF versus Linked-VF/CF

construction interval: In the implementation, we reconstruct the vacuum filter with a given interval. 2) The number of buckets: to choose a proper number of buckets when creating a new table, we set the number of buckets according to a pre-defined number, an estimated number of insertions in the next time interval.

Overall system performance. Fig. 2.16 shows the performance comparison of IUPR-VF, linked-VF, IUPR-CF, and linked-CF. In this set of experiments, the reconstruction interval is set to 20 seconds and λ is set to 10^6. From Fig.2.16(a), we find that IUPR-VF and IUPR-CF have higher lookup throughput than linked-VF and linked-CF over time. The reason is that the systems are always recovered to a good state every 20 seconds by resolving performance downgrading. We find that every 20 seconds, there is a sudden increase of the lookup throughput (in the 20s, 40s, and so on), meaning a replacement of the old version with a new version. However, the lookup throughput of linked-VF and linked-CF continuously decreases due to more linked tables caused by updates. We also find that IUPR-VF and IUPR-CF have a comparable lookup throughput. However, it may require much more memory for IUPR-CF to reconstruct the filter



(a) Memory cost of linked-VF



(b) Memory cost of IUPR-VF

Figure 2.17: Memory cost with different number of buckets

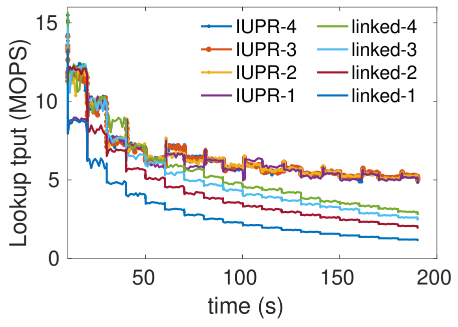


Figure 2.18: Lookup throughput vs. time (IUPR-x denotes that a VF contains about $x * 10^6$ items)

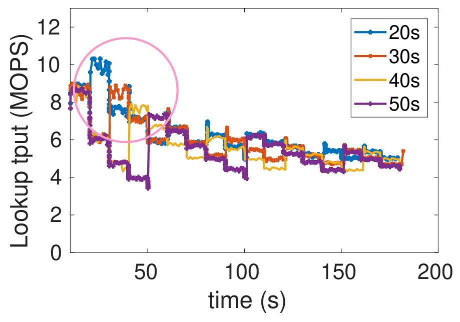


Figure 2.19: Lookup throughput with different reconstruction intervals

due to the restriction of the filter size as shown in Fig. 2.16(b).

Varying number of buckets in a new table. We use experiments to show the performance of varying the number of buckets for each new table. Every 10 seconds there will be about 10^6 new items need to be inserted into the data structure. In this experiment setting, every update we create a new table that can contain 10^6, 2×10^6, 3×10^6, and 4×10^6 items respectively. Fig. 2.18 shows the results of lookup throughput. We find that assigning a larger space when creating a new table has better lookup throughput for the linked-VF because it can decrease the number of table in the list. The drawback is that it consumes more memory when the newly linked vacuum filter is not full of sufficient updates, which can be shown in Fig. 2.17(a). In Fig. 2.17, the memory of both linked-VF and IUPR-VF increases about 1.6 MB every 10 seconds due to the new vacuum filter is created. And we find that the influence of the different size of a vacuum filter is small because of the reconstruction process. A larger number of buckets consume more memory during the system operation. In practice, assigning an estimated space for updates with proper reconstruction makes a good trade-off between the memory cost and lookup throughput.

Varying reconstruction interval. We evaluate the influence of different reconstruction intervals by setting the interval to be 20s, 30s, 40s, and 50s respectively. Fig. 2.19 shows the results of lookup throughput. A smaller reconstruction interval means reconstructions happen more frequently, which is shown in the pink circle in Fig. 2.19. We find that the lowest lookup throughput of 20s is higher than that of 30s, 40s, and 50s. We also find that the smallest reconstruction interval achieves best lookup throughput from the figure 2.19 in the whole process, although it incurs higher computation overhead in the slow memory due to more frequent reconstruction calls. In practice, a real deployment may choose the smallest time interval allowed by the workstation running the update program to achieve a better lookup throughput.

2.5.4 Case study: checking revoked certificates

Vacuum filters can be used in many real applications [18, 66, 83, 86]. In this section, we study a case of using vacuum filters to check the certificate revocation status, which originally uses Bloom filters.

The TLS protocol, which relies on the public key infrastructure (PKI), is the cornerstone of Internet security. One critical problem for the web's PKI is the overlooked certificate revocation. If an erroneous or compromised certificate should be revoked, otherwise the client software will believe that the certificate is valid until it expires. Attackers can utilize such certificates to conduct the Man-in-the-Middle (MitM) attacks. However, many client applications do not properly check the revocation status of certificates. CRLite [86] proposes a client-server model to check certificate revocation status. The server aggregates the revocation information for all-known revoked certificates and generate a filter based on them. The clients download the filter and use it to check the revocation status. CRLite relies on Bloom filters to achieve low memory cost for running on mobile devices. In this study, we show that replacing Bloom filters with vacuum filters results in smaller memory cost for the same target false positive rate ϵ.

We use the data downloaded from Censys [19], which contain both revoked and non-revoked certificates. In this set of experiments, the total number of certificates is 30 million. The number of unrevoked certificates is 29,725,074, the number of revoked certificates is 274,926. For this application, we build three AWQ data structures to store the revoked and unrevoked certificates respectively. We set the target false positive rate to be 0.1% and 0.01%. For Bloom filters, we set the number of hash functions to achieve the best false positive rate. Table 2.3 shows the results. We can find that vacuum filters achieve the lowest memory cost with a target false positive rate compared to CFs and BFs in all cases. We also evaluate the lookup throughput with only existing items and the mixed items (containing 50% alien items). We can find that VFs and CFs have similar lookup throughput. BFs are the worst in both existing and mixed lookup throughput.

2.6 Summary

Vacuum filter, which is a type of AMQ data structures, is a more memory-efficient and faster replacement of Bloom and cuckoo filters. We made three main contributions: 1) a fingerprint eviction strategy to achieve both high load factors and better data locality; 2) a new insertion algorithm that combines the advantages of both BFS and DFS to achieve a higher load factor and better insertion throughput; 3) an instant updates and periodical reconstructions (IUPR) method to resolve duplicate insertions and set resizing, both of which are considered difficult to handle in previous AMQ data structures. Experimental results and real case study show that vacuum filters require smaller memory in all cases and provide higher throughput in many situations, compared to existing methods.

Table 2.3: Performance of AMQs for certificate checking applications

	bits per item			**measured false positive r.**			**lookup thpt (pos)**			**lookup thpt (mixed)**		
	VF-ss	CF-ss	BF	VF-ss	CF-ss	BF	VF-ss	CF-ss	BF	VF-ss	CF-ss	BF
valid	**12.752**	13.546	13.542	0.081%	**0.076%**	0.117%	**7.593**	7.455	2.177	**7.383**	6.250	4.184
	15.808	16.932	18.750	0.011%	0.0098%	**0.0090%**	**7.127**	7.001	1.473	**7.042**	6.045	2.900
revoked	**12.873**	20.978	13.542	**0.087%**	0.097%	0.112%	14.704	**14.868**	4.149	**13.754**	11.052	8.198
	16.091	26.699	17.709	**0.011%**	0.012%	0.014%	14.166	**14.650**	3.232	**13.797**	10.775	6.580

Chapter 3

CCV: Collaborative Validation of Public-key Certificates for IoT by Distributed Caching

In this chapter, we introduce our designed CCV protocol to provide a fast certificate validation, enhancing the perception layer services for IoT devices.

In recent years, Internet of Things (IoT) has attracted significant attention due to the emergence applications of industrial automation, smart devices, vehicular communication, smart cities, and smart homes [31] [152] [112]. A widely accepted definition of IoT for smart environment is that IoT is an interconnection of sensing and actuating devices that is capable of sharing information across platforms through a unified framework such as cloud [68]. Thus, a great amount of data, including both public and private information, will be generated, processed and transmitted by IoT devices.

Data authenticity for IoT devices is a critical issue. Many current IoT devices rely on a central platform to verify data authenticity [133] [124]. However, emerging and future IoT devices, such as personal health monitors, unmanned aerial vehicles, robots, and self-driving cars, become multi-functional, self-organized, and interactive. Hence due to scalability and autonomy problems, there may not be a central platform to interconnect all these devices. IoT devices may directly communicate with each other to share data. Public key cryptography (PKC) enables fundamental security protocol-s for IoT data communication, based on a well-functioning public key infrastructure (PKI). In fact, PKC can be widely used in IoT environment. We list the following (incomplete) important use cases of PKC for IoT. **1)** The authentication process in protocols for establishing a secure channel between two end devices or one device and a server. For example, in an IoT-based healthcare system, wearable sensors that collect human-related data need to securely communicate with other sensors, caregivers and doctors [106]. Existing approaches modify traditional end-to-end IP security protocols to adapt to IoT environments, such as DTLS [122] and HIP DEX [108], which rely on PKC for handshaking. **2)** When an IoT device retrieves sensing data that were collected by other sensors and stored in the cloud, it needs to verify the data integrity and au-thenticity to guarantee that data have not been tampered with or partially dropped [90]. Digital signatures of sensing data are applied for this situation. In order to verify the correctness of a data signed by the private key of the data generator, a device first needs to validate the public key via its certificate. **3)** Recently studied Blockchain-based IoT systems [50] heavily rely on PKC. **For all situations, certificate validation is an**

essential step. Although certificate validation can be completed relatively easily on an ordinary computer, it incurs non-trivial overhead on resource-constraint IoT devices. For example, using an optimized method that requires only one signature verification, certificate validation still costs 1.9 seconds and certificate-based public key operations demand 95% of the overall processing time of handshaking on the WisMote platform [92], as reported in [72].

So we focus on a specific yet important problem: how to perform **fast certificate validation** in a large IoT network. We do **not** intend to improve handshaking protocols, PKI, or PKC schemes in IoT. Instead we study the certificate validation method that is compatible to most existing PKIs and PKC algorithms. There has been existing work on reducing certificate validation cost in classic network environments. One method is to delegate certificate validation to a third party [72] [106] [105], which causes high overhead to the third party, making it be the computation bottleneck and a single point of failure. Some recent work aims at reducing the cost of checking certificate revocation status [97] [143], but provides no solution to reducing the signature verification in certificate validation. Prefetching and prevalidation are also used for efficient certificate validation [73] [134], but they bring heavy storage cost. However, IoT devices are often deployed with limited memory.

Fast certificate validation on IoT devices seems to be a dilemma: the most effective approach is to cache as many validated certificates as possible, but it is not allowed on IoT devices with limited memory. We call the process of validating a public-key certificate via verification of CA signature as *individual validation.* Individual validation of every certificate is time-consuming. Hence none of the above methods is desired for

IoT.

In this thesis, we propose to utilize the power of distributed caching and explore the feasibility of using the cache spaces on all IoT devices as a large pool to store validated certificates, which can be accessed by any internal device. We design a **C**ollaborative **C**ertificate **V**alidation protocol (CCV), which adopts the cooperation strategy in a large IoT network and utilizes the overall computation power and storage resources. When one device d needs to validate a certificate that has been validated and cached by another device h in the network, d can request a collaborative certificate validation from h to confirm that the requested certificate matches the cached one. The design of CCV includes **three main challenges**. First, how each device can efficiently locate the holder of a certificate without storing a long index that maps every certificate to its holder. Second, how to avoid false validation results shared by the IoT devices controlled by the attacker (called malicious devices). Third, how to dynamically maintain a consistent collaborative validation when new certificates are validated and cached certificates are removed or revoked.

Our contributions of this work include the following. 1) We design a memory-efficient and fast locator for certificate holders, called OLoc, based on a recent data structure Othello Hashing [148]. 2) We introduce a trust model for CCV to evaluate the trustworthiness of each device to avoid dishonest collaborative validations from malicious devices. 3) We design a complete protocol suite for efficient OLoc update, cache replacement, and revocation status checking mechanisms in a dynamic network. Evaluation results show that CCV only uses less than 25% time compared to the certificate validation in a recent method [72]. The majority time cost of CCV is on network latency rather than local public key decryptions (reducing $> 90\%$ decryptions), significantly saving computation resource.

3.1 Problem Statement and Models

3.1.1 Problem Specification and Network Model

Fig. 3.1 shows the IoT system of this work. We consider a large IoT system including a large number (100 or more) of devices. The use cases of such system can be an industrial IoT control network [80], a community network with home IoT devices, or an organization/building/campus network with various IoT devices (cameras, sensors, smart office products, etc.). The system consists of the following units.

(1) IoT devices. An IoT device (or "device" in short) is a sensor or actuator with constrained computing, memory, and power resources. Each device can communicate to the Internet through the routers in the IoT system. A device sends and receives packets to/from a router using its wireless chip via either direct connection to a router ("infrastructure mode", such as those in a home WiFi network) or multi-hop forwarding ("ad-hoc mode", such as those in a low-power sensor network). Devices can communicate with each other.

(2) Routers. Routers are the forwarding units to support communication among IoT devices, or between a device and the Internet.

(3) Tracker. A tracker is a function running on a remote or edge server to help managing the IoT network. All IoT devices can communicate with the tracker. The tracker does **not** actually perform validation or caching for devices. **A tracker**

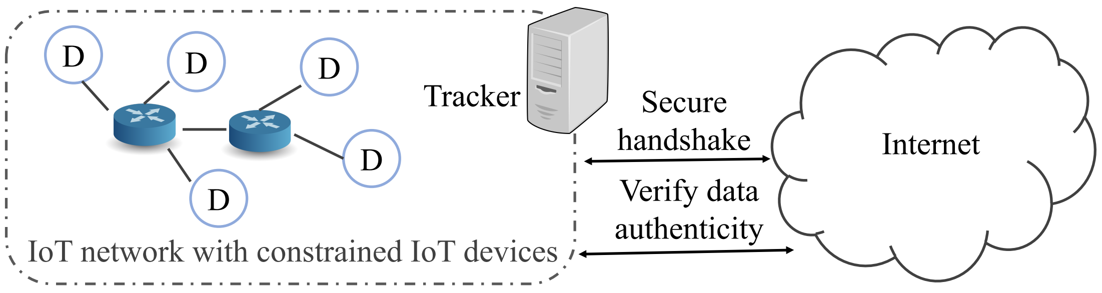


Figure 3.1: Network Model

in CCV is not a single point of failure. Our experiments will show that a tracker requires minimal computation, storage, and communication cost. Hence it can be easily replicated. In CCV, we adopt two trackers: the primary tracker and the secondary tracker. The primary tracker periodically sends the latest stored data required in CCV to the secondary tracker for backup. When the primary tracker stops functioning, CCV will use the secondary tracker. Even if the two trackers both stop working for a duration of time, IoT devices can still perform certificate validation – though not optimal.

This work focuses on the public key certificate validation problem of an IoT system. Each certificate is *uniquely identified* by its public key. We assume secure communication channels have already been established among the IoT devices and the tracker in the same local network using standard IoT security solutions such as that in WirelessHART [80]. Hence a device does *not* need to validate public keys of other devices and the tracker in the same network. A device needs to validate a public key certificate of an external node from the Internet in the following situations:

(1) Authenticate an external node during the handshaking to establish a secure session, such as that in DTLS [122] [72].

(2) Verify the authenticity of the data retrieved from a cloud, which carry the digital signatures of external nodes [90].

The *collaborative certificate validation scheme* investigated can be modeled as follows. When a device receives a public key certificate that has already been verified and cached by another device in the network (called the holder), the device needs to locate the cached certificate and ask the holder to confirm it. Otherwise it needs to run individual validation. Each device caches a (limited) number of certificates validated by itself. When a device receives a request from another device in the network to validate a certificate, it will response based on the result from its cache.

3.1.2 Security Model

We assume the internal communication among IoT devices or between a device and the tracker is secure. The secure communication channels have been established using standard solutions such as the security protocol of WirelessHART [80]. Group keys and session keys have been successfully distributed. We do not consider attacks on the communications between two IoT devices or a device and the tracker. All devices, except those controlled by an attacker, are willing to collaborate. The goal of a device is to maximize the functioning of the entire network rather than maximizing the functioning or lifetime of itself. We assume the tracker is trusted.

This work focuses on the research problem of efficient validation of public key certificates, assuming there exists a well-functioning PKI. This research is not about building a better PKI. Hence we do not consider attacks during the PKI validation process.

An attacker can control a number of devices in the network to conduct malicious behaviors, which are referred as *malicious devices*. Malicious devices are "malicious-but-cautious" and may collude. The tracker stores a *trust value* for every device to indicate the likelihood that the device is legitimate. Malicious devices may conduct the following attacks.

(1) **False validation attack:** A malicious device provides false certificate validation results to other devices.

(2) **Self-promoting attack:** A malicious device promotes its trust value by claiming that it helped other devices validate certificates. However, it did not.

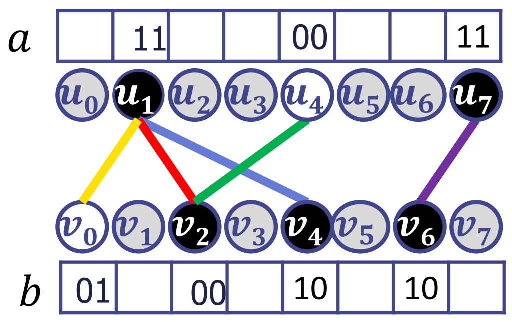
(3) **Defamation attack:** A malicious device claims that a legitimate device provides wrong certificate validation results.

(4) **Traitor attack:** When a diplomatic attacker senses their reputation is dropping because of providing malicious devices, it can provide good services for a period of time to gain a high reputation. Then it provides malicious services after it gains high reputation.

(5) **Whitewashing attack:** Attackers can discard their current identities and re-enter the systems when they have very low trust levels and cannot be selected as collaborators.

(6) **Collusion attack:** Two or more malicious devices improve their trust values by claiming that they helped each other, However, they provide false validation results when helping other devices to validate certificates.

(7) **Sybil attack:** A malicious party creates many fake identities to enter the system. The malicious party can destroy the CCV protocol because fake identities cannot cache certificates. Or the malicious party can conduct further attacks such as


a
11
00
11
u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7
b
01
00
10
10

k	*value*	$h_a(k)$	$h_b(k)$
k_0	01	7	6
k_1	10	1	0
k_2	11	1	2
k_3	01	1	4
k_4	00	4	2

Figure 3.2: Example of Othello Hashing

the false validation attack based on fake identities.

In addition, a device never individually validates a certificate that is not required by its own need, in order to avoid DoS or resource exhaustion attacks. It only caches a certificate validated by itself in prior communications and provides validation confirmation of this certificate to another device.

3.2 Design Consideration of Certificate Locator

One major challenge of collaborative certificate validation is to allow each device efficiently locate another device that has validated and cached the certificate of the public key to use. As the tracker knows the cached certificates of each device, the problem is how to design a structure to store all certificate-to-device information for each device. A simple solution is to let each device maintain a complete index of all certificate-to-device mappings. This method is not scalable because every mapping requires more than 1000 bits of memory, assuming a public key is 1024-bit long. A more advanced method is that each device maintains $m-1$ counting Bloom filters [60] (m is the number of devices in the network). Each Bloom filter represents the set of certificates of a device. The drawback of this method is that locating the holder of a certificate requires up to $m-1$ Bloom filter lookup operations, which is time-consuming especially on IoT devices. Hence they are both impractical for IoT.

In CCV, we utilize and improve a recent innovation called Othello Hashing [148] to design a memory-efficient and fast locator for certificate holders. In addition, existing design of Othello Hashing does not fully satisfy the requirement of the locator hence we propose an improvement design called Othello-based Locator (OLoc). Every device stores an OLoc.

Othello Hashing is used to represent a set of key-value pairs. Given a set of keys K and each key k is mapped to a value $v \in V$. Let $n = |K|$. An Othello Hashing structure is a seven-tuple $< m_a, m_b, h_a, h_b, \mathbf{a}, \mathbf{b}, G >$ defined as follows.

- Integers m_a and m_b is the size of Othello. $m_a \approx 1.33n$, $m_b \approx n$.
- A pair of uniform random hash functions $< h_a, h_b >$ maps keys to integer values $0, 1, ..., m_a - 1$ and $0, 1, ..., m_b - 1$ respectively.
- $\mathbf{a}$ and $\mathbf{b}$ are two arrays including m_a and m_b elements respectively.
- G is a bipartite graph which is used to determine the values in $\mathbf{a}$ and $\mathbf{b}$.

Fig. 3.2 shows an example of Othello Hashing of 5 keys k_0 to k_4. Each key has a corresponding 2-bit value. The bipartite graph G consists of two groups of vertices us and vs. We use a pair of uniform hash functions h_a and h_b. For each key k, we will place an edge in G based on $h_a(k)$ and $h_b(k)$. For example, $h_a(k_0) = 7$ and $h_b(k_0) = 6$. Then an edge is placed to connect u_7 and v_6. The constructed bipartite graph G needs to be acyclic. If there is a cycle, Othello needs to choose another pair of uniform hash functions to re-build G. The possibility to re-build G is small ($\Theta(1/n)$) and it has proved that the cost to build Othello for n keys is $O(n)$ [148]. Once such bipartite graph G is built, the elements of $\mathbf{a}$ and $\mathbf{b}$ can be trivially assigned proper values such that $\mathbf{a}(h_a(k)) \oplus \mathbf{b}(h_b(k))$ is the value of k.

After the arrays $\mathbf{a}$ and $\mathbf{b}$ being built, finding the value of a given key k is extremely fast. Othello can simply retrieve $\mathbf{a}(h_a(k))$ and $\mathbf{b}(h_b(k))$ and compute their XOR, requiring only two memory access operations.

Complexity. The space cost of the two arrays in Othello is small, around $2.33nl$ bits, where l is the length of each value. Each lookup only requires two memory access and one XOR operations –very small constant. It has also proved that the expected time to add, delete and update a key-value pair is $O(1)$ [148].

Opportunities and challenges of using Othello. We find that Othello Hashing is a good fit for the application of memory-efficient certificate locator. Let each key be a public key and the corresponding value be the holder of the certificate. We realize that, to perform locator lookups, only the two hash functions $< h_a, h_b >$ and arrays $\mathbf{a}$ and $\mathbf{b}$ need to be stored in a device. The construction information, such as the key-value pairs and the bipartite graph G are shared by the entire network and not needed for lookups. Hence these information can be stored at the tracker. Note the Othello construction and update operations are relatively more complex than lookups. Hence the IoT devices can avoid these operations and only be responsible for efficient lookups. The tracker has plenty of resources for construction and is responsible for updating and sending the updated Othello arrays to the devices.

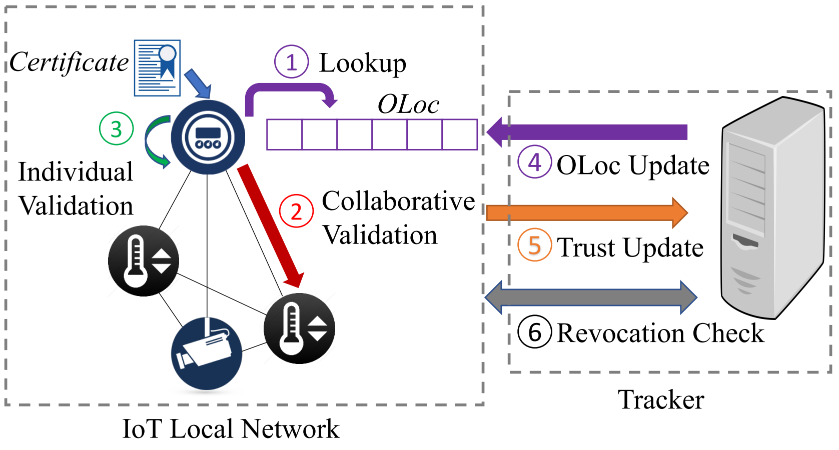


Figure 3.3: Protocol overview of CCV

However one limitation of Othello Hashing is that, if we search a key k' that is not in K, Othello will return an arbitrary value. It is because $\mathbf{a}(h_a(k'))$ and $\mathbf{b}(h_b(k'))$ will be two arbitrary elements. Hence if a certificate C is not cached by any device in the network, the locator will point to an arbitrary device. Falsely locating a holder will waste both communication bandwidth and latency. In the next section we will present an improved design of an *Othello-based Locator* (OLoc) to reduce the rate of false holder locating.

3.3 Protocol Design

3.3.1 Protocol Overview

Fig. 3.3 illustrates the overview of the proposed protocol CCV. The whole system contains many IoT devices and a working tracker. The CCV protocol runs on both the IoT devices and the tracker.

Protocol on an IoT device d. When a device d needs to validate a certificate

C, CCV works as follows.

Part 1. d searches the Othello-based Locator (OLoc) to look for a holder of C.

Part 2. If the OLoc indicates that there is a holder h of C, devices d and h conduct collaborative validation based on the cached certificate on h.

Part 3. If the OLoc indicates that there is no holder of C, d runs individual validation.

Part 5. If a holder h confirms the validation of C, d will forward this event to the tracker with a probability to allow the tracker to monitor the trustworthiness of h.

Protocol on the tracker. The tracker is responsible for updating the OLoc of different devices, monitoring the trust values, and removing revoked certificates. The protocol on the tracker operates as follows:

Part 4. Since the certificates cached in the network change gradually, the tracker needs to update the OLoc for each device to keep track of the update-to-date holder information. It then distributes the updated OLoc to each device.

Part 5. When the tracker receives a forwarded validation confirmation showing that h just helped d, it will verify the correctness of this confirmation and update h's trust value accordingly.

Part 6. When the tracker receives new revocation lists from CAs, it notifies the holders to remove these certificates from their caches.

3.3.2 Bootstrap phase and devices management

Each IoT device or the tracker is associate with a digital certificate that is issued by a certificate authority (CA). Two steps are required in the bootstrap phase. 1) Registration. Each device is registered on the tracker using its certificate. The tracker then audits the identity of the device to filter fake identities. One way is to put physical device information in its certificate [140]. Or CA charges money for the certificate to prevent attackers from getting enough identities for the Sybil attack [40]. Legitimate devices will be added into the device list. 2) Keys generation. Communication channels are built and session keys are established between each two IoT devices and between the tracker and a device. All two entities can create a session key to build a secure channel through the certificate-based key agreement protocol or other efficient key-generation protocols, such as [117].

The system is dynamic with IoT devices join and leave at any time. If a device joins the system, the device follows the two steps in the bootstrap phase. If a device leaves the system, the tracker will delete it by deleting the device in the device list and not choosing it as collaborators in the OLoc. Then the tracker sends messages to inform all the other devices to delete the session key with this device.

3.3.3 Definition of Messages

There are many types of messages exchanged among devices and the tracker in CCV. The CCV protocol is driven by messages and events. We define the following types of messages before we describe details of CCV. Each message is defined by a tuple where the first element is the message type. All messages are exchanged above the secure communication channels inside the network.

- $< \mathit{REQUEST\_VALI}, C, u, v >$: Device u sends this message to request device v to validate the certificate C.
- $< \mathit{REPLY\_VALI}, C, u, v, r >$: Device u sends this message to reply device v the certificate validation result r.
- $< \mathit{REPLY\_NO\_CERT}, C, u, v >$: Device u sends this message to reply device v that it does not cache the certificate C.
- $< \mathit{UPDATE\_TRUST}, u, t, E >$: Device u sends this message to tell the tracker t to update the trust value according to the collaborative validation events E. E includes the information of the prior $\mathit{REPLY\_VALI}$ messages.
- $< \mathit{FALSE\_VALI\_RESULTS}, u, t, F >$: The tracker t sends the message to tell the device u that the prior validation confirmation F is false.
- $< \mathit{NEW\_CACHE}, C, u, t >$: Device u sends this message to tell the tracker that it caches the certificate C.
- $< \mathit{DELETE}, C, u, t >$: Device u sends this message to tell the tracker that it no longer caches the certificate C.
- $< \mathit{UPDATE\_OLOC}, O, u, t >$: The tracker sends this message to device u to request u to update its OLoc to O.
- $< \mathit{REVO\_CERT}, C, u, t >$: The tracker sends this message to device u to remove revoked certificate C.

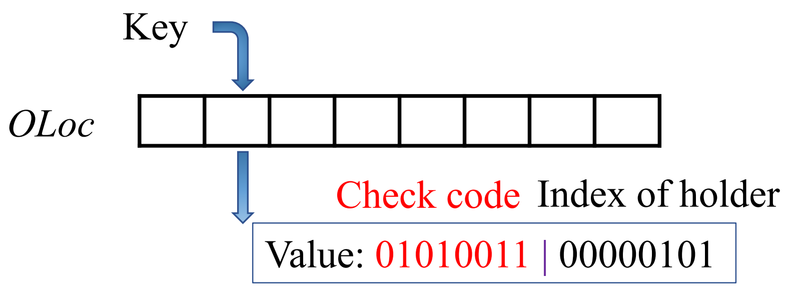


Figure 3.4: OLoc lookup in CCV

- $< CHECK\_REVO, C, u, t >$: Device u sends this message to the tracker to check whether the certificate C is revoked or not.
- $< REPLY\_REVO, C, u, t, r >$: The tracker replies device d the result r of the revocation status of certificate C.

3.3.4 Othello-based Locator and Update

This subsection describes Parts 1 and 4 presented in Sec. 3.3.1 and Fig. 3.3.

Upon receiving a validation requirement of a certificate C of a public key k^+ from the upper layer, a device d first checks its local cache to see whether it has cached C. If C is cached and the two versions are identical, then d confirms the validity of C. Otherwise, d needs to determine whether there is another device being the holder of C and which device it is.

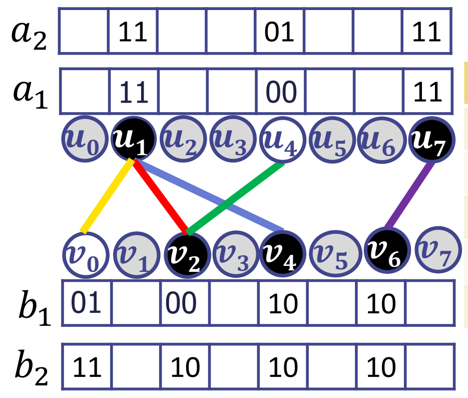
Every device stores an Othello-based Locator (OLoc). The lookup key of OLoc is a public key (identifier of a certificate) and the lookup result should indicate the holder of the certificate. As discussed in Sec. 3.2, one limitation of Othello Hashing is that, if a certificate C is not cached by any device in the network, the locator will point to an arbitrary device. Assume the network has n devices and each device can be referred by a l-bit index: $l = \lceil \log_2 n \rceil$. Our innovation is to extend the lookup value τ of an Othello to $l + l'$ bits for the certificate C of the public key k^+. The l least significant bits (LSBs) of τ is the index of the holder i and the l' most significant bits (MSBs) is the check code c of this certificate. c is determined by the hash value $H(k^+)$ using a CRC hash function H. Since $H(k^+)$ is longer than l' bits,c can simply be the l' LSBs of $H(k^+)$.

Fig. 3.4 illustrates an example of the OLoc lookup. When a device searches its OLoc for the holder of the certificate C of k^+, it compares whether the l' LSBs of $H(k^+)$ matches the check code c return by OLoc. If they match, it is highly likely that the certificate is actually cached by the holder. By "highly likely", we mean that there is still a probability that C is not cached but matches the check code, called a *false matching*. Such probability is around $1/2^{l'}$ depending on the length of the check code l'. Longer check code causes lower false matching rate. The existence of false matchings does not hurt the correctness and security of CCV, but will slightly increase the communication cost. In the example of Fig. 3.4, both l and l' are set to 8, which can be adjusted based on the system requirements.

When the device d gets index i and the check code matches $H(k^+)$, d sends message $< REQUEST\_VALI, C, d, h >$ to another device h whose index is i to perform collaborative validation. If the check code does not match $H(k^+)$, the device terminates CCV and conducts individual validation.

On the tracker side, the OLoc at every device should be dynamically updated to reflect the update-to-date certificate to holder mapping. At the very beginning, Othello is empty. Then the tracker updates the OLoc of all devices at a fixed interval and distributes newly updated OLoc to the devices. Although the tracker is responsible for updating all devices in the network, *these updates are efficient and scalable because all devices may share a same OLoc.* When a device caches a certificate, it sends a *NEW_CACHE* message to notify the tracker. Hence the tracker keeps track of all cached certificates in the network and updates the OLoc. The updated OLoc is then sent to the devices using the *UPDATE_OLOC* message.

We apply another optimization based on the IoT network features. It is possible that one certificate C is cached by multiple devices in the network. Hence C may have multiple holders, any of which can be a valid result of a holder locator. In the construction stage of Othello, we choose the index of one holder to be the lookup result of OLoc for the public key k^+ in C. However, it is reasonable to choose the most suitable holder of C for different devices when there are multiple feasible options. To construct the OLoc of a device d, CCV may choose the holder with the shortest network distance (e.g., smallest hop count) to d. One may note that using this optimization, the OLoc on different devices may be different. Two devices on different locations in the network may be close to different holders. However, constructing these different versions of OLoc is still efficient. They share the same set of keys and the same bipartite graph G, because G depends only on the set of keys, rather than their values. Note that computing G is the most time-complex step during the construction of an Othello. Once G is obtained, determining the arrays $\mathbf{a}$ and $\mathbf{b}$ is trivial. Hence all devices can still share a same G and the arrays $\mathbf{a}$ and $\mathbf{b}$ can be computed in a short time. An example of two different OLoc sharing a same G in shown in Fig. 3.5. In addition, many devices in network proximity are still able to use a same OLoc.


a_2
11
01
11
a_1
11
00
11
u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7
b_1
01
00
10
10
b_2
11
10
10
10

k	$value_1$	$value_2$	$h_a(k)$	$h_b(k)$
k_0	01	01	7	6
k_1	10	00	1	0
k_2	11	01	1	2
k_3	01	01	1	4
k_4	00	11	4	2

Figure 3.5: Multiple OLoc Construction

In addition, the trust values of IoT devices maintained by the tracker are used to filter malicious devices. Hence the tracker will only select the holders whose values are above a pre-determined threshold θ ($\theta \geqslant 0.5$). The tracker updates trust values periodically.

3.3.5 Collaborative Validation

To request a collaborative validation of certificate C, device d sends a message $< REQUEST\_VALI, C, d, h >$ to the holder h. Upon receiving this message, the holder h searches its cache to find the certificate of the public key k^+ on C. If such certificate exists and is identical to C, it replies d with a message $< REPLY\_VALI, C, h, d, `Correct' >$. If the certificate exists but is different, it replies $d$ with a message $< REPLY\_VALI, C, h, d, `Wrong' >$. If no certificate of k^+ is cached, it replies d with a message $< REPLY\_NO\_CERT, C, h, d >$. The main reason of a missing certificate is that previously cached certificates may be replaced by others due to the lack of cache space, while this information has not been updated to OLoc. The other reason is the false matchings. Note all these messages should be signed by h's private key for non-repudiation purposes. To avoid malicious

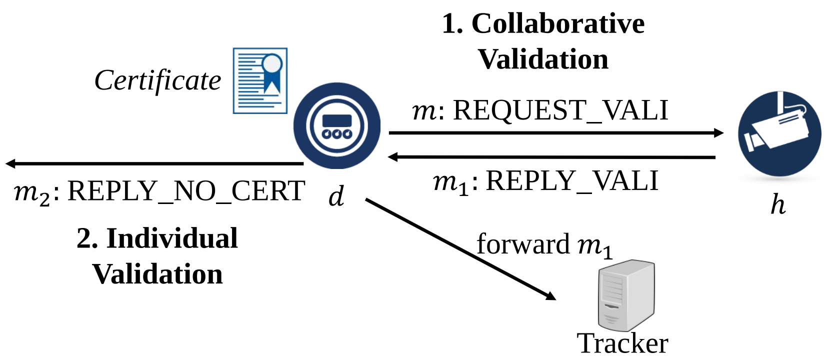


Figure 3.6: Collaborative Validation Process

devices send lots of $REQUEST\_VALI$ messages to exhaust the resource, the holder can record the number of request messages from other devices during a time interval. Once the holder detects an unusually larger number of request messages from a device, it reports this abnormal behavior to the tracker. The tracker can then remove this malicious device from the system.

Once the device d receives the $REPLY\_VALI$ message with a 'Correct' value from a holder h, it knows the validation of certificate C is confirmed and it can use C for incoming communications. If d receives the $REPLY\_VALI$ message with a 'Wrong' value. It will discard the certificate C and still forward this event to the tracker. If d receives the $REPLY\_NO\_CERT$ message, it runs individual validation. Fig. 3.6 shows the collaborative validation process.

One additional step is that d may forward the confirmation events to the tracker, in order to improve the trust value of the holders that provide validation. It sends a message $< UPDATE\_TRUST, d, t, E >$, where E includes one or more $REPLY\_VALI$ messages received during the past period of time as well as their digital signatures. In order to save message cost and reduce tracker overhead, d does *not* forward every collaborative validation event but on a sampling basis. The intuition of using sampling is that when a malicious device keeps providing false validation results, then statistically it will be detected. The sampling rate can be dynamically adjusted. For example, the sampling rate can be higher at the beginning to quickly filter malicious devices. When the system is stable, the sampling rate is set to be lower to reduce communication cost and the computation overhead of the tracker.

3.3.6 Individual validation and caching

This subsection describes Part 3 of CCV.

If the device d chooses to run individual validation of certificate C, this process consumes computation resource and relatively long latency on d. After C being validated, d will cache C in its local memory. One of the following three cache replacement strategies will be used to replace old certificate: random, FIFO (first in, first out), and LRU (Least recently used). If an old certificate C_o is replaced by a new one, d sends a message $< DELETE, C_o, u, t >$ to the tracker. The updates of cached certificates will be sent to the tracker immediately to make the tracker knows the up-to-date certificate-to-device information.

Besides validating the certificate, d also needs to check the revocation status of the certificate. It sends $< CHECK\_REVO, C, d, t >$ to the tracker to query the revocation status. If the certificate is included in the revocation list stored in the tracker. The tracker will send a $< REVO\_CERT, C, d, t >$ message to call back C and let d stop using C and remove it from the cache.

3.3.7 Trust Model and Updates

As discussed in the security model, the malicious devices may conduct seven types of attacks to the whole system. In order to facilitate the detection of malicious devices and make them have lower probability to be the selected holder, it is necessary and essential for CCV to introduce a trust model in the protocol. In this section, we first introduce the trust model and then describe the trust update protocol of CCV (Part 5).

Trust Model. In CCV, we adopt the following definition of trust from [45].

Definition 1. In the IoT network, a device d's trust to another device d' is the subjective expectation of d of receiving positive outcomes through the communications with d'.

Specifically, the trust value in CCV quantifies the expectation to receive correct validation results of certificates. The range of trust value between two devices is $[0, 1]$. At the beginning, the trust value between each two devices is set to be 0.5.

In general, trust can be categorized into two classes: direct trust and indirect trust. **Direct trust** is the trust that is calculated by direct communications between two devices. **Indirect trust** is the trust that is calculated by indirect recommendations, which will be explained later.

We consider direct and indirect trust and use past collaboration behaviors between every two devices to measure the trust value. Direct trust is based on the certificate validation results between a validation requester and the holder. The tracker maintains two arrays to record the number of honest validation events s and the number of dishonest validation events f between a requester d and a holder d' during the trust update interval, respectively. Note that every device forwards collaborative validation events to the tracker at a sampling rate. Then the tracker will audit whether the holder honestly validated the certificates or not. Once a validation result is audited, the tracker will update the corresponding number in the arrays. With the information of s and f for any two cooperative devices, we adopt a subjective logic framework [75] [76] in the binary domain to compute the corresponding direct trust value.

Let $\mathbb{X} = \{x, \bar{x}\}$ be a binary domain with binomial random variable $X \in \mathbb{X}$. In our trust model, x represents that device d trusts the device d'. In the subjective logic framework, a binomial opinion about a state value x is the ordered quadruplet $w_x = (b, d, u, a)$, where $b + d + u = 1$ and $b, d, u \in [0, 1]$. b represents the belief mass in support of x being true, d represents the disbelief mass in support of x being false, and u is the uncertainty mass representing the vacuity of evidence. a is the base rate which represents the prior probability of x without any evidence. a is set to be 0.5. The probability of a binomial opinion about value x is

$$p = b + au \tag{3.1}$$

It can be shown that posteriori probabilities of binary events can be represented by the beta distribution $Beta(\alpha, \beta)$ [75]. α and β parameters can be expressed as a function of the observations (s, f) and the base rate a.

$$\alpha = s + 2a, \qquad where \quad 0 < a < 1, s \geq 0$$

$$\beta = f + 2(1 - a), \qquad where \quad 0 < a < 1, f \geq 0$$

Then the beta distribution can be expressed by

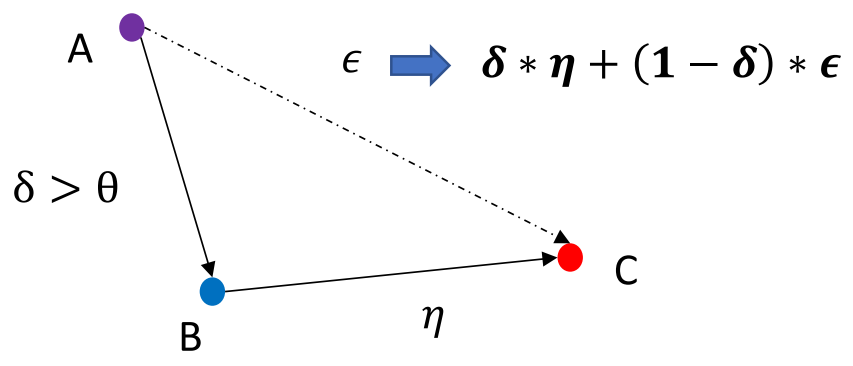


Figure 3.7: Calculation of Indirect Trust

$$f(p \mid \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} p^{\alpha-1}(1-p)^{\beta-1}$$

The expectation probability of value x is

$$E(x) = \frac{\alpha}{\alpha+\beta} = \frac{s}{s+f+2} + \frac{2}{s+f+2}a \tag{3.2}$$

Combining formula (3.1) and (3.2), b, d, u are calculated by the following formula.

$$b = \frac{s}{s+f+2}, d = \frac{f}{s+f+2}, u = \frac{2}{s+f+2}$$

One consideration is that more punishment need to be given when there exist dishonest validation events. Thus the trust value of malicious device can be sharply decreased when the tracker detects its dishonest validations. We add γ parameter to adjust the influence of dishonest validation on the trust value. Therefore, b, d, u is are finally calculate by the following formula.

$$b = \frac{s}{s+\gamma f+2}, e = \frac{\gamma f}{s+\gamma f+2}, u = \frac{2}{s+\gamma f+2}$$

Direct trust value is $b + au = \frac{s+1}{s+\gamma f+2}$. Larger γ causes lower trust value with dishonest validation results.

Indirect trust is based on the recommendation. A device can aggregate trust recommendations from its trusted devices. The trust value between two devices is used as the measurement for choosing recommenders. Then threshold θ is introduced to select recommenders. Fig. 3.7 shows how to calculate indirect trust. Device A trusts B with a direct trust value δ and device A trusts C with a direct trust value ϵ. When there is an update between device B and C, and the new trust value is η. If the trust value from A to B exceeds the threshold θ, B can recommend C to A with trust value η. The updated value is $\delta\eta + (1 - \delta)\epsilon$. Hence, A trusts C with the updated trust value with this recommendation scheme. We set θ to 0.5 in our experiments.

Trust Updates. When the tracker receives an *UPDATE_TRUST* message, it verifies the collaborative validation events in E. The tracker maintains two arrays A and B to store the numbers of honest and dishonest collaborative validation events between two devices during the trust update interval respectively. $A[i][j]$ denotes the number of honest validations that j provides to i, and $B[i][j]$ denotes the number of dishonest validations that j provides to i. The tracker also stores the trust value $T[i][j]$ at current time which denotes the degree that i trusts j. The update algorithm is shown as Algorithm 5.

Algorithm 5: Trust Value Update Algorithm

```
parameter: $A$, $B$, $\theta$

for Every event of holder $v$ helping device $u$ do
    Compute direct trust $t_d$;
Update $T[u][v] = pT[u][v] + qt_d$; for Every device $i$ which has $T[i][u] > \theta$

do
    Compute indirect trust $t_i = T[i][u] * T[u][v] + (1 - T[i][u]) * T[i][v]$;
Update $T[i][v] = pT[i][v] + qt_i$;
```

For every computed direct or indirect trust value of device u for another holder device v, the tracker calculates the moving average combining the historical trust value $T[u][v]$ and the newly computed trust value t_{uv}: $T[u][v] = pT[u][v] + qt_{uv}$, where $p, q \in (0, 1)$ and $p + q = 1$. p and q denotes the weight of historical trust values and newly updated trust values respectively. We set p to be 0.4, which gives more weight to the updated trust value for the device.

In our model, assume the number of honest validation events of device u and the holder device v during the update interval is s. The number of dishonest validation events is f. When $f \geqslant 1$, for $\gamma > \frac{s}{f}$, the newly computed trust value $\frac{s+1}{s+\gamma f+2} < \frac{s+1}{2s+2} < 0.5$. Larger γ causes a higher decrease. Therefore, our model can make the trust value dramatically decrease below 0.5 to detect the malicious device by setting a larger γ when the false validation behaviors are detected by the tracker.

If the tracker finds that a validation is dishonest and the certificate is not valid, it sends a *FALSE_VALI_RESULTS* message to tell the device.

3.3.8 Revocation Check

Revocation status check is important in certificate validation but time-consuming on devices. In CCV, the tracker actively downloads the Certificate Revocation List (CR-L) [86]. A device can send a *CHECK_REVO* message to request the revocation status of a certificate. This design makes the device start using the certificate simultaneously while waiting for the reply from the tracker. The tracker replies about the status using a *REPLY_REVO* message. If the certificate is revoked according to the tracker, the device stops using it, removes it from the cache, and rolls back to the prior state. In addition, when the tracker updates its local revocation list and finds existing certificates cached in the network are expired, it will send *REVO_CERT* messages to the holders of these certificates for removing them. If all the trackers stop functioning, the device uses Online Certificate Status protocol (OCSP) [110] to query the CA's OCSP server about the revocation status of the certificate.

3.4 Simulation Results and Security Analysis

We implement a complete version of CCV in a packet-level discrete-event simulator running on a desktop with 3.6GHz Intel(R) Core(TM) i7-7700 CPU. To better simulate the performance of CCV for IoT devices, the actual processes of cryptographic operations are implemented and the latencies, including cryptographic latency and network latency, are simulated. The reason is that the cryptographic latency on a desktop does not reflect the actual cryptographic overhead on an IoT device. Hence in the simulator, we use the latency data of cryptographic operations gathered from a WisMote platform featuring a 16MHz MSP430 micro-controller [92] [72]. We use SHA256 for hash operations, elliptic curve NIST P-256 for PKC, and AES-128 for symmetric-key operations in secure communications among in-network IoT devices. We compare CCV to individual certificate validation [72], in which validating the certificate chain only requires one single decryption operation. The average time of such individual certificate validation on WisMote is around 1.9sec with 13.9ms standard deviation as reported in [72]. Note most existing certificate validation methods typically require multiple intermediate certificates in a chain, and the validation overhead grows linearly with the number of intermediate certificates [72]. For the simulated network latency, every two IoT devices are connected by one or more hops of routers. We define one hop latency in our simulation to be 20 ms [118], and the maximum hop count in the network is set to be 5.

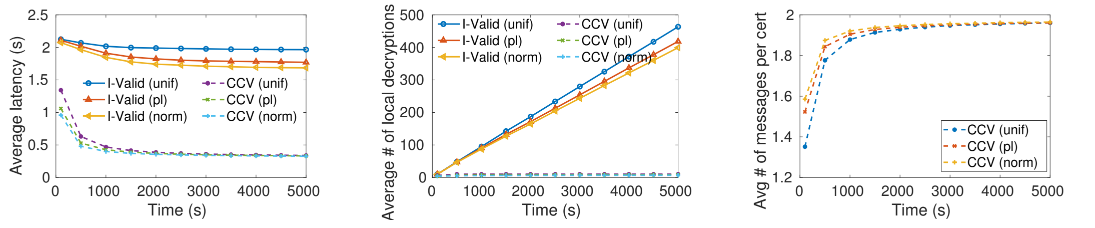
In our experiments, we simulate a number of IoT devices running the CCV protocol to collaboratively validate the certificates. We use ‘I-Valid’ to refer to individual validation. **For fair comparison, in CCV we assume at the system start time, no certificate is validated and cached in the network. OLoc will use the cache space.** Each device in CCV also stores session keys between each other devices and the tracker. Every certificate in CCV must be individually validated once and cached for further use. The tracker updates the OLoc every five seconds.

We evaluate and compare the following six metrics of CCV.

1) **Latency** is the average time from receiving a certificate to finishing validation on a device.

2) **Number of local decryptions** is the number of times of running public key decryption to validate certificates. It characterizes the computation overhead on devices.

3) **Average number of messages per certificate** evaluates the communication cost. The number of messages for the certificate validated through caching by its received device is 0. When the received device individually validate the certificate after not finding the helpers, the number of message is 1, because NEW_CACHE message will be generated and sent to the tracker. When a certificate is successfully validated by the helper, the number of messages is 2. But when the helper has not validated



(a) Validation latency (b) Number of local decryptions (c) Ave. # of messages per cert

Figure 3.8: Performance varies with time

the certificate and then the certificate is validated by its received device, the number of message is 3.

4) **Throughput** is the maximum number of certificate validations on the simulator. Although the computation resource on the simulator is different from that on a device, this metric still reflects whether CCV reduces resource overhead.

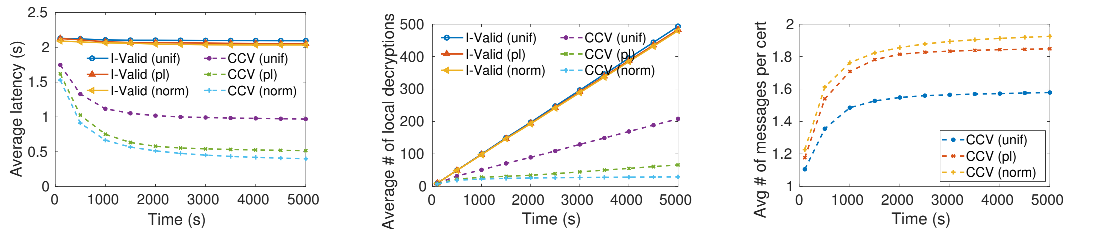
5) **Computational cost for OLoc update** measures the overhead of the tracker.

6) **Trust value changes** are used to detect malicious devices.

The number of events that require certificate validations happening on devices follows the Poisson distribution. The parameter λ denotes the average number of events happening in one second, which is used to adjust the frequency of events. Note the number of events that requires a particular certificate may not follow a uniform distribution. For example, some popular data or Internet server may be accessed by many devices. Hence we vary this distribution in three types: namely uniform, normal, and power law distributions.

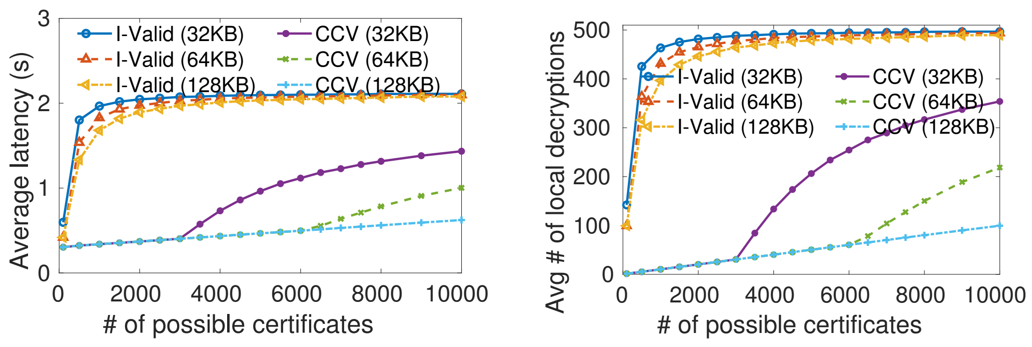
3.4.1 Evaluation Results

Performance varies with time. Assuming at the system start time, no certificate is validated and cached in the network. Then the devices validate and cache certificates gradually. We may expect that the validation latency of CCV will decrease when time increases. Fig. 3.8 shows the performance comparison of CCV and I-Valid, by varying the time. In this set of experiments, the number of IoT devices is 100, the total number of certificates is 1000, λ is set to be 10, and the memory size (OLoc, symmetric keys and cached certificates) is set to be 32 KB. I-Valid also uses cache space to store certificates. We show the results for uniform, normal, and power law distributions. From Fig. 3.8(a), we find that the average latency to validate a certificate for CCV decreases during time 0s to 2000s and then remains stable after 2000s due to collaborative validations. CCV only uses 25% time compared to I-Valid. More importantly, Fig. 3.8(b) shows that CCV always requires around 10 decryptions per device, while this number can be > 400 for I-Valid at time 5000s. CCV reduces more than 99% local decryption operations and significantly saves the computation cost. The latency of CCV is mainly the network latency. From Fig. 3.8(c), we can see that the average number of messages for each certificate in CCV increases during time 0s to 2000s, but will be stable after 2000s. It is because more collaborative validation will be used than individual validation. CCV is very communication-efficient: the number of messages per certificate is close to two. The two messages are *REQUEST_VALI* and *REPLY_VALI* message respectively. We also find that different distributions have relatively small influence on the performance of CCV in this experiment, because 100



(a) Validation latency (b) Number of local decryptions (c) Ave. # of messages per cert

Figure 3.9: Performance varies with different distributions

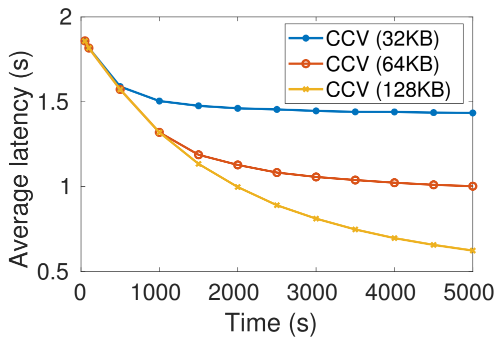


(a) Validation latency (b) Number of local decryptions

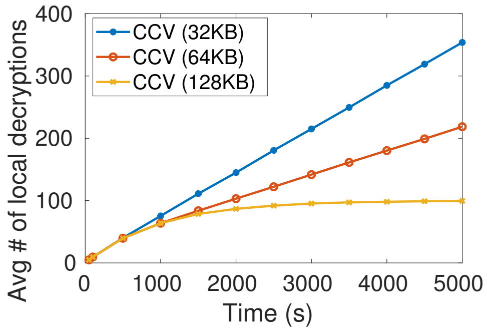
Figure 3.10: Performance varies with the number of possible certificates and cache size

devices with 32kb cache size can cache almost all the 1000 certificates. In fact, power law and normal distributions are more desired for I-Valid protocol because it prefers a small part of certificates which be used for many times, causing high cache hit rates. We then conduct an experiment by changing the number of possible certificates to be 5000, other settings are the same. Fig. 3.9 shows the results. We can find that power law and normal distributions achieve better performance than uniform distribution when it is hard for cache pool to cache all the certificates. In the following experiments, we use uniform distribution, which reflects the real number of possible certificates.

Varying the number of certificates and cache size. We conduct experiments by varying both the number of total possible certificates and the cache size to



(a) Validation latency



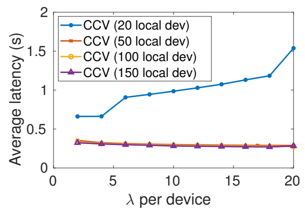
(b) Number of local decryptions

Figure 3.11: Performance varies with time and cache size

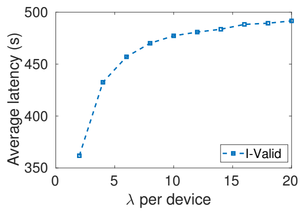
evaluate their influence. In this set of experiments, the number of devices is 100, λ is set to be 10, and the time is a 5000s duration, we set the cache size to be 32kb, 64kb and 128kb respectively. The total number of possible certificates varies from 100 to 10000. The results are shown in Fig. 3.10. We find that CCV outperforms I-Valid protocol with lower latency and lower number of local decryptions. The total number of possible certificates influences the performance of CCV and I-Valid, because the whole system needs to cache more certificates with larger number of possbible certificates, otherwise certificates are validated by individual validation. When the number of total possible certificates increases, the latency (Fig. 3.10(a)) and number of decryptions (Fig. 3.10(b)) both increase in CCV, but still provides advantages compared to I-Valid. There is a sudden increase at 3000s certificates for CCV with 32KB cache size and a sudden increase at 6000s with 64KB cache size in both latency and the number of decryptions. This is because it is hard for cache pool to cache all the certificates, causing more individual validations. However, when the cache is 128KB, the whole cache pool enlarges and there is no such problem. We also conduct another experiments to analyze the influence of cache size. In this set of experiments, the number of devices is 100, λ is set to be 10, the total number of possible certificates is 10000, the cache size is set to be 32KB, 64KB and 128KB respectively. Fig. 3.11 shows the results. We can find that the average latency with 128KB keeps decreasing during time 0s to 5000s. CCV with 128KB cache provides the best performance among them. This is because the cache pool capacity with 32KB and 64KB per device is not big enough.

Varying λ per device and the number of devices. Larger number of validation events happening per second indicates more validation requirements, which brings higher resource pressure for the devices. In this experiment, we evaluate the performance varies with λ per device. From Fig. 3.12(b), we can see the obvious increase of average latency for I-Valid when λ increases. In CCV more IoT devices can release the pressure, because the overall cache pool capacity increases. Fig. 3.12 shows the performance varying with λ per device and the number of devices. In this set of experiments, there are 3000 possible certificates, the cache size is set to be 64KB, and the time is 1000s duration. We can find that the average latency of CCV with 50, 100 and 150 devices is small and stays stable versus λ per device, which shows that CCV has a good performance with concurrent certificates validation requests with the shared distributed cache. There is a performance degradation with 20 devices because the cache pool with 20 devices cannot cache all the 3000 certificates, causing many individual validations for each device.

Cache replacement. This set of experiments vary cache replacement strategies including random, FIFO and LRU. Fig. 3.13 shows that the strategy has little influence on the latency of CCV, with random and LRU being slightly better.



(a) Validation latency



(b) Validation latency of I-Valid

Figure 3.12: Performance varies with λ per device and # of devices

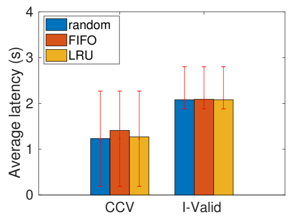


Figure 3.13: Validation latency with cache replacement strategies

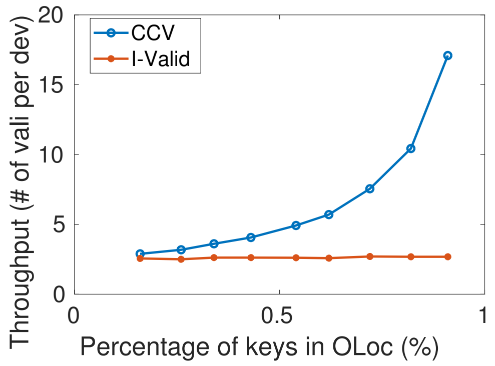


Figure 3.14: Throughput comparison on simulator

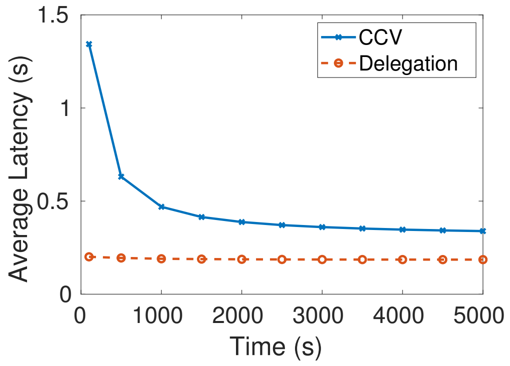


Figure 3.15: Latency comparison on simulator

Throughput. In this set of experiments, we compare the validation capacity of CCV and I-Valid by keeping devices performing validations in the simulation. The simulator simulates 100 devices simultaneously. The signing and verification algorithm is ECDSA with 160 bits keys. Fig. 3.14 shows that the number of validations on a device in one millisecond. From Fig. 3.14, we can see that CCV has a much better throughput compared to I-Valid protocol, especially when large percentage of public keys are recorded in OLoc.

3.4.2 Comparison with delegation-based method

We also compare our CCV with delegation-based method. In delegation-based method, there is a trusted third party to help IoT devices to validate certificates. The third party can be a smart gateway, a delegation server and etc. IoT devices do not individually validate certificates. Instead, they send certificates validation requests to the third party, and then wait for the reply. After receiving the validation results, IoT devices can cache the validated certificates. Certificates can be validated by directly comparing with the cached ones for each IoT device. When the cache is full, cached certificates are replaced by newly cached ones according to three cache replacement strategies: random, FIFO (first in, first out) and LRU (Least recently used). In this set of experiments, the number of IoT devices is 100, the total number of possible certificates is 1000, λ is set to be 10 and the memory size of each device is 32KB. We assume there is a server which serves as the trusted third party. And the server has adequate computation power and memory. Fig. 3.15 shows the results. After 5000s, the average latency of each certificate for CCV and delegation-based method is 0.34s and 0.19s respectively. We can find that delegation-based method has better performance than CCV, because all the certificates are validated by the server with good computation and storage power. However, the server needs to continuously help IoT devices validate the certificates, once the server fails, the whole system cannot validate certificates. CCV can still perform certificate validation when the tracker stops functioning for a duration of time.

Table 3.1: Time for building OLoc

# certificates	100	1000	5000	10000	100000
Time (ms)	0.20	1.48	6.59	13.03	131.51

3.4.3 Tracker overhead

The heaviest task for the tracker is to update the OLoc. Table. 3.1 shows the time to construct an OLoc with different number of certificates. The results show that it is very time-efficient for the tracker to update the OLoc with a gigantic size of certificates. CCV also builds different OLoc to choose the most suitable holder of C for different devices when there are multiple choices. The reasons for multiple copies of C are the update delay of OLoc at the begining and the detection of malicious devices. The number of different values among different OLoc is small. Table. 3.2 shows the time to construct an OLoc based on an existing OLoc with 10000 shared keys but different values. The results show that it is very time-efficient for the tracker to build the OLoc when there is little difference in values with an existing OLoc.

We also evaluate the overhead for updating trust values. Table 3.3 shows the time to update trust values with different number of collaborative events when the number of device is 1000. The results show that it is time-efficient to update trust values.

Storage cost. We evaluate the storage cost for the tracker in CCV. The tracker needs to store the following data. 1) The tracker stores trust values between IoT devices and two arrays to record the number of honest and dishonest validation events between devices. 2) The tracker stores all cached certificate-to-device information to

Table 3.2: Time for building OLoc with same keys but differnet vals

# different vals	10	100	200	300	400
Time (ms)	2.05	2.98	4.33	5.98	7.52

Table 3.3: Time for updating trust values

# events	100	500	1000	5000	10000	100000
Time (ms)	0.87	3.9	7.754	37.422	105.434	1238.2

build OLoc. 3) The tracker stores OLoc. 4) The tracker stores keys which include public keys of devices and symmetric keys between each device. 5) The tracker stores devices information including the IP of each device and the distance information between each two devices. Fig. 3.16 shows the storage cost. We can find that the tracker requires a small memory cost. The tracker needs about 1.24 MB storage with 200 devices and 10^4 possible certificates.

3.4.4 Security Results and Analysis

We first provide the evaluation results and then analyze the influence of parameters in the trust model. We also provide a detailed analysis about how CCV defends against six major attacks conducted by malicious devices mentioned in Sec. 3.1.2.

Trust value evaluation. We monitor the trust values changes for both honest and malicious devices. In the set of experiments, the number of devices is 100, the number of certificates is 1000, time is set to a 2000s duration, and the cache size is 32 KB for each device. The initial trust value between any two devices is set to be 0.5. γ for the trust model is set to be 10. The trust value will be dynamically updated due

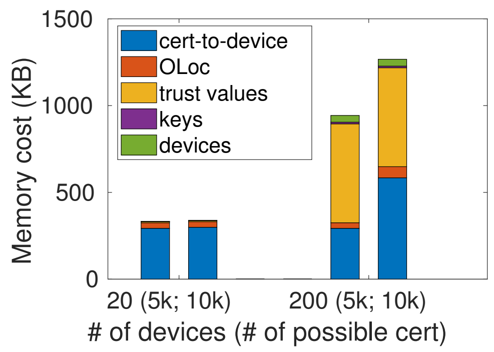


Figure 3.16: Storage cost for the tracker

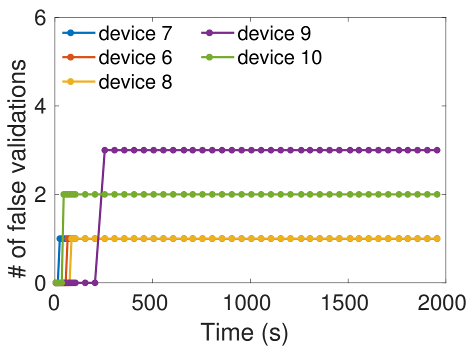
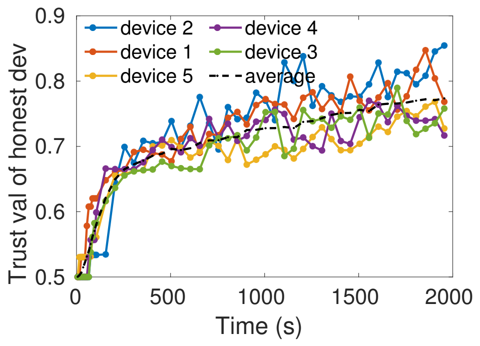
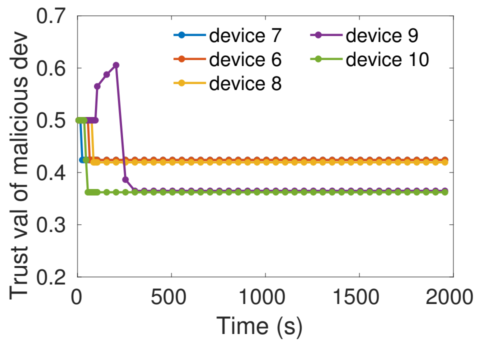


Figure 3.17: Number of false validations



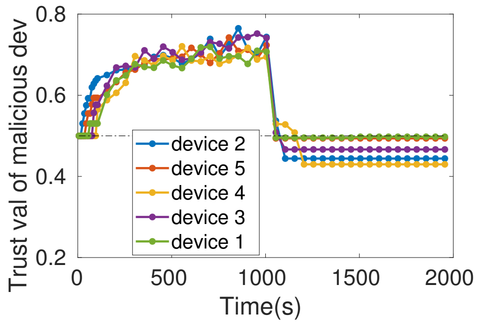
(a) Trust values of honest devices



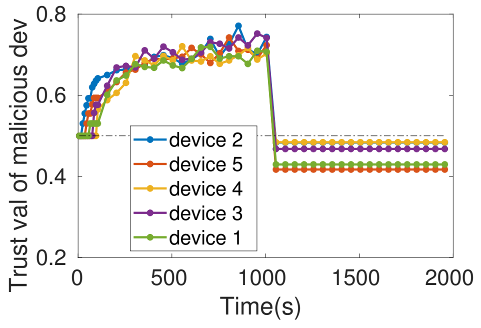
(b) Trust values of malicious devices

Figure 3.18: Trust values

to collaborative validation. The percentage of malicious devices is 5%. For malicious devices, they randomly choose to provide honest or dishonest validation when receiving collaborative validations. Fig. 3.18 shows the trust value changes. We show the results of five randomly chosen honest (Fig. 3.18(a)) and the five malicious devices (Fig. 3.18(b)). The other honest devices show similar results. We also show average trust values changes of all the honest devices, which is denoted by average in Fig. 3.18(a). We find that the trust values of honest devices increase to > 0.5 and are maintained in a high level. On the other hand, the trust values of malicious devices are all < 0.5 once they provide false validations. They will be filtered by the tracker during OLoc construction. We

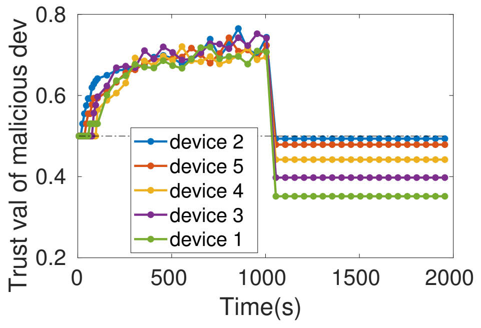


(a) $\gamma = 1$

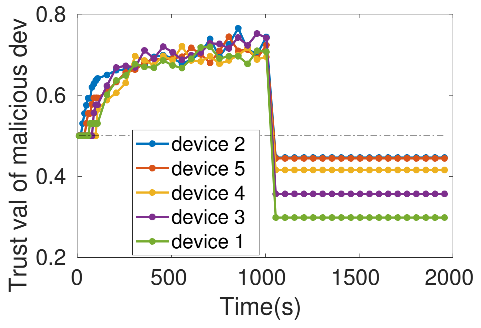


(b) $\gamma = 2$

Figure 3.19: Trust values of malicious devices with $\gamma = 1, 2$



(a) $\gamma = 5$

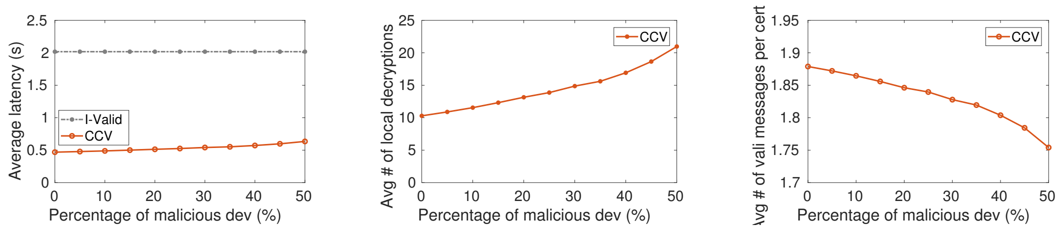


(b) $\gamma = 10$

Figure 3.20: Trust values of malicious devices with $\gamma = 5, 10$

also show the number of total false validations by malicious devices in Fig. 3.17. When malicious devices are selected as helpers, they may conduct false validations. However, their trust values will dramatically decrease below 0.5 once the dishonest behaviors are detected by the tracker. Thus, they cannot be chosen as helpers anymore and the number of false validations will remain same.

Varying parameter γ. Parameter γ is used to adjust the punishment degree of dishonest validations. Larger γ will cause lower trust value of the device when the tracker detects it has done dishonest validations. In this set of experiment, the setting is the same as that in the trust value evaluation experiment, except that γ is set to be 1, 2,



(a) Validation latency (b) Number of local decryptions (c) Ave. # of messages per cert

Figure 3.21: Performance varies with percentage of malicious devices after 2000s

5, 10 respectively. The malicious devices begin to provide false validations after 1000s. Fig. 3.19 and Fig. 3.20 show the trust values changes of malicious devices with different γ. There is a sharp decrease of the trust values near 1000s because of false validations provided by the malicious devices. The sharp decrease shows that the tracker detects the malicious behaviors and updates trust values of them timely. Different γ causes different degrees of decrease. Fig. 3.20(b) shows the largest decrease. The trust values dramatically decreases below 0.5. Thus we set γ to be 10 in our experiments, because it is sufficient to make malicious devices not chosen to be the collaborators once the dishonest validations are detected.

Varying the percentage of malicious devices. Fig. 3.21 shows the performance of CCV varying with the percentage of malicious devices, ranging from 0% to 50%, after 1000s. At that time, most malicious devices will be detected and not used but the capacity of the whole cache pool decreases. Hence we find that the average number of decryptions (Fig. 3.21(b)) increases for CCV protocol with more malicious devices. The average number of decryptions for I-Valid keeps increasing with time, which is shown in Fig. 3.8(b). It reaches 95.15 at 1000s, which is much larger than 20.97 for CCV. The latency (Fig. 3.21(a)) has a slightly increase from 0.469 to 0.635. The reason is that more malicious devices indeed decreases the capacity of cache, causing the increase of average latency. But the cache pool of the remaining good devices can still support such number of total certificates, thus the increase of the latency is small. The average message per certificate decreases from 1.88 to 1.75 in Fig. 3.21(c) because more individual validation is conducted. However, CCV protocol still achieves a much better performance on average latency and number of local decryptions, compared to I-Valid.

False validation attack: The evaluation of trust value changes has been analyzed and the results, such as those in Fig. 3.17, indicate that after a short period of time, the malicious devices are not able to conduct false validation attacks.

Self-promoting attack: Trust value of each device is maintained and updated by the tracker. It is hard for a malicious device to promote itself to be a collaborator.

Defamation attack: Each collaborative validation event must carry a digital signature of the holder for authenticity and non-repudiation purposes. The digital signature will be verified by the tracker. Hence a malicious device cannot forge a false validation event from a honest device unless it owns the private key of the honest device.

Traitor attack: As shown in Fig. 3.18(b), Fig. 3.19, and Fig. 3.20, once the malicious device provides dishonest validations, the trust value will drastically drop below the threshold, thus it can be selected as a helper in CCV anymore. Even the attacker intends to provide good validation when it senses its lower trust value. However, it does not have the chance anymore.

Whitewashing attack: The tracker can audit the identity of the device. Thus, the high cost will effectively prevent the whitewashing attack.

Collusion attack: Two or more malicious devices may improve their trust values by claiming that they helped each other. However, a malicious device will eventually provide a number of dishonest validations, which will be detected by the tracker statistically. In our model, the trust value reduction from one dishonest validation will be much larger than the trust value improvement from one honest validation. Hence it is only possible that the colluding malicious devices claim collaborative validations much more frequently than providing dishonest validations. Extremely high frequency of collaborative validations will also be detected by the tracker.

Sybil attack: The tracker can audit the identity of the device, which is shown in the bootstrap phase. Thus, the high cost will effectively prevent the Sybil attack.

3.5 Related Work

Certificate validation. Certificate-based PKIs are responsible for creating, managing, distributing, storing and revoking public key certificates, such as X.509. They are widely used in Web browsing (TLS), email (S/MIME) and document authentication. Efficient certificate validation has attracted a broad attention of the research community in recent years [72] [106] [26] [25] [81] [143] [73] [134]. Some approaches delegate validation task to a trusted third party, such as a smart gateway [106] [138], a delegation server [72] and local ISPs [26]. Some approaches use prefetching and prevalidation techniques to reduce certificate validation cost [134] [73], which can remove the time pressure from the certificate validation. However, this approach brings huge cost for memory. Some approaches aim to reduce cost for checking certificates revocation status [64] [97] [143]. For example, Gañán et al. [64] proposes a collaborative certificate status checking mechanism to distribute certificate revocation information in Vehicular Ad Hoc Networks, which can provide a quick response to check revocation status.

Trust Model. A trust model is used to build trust relations between two nodes in a network. Trust is mainly divided into direct trust and indirect trust. Direct trust is computed by direct communications, while indirect trust is calculated by recommendation. Some algorithms can be used to calculate direct trust, such as subjective logic [75] [76], fuzzy method [45] [130], Bayesian [8] and game theory [55]. For indirect trust, devices in the trust model can get trust information from their trusted devices (recommender) [6] [47], then the trust value can be updated. In the trust model, direct trust and indirect are always combined to calculate the trust value [6] [47] [61] [74] [154]. For example, Feng et al. [61] proposes the NBBTE algorithm to establish the direct trust and indirect values between two nodes by comprehensively considering and combining various factors. Another efficient distributed trust model (EDTM) has been proposed in [74], which considers both direct and indirect trust and takes more trust metrics such as the energy level information into consideration besides communication behaviors.

3.6 Discussion

One critical concern is the false validation results of the certificates provided by malicious IoT devices. In CCV, the IoT device will continue to communicate with another devices or servers when their certificates are validated, regardless of the wrong validation results from malicious devices. However, collaborative validation results will be also forwarded to the server on a sampling basis at the same time. The tracker will audit the validation results. If the tracker detects false validation results, it will send *FALSE_VALI_RESULTS* message to the corresponding device. The device will then stop the connection immediately when it receives the message. The latency to discover false certificates for the device is not too long. The main cost lies in the communication overhead between the device and the tracker. On the other hand, the tracker will punish the malicious device by drastically decreasing its trust value, which can be shown in Fig. 3.18(b). Thus, the malicious device cannot be chosen as collaborators anymore. To balance the overhead of the tracker and the security of whole system, all the certificates validation results are forwarded to the tracker at the beginning in order to quickly filter malicious devices. When the system is stable after a period of time, we can then set a proper sampling rate.

3.7 Summary

In this chapter, we design and evaluate the CCV protocol for fast public-key certificate validation. Our contributions include a memory-efficient and fast locator for certificate holders, called OLoc; a trust model for CCV to evaluate the trustworthiness of each device to avoid dishonest collaborative validation from malicious devices; and a complete protocol suite for efficient OLoc update, cache replacement, and revocation status checking mechanisms in a dynamic network. Evaluation results show that CCV significantly saves computation resource and validation latency on IoT devices.

Chapter 4

LOIS: Low-cost Packet Header Protection for IoT Devices

This chapter introduces the thesis work on protecting user privacy, one of the essential services for IoT devices. Most IoT devices (such as smart sensors) communicate in a passive and on-demand way: they transmit data to other devices or remote servers as necessary with minimum user involvement. For example, many IoT devices collect data from their surroundings or capture the users' activity information, then automatically transmit the data to the remote cloud servers for data analysis services.

However, as most of the sensing data contain sensitive information about the users, this cloud-based IoT service framework posts severe privacy leakage concerns [158]. One primary privacy concern is that some users' activities and their surrounding environment information are exposed to unknown parties, such as the passive adversaries in the network path, even though the payload data is encrypted with TLS/SSL. A successful attack can be as simple as *looking at the packet headers*! Researchers have found that the encrypted IoT traffic can still be used to infer the device identity or the related user activities by analyzing the packet headers and traffic patterns [126] [2] [100]. For example, when devices monitor users' sleep, or when devices surreptitiously record users' activity data such as audio [10], they need to send the data packets to the specific destination IP addresses (also called the service IPs) and ports of the cloud servers with a specific pattern. Hence, by observing the IP addresses, ports, time, and frequency of those encrypted data packets, the passive adversaries can successfully obtain some sensitive information of the users, such as the type and function of IoT devices [103], the type of the activity, and the communication pattern information [2, 24]. Therefore, to enhance the data security of these IoT services, it is of significant importance to protect not only the privacy of the service data but the *headers* of the packets generated during the services.

Thus, we focus on an important yet challenging problem: hiding sensitive packet header information in IoT traffic to protect user privacy. The requirements to achieve this goal are summarized as follows.

1) **Oblivious service IPs.** The destination IP of an IoT packet needs to be oblivious to a passive adversary. The packet header cannot reveal which application or service the packet is used for. In particular, a cloud may host many IoT services, and each service is usually assigned a dedicated IP address called the service IP [56, 65]. Hence the service IP should be hidden from the passive attackers in the network path.

2) **Hide device identity and activities.** Adversaries cannot link the identity of an IoT device to the packets sent by it.

These requirements are challenging for two reasons. First, packet header fields, such as IP addresses and port numbers, are used to identify the packets by the upper layer applications and route traffic to their destinations. Hiding this information without influencing its function is challenging. Second, IoT devices are resource-constrained. Hence, to make it friendly for IoT, the protection method should incur light overhead.

An intuitive solution for protecting packet headers is to use virtual private networks (VPNs), as suggested in a recent study [28]. A VPN tunnel can wrap all traffic through the tunnel by encrypting the packets, including their headers. However, using VPNs has several limitations. 1) VPN performs encryption and decryption operations on both packet headers and payloads, causing considerable performance degradation, as shown in our evaluation in Sec. 4.6. In fact, the majority of IoT packets are small packets [100], and the payloads have already been encrypted by TLS/SSL [121, 158]. Hence the overhead is not necessary. 2) VPN technology cannot protect the device's identity (IP address) when building a VPN tunnel between the device and the remote server.

We present a system for **L**ightweight **O**blivious **I**oT **S**ervices called LOIS, which achieves the above requirements of protecting packet headers and hence user privacy. The main idea is built upon the fact that some major cloud providers, such as Amazon and Google, host a large number of IoT applications – either by themselves or their customers. Hence, each such cloud can offer a unified IP for all the supported services, which can be the destination address for all the packets to the cloud. The unified IP to server IP translation can be performed on the cloud load balancers, which are currently doing virtual IP (VIP) to server IP translation [99]. Then the sensitive fields in the packet header are encrypted using *stream cipher* with a one-time keystream chosen from a list of keystreams of the requested service, which can protect the device identities on the entire path to the server with much less overhead compared with VPNs.

The proposed LOIS framework consists of the following modules: the keystream management module and the packet header modification module. In addition, LOIS can effectively integrate the stochastic traffic padding (STP) algorithm [28] to defend against traffic analysis attacks. We implement LOIS using the Intel data plane development kit (DPDK) on commodity servers and compare it to IPsec and the pure forwarding method (Vanilla). We find that LOIS incurs a small end-to-end overhead for the bidirectional traffic, around 250–365 ns on average for the upload traffic and around 164–250 ns for the download traffic, which is 80%–90% less compared with IPsec. In addition, in contrast to IPsec, LOIS can directly send the download packets from the server to the device, which significantly saves the network bandwidth for the load balancer.

4.1 Characteristics of IoT traffic

In this section, we analyze some characteristics of IoT traffic. We adopt a public-available IoT traffic data set [131], which contains the network traffic of 28 unique IoT devices in a smart home representing six different categories: cameras, switches, and triggers, hubs, air quality sensors, electronics, and healthcare devices. We download the 20 days of IoT traffic data for analysis.

We first find that it is very easy for a passive adversary to infer which IoT application a packet is used for because **the destination IP and port of the packet**

headers are completely visible although the payloads are encrypted by TLS/SSL.

We then choose one device from each category to study the distribution of the packet size. Fig. 4.1 shows the statistics result. We can find that IoT devices tend to exchange a small amount of data in each packet. Most of the packet sizes are less than 250 bytes.

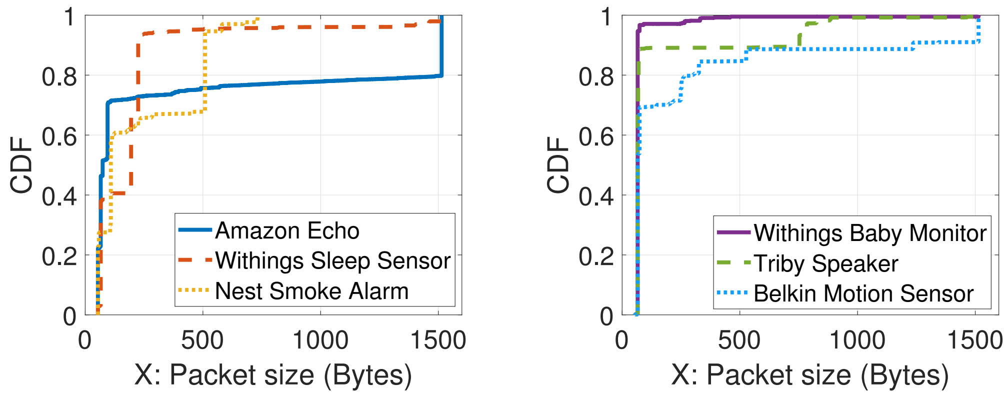


Figure 4.1: Distribution of IoT packet size

We further measure the average number of packets generated or received by each device in one day. Fig. 4.2 shows the results of twelve different IoT devices. We can find that the average number of packets in one day is quite different for various IoT devices.

Next, we use one-day traffic data to measure the average traffic rate for different devices. The data were collected from 7:00 am. We sum the traffic size in one second to calculate the average traffic rate per second. Fig. 4.3 shows the results of the Amazon Echo and Belkin Motion Sensor. We can find that different devices have different traffic patterns. For example, the average traffic rate is around 1 KBps for Echo and 15 KBps for the Belkin motion sensor. In addition, we find there are many clear pattern changes (such as the temporal traffic peaks) that can tell some specific user activities for both

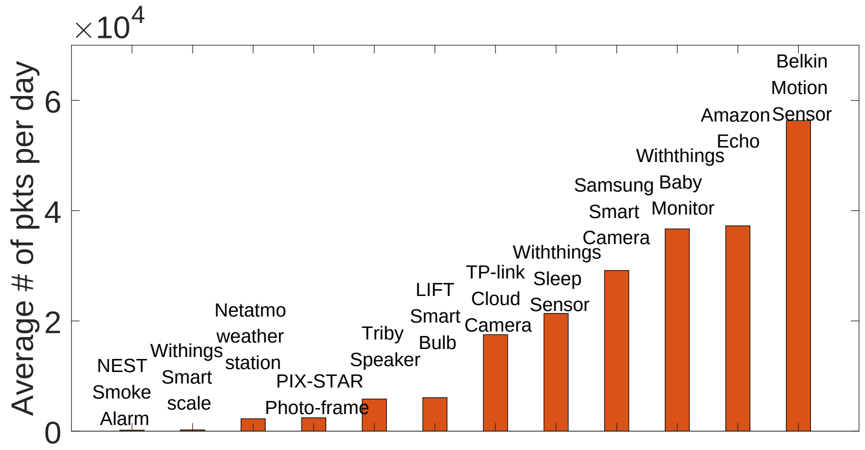
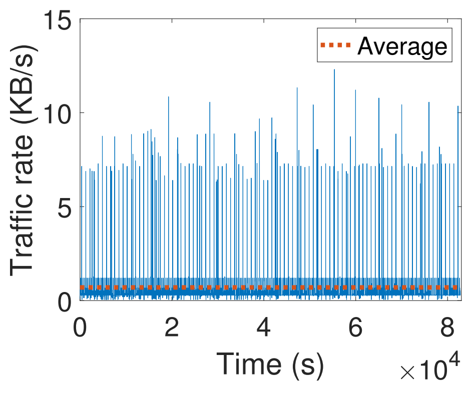
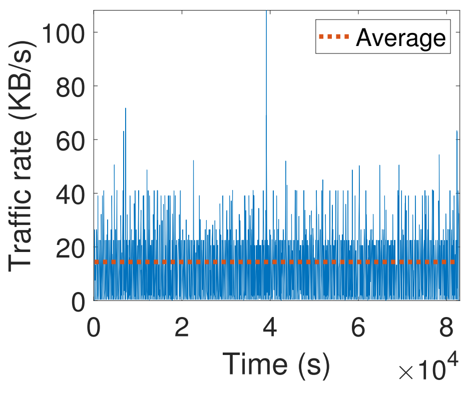


Figure 4.2: Average number of packets

applications. Passive adversaries can utilize the traffic pattern to infer device type and user behaviors. For example, an attacker can use machine learning models to classify the traffic types and detect or recognize the user's activities from the encrypted IoT packet trace data [101, 137].



(a) Amazon Echo



(b) Belkin Motion Sensor

Figure 4.3: Average traffic rate

Take-aways. The headers of the IoT packets are completely not protected, and they can reveal user privacy. The sizes of IoT packets are small, mostly under 250B. The traffic rates are low, $1-15$ KBps on average, and their patterns also reveal user privacy.

4.2 Models and Problem Statement

4.2.1 Network model

We consider the scenarios where the IoT devices communicate with the cloud servers for some service tasks such as sensing data reporting and analysis. The IoT device connects to the Internet through an access point. Fig. 4.4 shows the network model, which consists of the following four main components.

1) IoT devices. An IoT device (or "device" in short) is an object with sensors or actuators, which has constrained computing, memory, and power resources. The devices can sit in an IoT community, such as the smart home. Each device connects to the Internet through an access point. The devices capture and collect data, then transmit the data to the cloud. The devices may also request services from the cloud.

2) Access points. An access point is a network device that helps IoT devices to connect to the Internet. The access point usually connects to a router as a standalone device. In some network setups, the access point is an integral component of the router.

3) Servers. Servers, which provide various services for IoT devices, are located in the cloud.

4) Load balancers. A load balancer, which is deployed in a cloud, manages the traffic to the cloud. The load balancer should process every packet to the cloud in both current practice and the LOIS system. It is worth noting that packets sent from servers to devices do not need to pass through the load balancer. Load balancers int the cloud

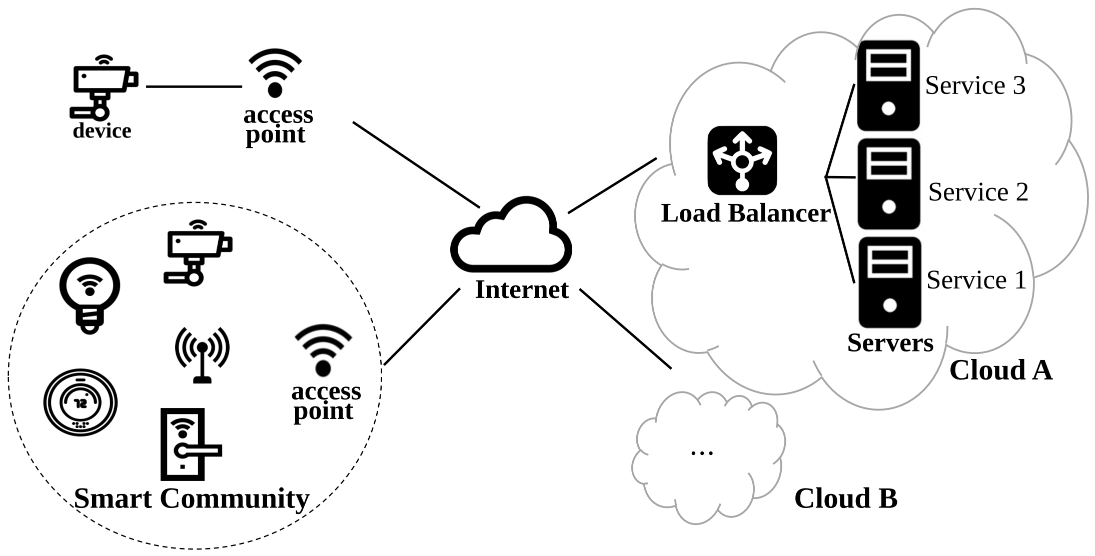


Figure 4.4: Network model

provide a unified IP address called CIP for the IoT devices to request all the supported services and use unique service IDs (SID) to distinguish services. The load balancer is responsible for translating the unified IP address and the service ID to the destination server IP and the port number.

4.2.2 Threat model

We assume the traffic between IoT devices and remote cloud servers is encrypted using the TLS/SSL protocol. Thus, the traffic packet content is not accessible to the entities except for the sender and the receiver. The servers and the load balancer in the cloud are trusted.

We are concerned with the passive adversaries that can collect network traffic and infer the user's private information from the traffic data. Although the passive adversaries cannot view the packet payload of encrypted traffic, they can easily view the source and destination IP addresses and the port numbers from the packet headers, and infer the device identity and requested service information. Furthermore, the passive adversaries can get traffic rates, inter-packet intervals, and packet size information to infer more sensitive information, such as the device type and users' activities that trigger the traffic. The passive adversaries can reside along the path from the device to the access point, from the access point to the cloud, or within the cloud. We divide the adversaries into two different categories.

1) Local adversaries. Local adversaries are the passive adversaries that are located on the path from the device to the access point. Or, if the IoT device is deployed in a smart community, local adversaries sit in the local area network (LAN). Local adversaries with access to the Wi-Fi network can view all the packet headers. Local adversaries without access to the Wi-Fi network can only know the sending time of the packet and view the link layer header information, including MAC address, sizes of Wi-Fi packets, while other information like IP headers and transport layer headers are encrypted.

2) External adversaries. External adversaries can view the traffic only after packets leave the access point. They are either located along the path from the access point to the cloud (on-path adversaries) or sit within the cloud (cloud adversaries). So external adversaries can view all the IP header and transport layer header information. But they cannot get the MAC address of the packets. One representative external passive adversary is the Internet Service Provider (ISP). We assume the cloud provider and the servers are trusted, but there could be a passive adversary in the cloud network, such as a compromised router. If the servers (the receiver) are not trusted, all possible protections of packet headers will fail.

We assume adversaries do **not** have the power to act as **global passive adversaries** that can observe both the traffic inside the cloud and outside the cloud. They can only sit in one place of the network path and view part of the traffic.

4.2.3 Problem specification

The detailed objectives of LOIS are: 1) Hidden service. If a passive adversary is a local adversary or a on-path adversary, it cannot know the services IPs and corresponding ports of the packet. 2) Sender anonymity. Passive adversaries cannot discover the identity of the IoT device (IP address) for the upload traffic sent from the device to the cloud. 3) Packet unlinkability. Passive adversaries cannot determine whether the packets belong to the same connection by directly viewing packet information. This property helps to defend against traffic analysis attacks or user tracking based on packet header information. We also consider the traffic statistic analysis attack based on the traffic pattern information. We assume that passive adversaries can gain prior knowledge about the characteristics of IoT traffic. Thus, they can utilize this knowledge to identify the device type and infer related user activities. We integrate the existing stochastic traffic padding (STP) algorithm [28] to defend this attack.

4.2.4 VPN is not an optimal approach for our goals

In this section, we discuss the feasibility of utilizing the VPN technology to defend against passive adversaries. Fig. 4.5 shows the design. Here, the load balancer scales out services hosted in the cloud by mapping packets destined to a provided service with a virtual IP address to a pool of servers with multiple direct IP addresses [99]. We

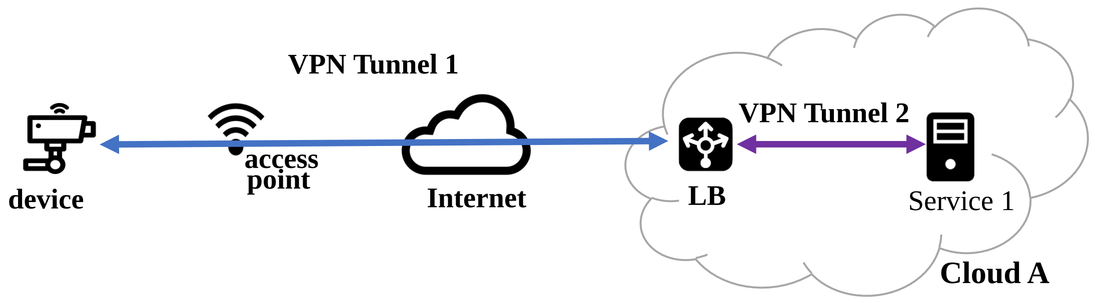


Figure 4.5: VPN tunnels

build two VPN tunnels: Tunnel 1 wraps all traffic between the IoT device and the load balancer, and tunnel 2 wraps all traffic between the load balancer and a server. For the upload traffic sent from the device to the server, the requested service is hidden by tunnel 1, and the device identity and service type are protected by tunnel 2. *Since servers that provide services dynamically change, the client cannot directly build a VPN tunnel between the device and the target server.* Therefore, the download traffic also needs to pass through the load balancer and be protected by tunnel 1 and tunnel 2.

However, there are several limitations of utilizing VPN technology to solve the problem. 1) VPN tunnel 1 cannot provide sender anonymity, exposing the device's IP address to passive adversaries. On-path adversaries can simply collect traffic with the same IP address to a cloud within a period, linking these packets to a connection. Thus, VPN technology cannot achieve our objectives. 2) For bidirectional traffic, the VPN technology poses two encryptions and two decryptions on every packet, causing considerable computational overhead. 3) The download traffic also needs to pass through the load balancer, bringing extra computational overhead and bandwidth consumption on the load balancer.

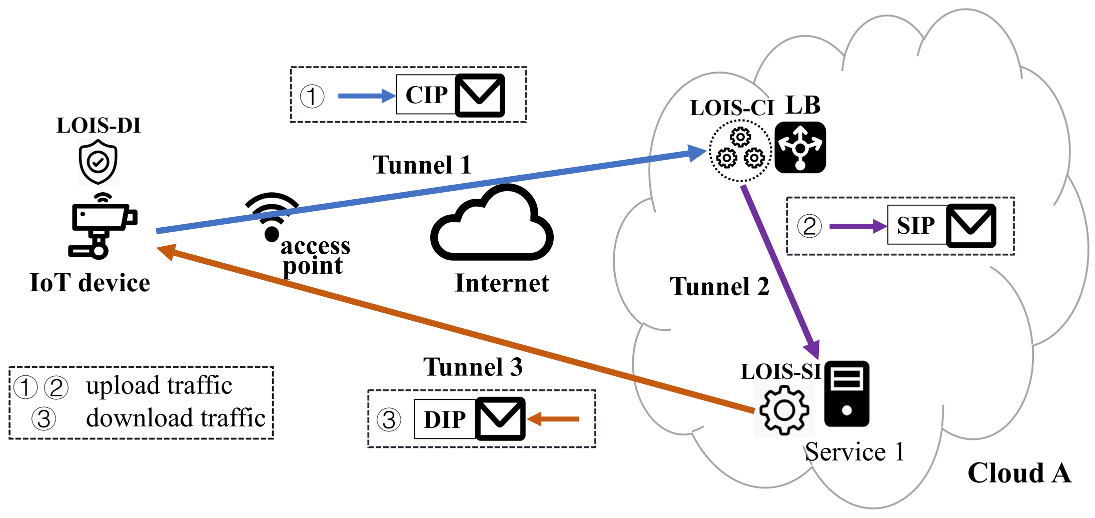


Figure 4.6: Overview of the LOIS framework

4.3 Overview of The LOIS Framework

Fig. 4.6 shows the overview of the LOIS framework. LOIS is installed as three types of interfaces: 1) the device interface called LOIS-DI; 2) the interface on cloud load balancer called LOIS-CI; and 3) the server interface LOIS-SI. Each interface is responsible for certain operations on packet headers.

Each LOIS-CI is assigned an IP address (CIP) which is served as the destination IP for all the services it manages. Simultaneously, LOIS-CI provides a service ID (SID) for every managed service. The CIP and the SID are used to identify the requested service by setting them to be the destination IP address and the destination port number of a packet sent to the cloud. LOIS-CI is implemented on the load balancer. LOIS-DI is directly implemented on the device, and the IP address of this device is called DIP. LOIS-SI is implemented on each server that provides services. Each LOIS-SI is assigned with an IP address called SIP.

For the upload traffic which is sent from the device to the remote cloud server, LOIS performs as follows.

Step 1. LOIS-DI hides all the sensitive information in the packet header. Adversaries can only view that the packet is sent to the LOIS-CI with CIP.

Step 2. LOIS-CI only recovers the IP address and the port number of the target server. Other fields are hidden to prevent passive cloud adversaries from inferring sensitive information from the packets, such as the source device and the flow-related information. Then, the packet is sent to the target server with SIP. After the server receives the packet, it will recover all the hidden fields.

For the download traffic, LOIS performs as follows.

Step 3. LOIS-SI directly hides all sensitive fields and sends it to the destination device with DIP. Adversaries can only view that the packet is sent to the DIP.

The main idea behind the LOIS framework to hide the packet metadata is to use *a one-time keystream* to encrypt sensitive fields in the packet header for each packet. There are two main modules which are listed as follows.

(1) **Keystream management module** generates, stores, and updates keystream-s. Simultaneously, a LOIS handshaking process is proposed for LOIS-DI and LOIS-CI to create the same keystream table.

(2) **Packet header modification module** modifies the packet header to protect the packet metadata information.

As the traffic pattern exposes times and user activities, the LOIS framework integrates the existing traffic padding algorithms called STP [28] to hide it. The traffic pattern is hidden for the traffic in tunnel 1, tunnel 2, and tunnel 3.

LOIS-DI can also be implemented on a local server or any programmable network middlebox, such as Wi-Fi access point or access gateway router. Multiple devices can sit behind LOIS-DI, and LOIS-DI protects those managed devices by preventing external adversaries from sensing sensitive information with less overhead compared to the VPN method.

4.4 Design of the keystream management module

Each packet requires one unique keystream to encrypt sensitive fields in the packet header. For each packet, the receiver needs to know which keystream is used without exposing the keystream itself. Hence we use part of each keystream as an identifier. The identifier is sent in plain text and used by the receiver to identify the keystream used in the packet and recover the original packet header. Identifiers are in plain text, but passive adversaries cannot infer the corresponding keystream from an identifier without knowing detailed identifier-to-keystream mappings. These mappings are shared secret by the IoT devices, cloud load balancer, and servers. Besides knowing the keystream for an identifier, its related service ID also needs to be stored to obtain the service information from an identifier efficiently. In LOIS, the identifier is the key k, and the keystream is the value v for key-value lookups. Fig. 4.7 shows an example of the key-value item that needs to be stored in the LOIS framework. The identifier is used for two purposes: 1) to retrieve the keystream from the local mapping table at the packet recovery side and decrypt the packet header; 2) to get the service ID.

There are three requirements for the design of the keystream management module, which are summarized as follows. 1) Each communicating LOIS pair, which contains one LOIS-DI and one LOIS-CI, owns the same list of keystreams for all the requested

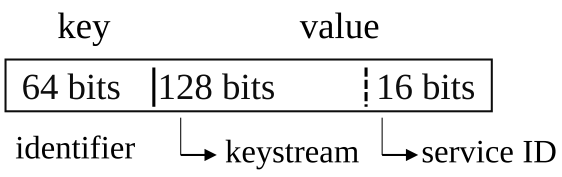


Figure 4.7: An example of key-value item

services by the device. Also, LOIS-DI and LOIS-CI need to distinguish keystreams for different services. 2) Keystreams cannot be used twice to prevent adversaries from recovering encrypted fields by the reused key attack. 3) Efficiently obtaining the keystream for each packet is required.

4.4.1 Keystreams generation

Keystreams are generated using a pseudorandom number generator by agreeing on a seed and a nonce. Then the long keystream is cut into segments, following the format as shown in Fig. 4.7 (we set 64 bits for identifiers in LOIS). Passive adversaries cannot infer the corresponding keystream according to the identifier unless they know identifiers-to-keystreams mappings. Since the traffic are bidirectional and packets in the upload traffic and the download traffic both need keystreams to protect packet metadata, each keystream table contains keystreams for both request packets and reply packets.

In this work, we propose a LOIS handshaking process to generate the same list of keystreams and their identifiers for a LOIS pair. We assume LOIS-DI and LOIS-CI have already built a secure channel based on the public key infrastructure (PKI). The LOIS handshaking process is initialized by LOIS-DI when it tends to request services from a cloud. LOIS-DI sends a hello message to LOIS-CI, which includes a nonce, a random number r_d, requested services information, the number of keystreams k for each service. After LOIS-CI receives the hello message, it also generates a random number r_c and sends it to LOIS-DI. The seed s is generated based on $< r_d, r_c >$, which is unique for different LOIS pairs. After the message has been exchanged in the handshaking process, both LOIS-DI and LOIS-CI generate keystreams using the same seed s.

4.4.2 Keystreams storage

The LOIS framework needs to use **a key-value table** to store keystreams information which is shown in Fig. 4.7, **with unique requirements**: 1) It is well known that using two identical keystreams is dangerous for a stream cipher. Thus, each identifier in the table of a device should be unique. 2) Each cloud serves many devices simultaneously. Hence **global duplication** should be applied for these tables; 3) The table should cost very small memory and support fast lookup and update operations.

Core algorithms. To meet the requirements, our innovation is to improve a recent tool called Vacuum filters [142], which is an enhanced version of cuckoo filters [59] and achieve $O(1)$ lookup time and amortized $O(1)$ insertion time for approximate membership queries. We *enhance the original Vacuum filter design for a space-efficient and fast key-value store with deduplication across the tables on different devices.* Instead of storing the complete key, our design stores the fingerprint of each key to save memory. Thus, we call the designed key-value store as a partial key Vacuum table. We adopt 16 bits fingerprints in the experiments. Fig. 4.8 shows the (2,4)-partial key Vacuum table (hereinafter called the (2,4)-PK Vacuum table). A (2,4)-PK Vacuum table consists of a number of buckets. Each bucket has four slots. (2,4)-PK Vacuum table stores the l-bit digest of k, which is represented as f, to save memory. Every (f, v) pair is stored in one

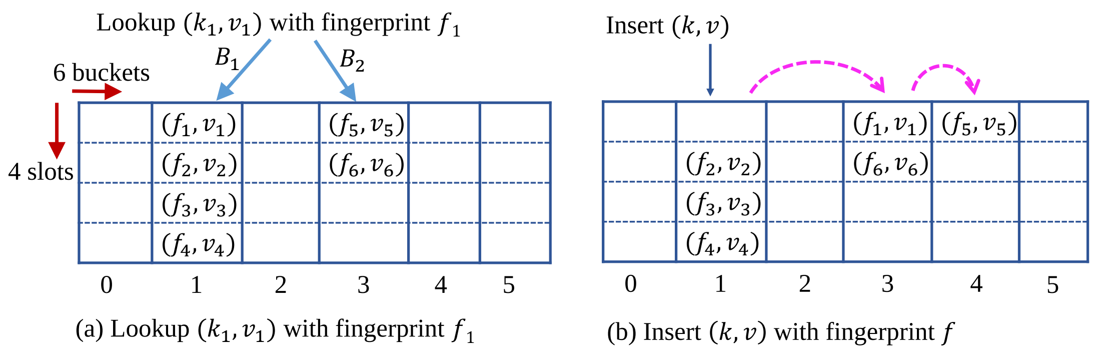


Figure 4.8: Partial key vacuum table

slot of the two candidate buckets B_1 and B_2 based on the hash values of the key k.

(2,4)-PK Vacuum table insertion. For any (k, v) pair, (2,4)-PK Vacuum table stores its fingerprint and value (f, v) in an empty slot in bucket $B_1(k)$ or $B_2(k)$. If neither $B_1(k)$ nor $B_2(k)$ has an empty slot, the PK Vacuum table will perform the **eviction** process. It chooses a non-empty slot in bucket B (B is $B_1(k)$ or $B_2(k)$). The (f', v') stored in the slot ((f_1, v_1) in Fig. 4.8) will be removed and replaced by (f, v). Then (f', v') will be placed to a slot of its alternate bucket. If the alternate bucket is also full, the PK Vacuum table recursively evicts an existing stored pair (f'', v'') to place (k', v'), and looks for an empty slot for (f'', v''). When the number of recursion process exceeds a predefined threshold, this insertion is failed and a reconstruction of the whole table is required.

(2,4)-PK Vacuum table lookup. The lookup of value for a key k is to fetch the two candidate buckets and match the fingerprint f of the key in all eight slots until a fingerprint matches f. Then, the corresponding value of the key k is obtained. Otherwise, the item is not stored in the table.

In this work, we choose (2,4)-PK Vacuum table to achieve high memory utilization and fast lookup throughput. One particular requirement is that no duplicate identifiers and no keystreams exist in the table. The PK Vacuum table can detect the duplication of identifiers by first querying the table with the identifier. If the result is negative, the identifier is unique. To detect the duplication of keystreams, we store fingerprints of the existing keystreams in a Vacuum filter. Therefore, to successfully insert a key-value item (k, v), the first step is to query the Vacuum table with the identifier k and query the filter with the keystream. If both results are negative, this key-value item can be inserted into the PK Vacuum table.

Update of keystreams table. Keystreams are dynamic in the LOIS framework with new keystreams that are continuously inserted. A single PK Vacuum table and the corresponding Vacuum filter are not sufficient because the table's size may not be big enough to store all the key-value items with continuous insertions after the table is created. In this work, we design a dynamic partial key Vacuum table (DVT), which is inspired by dynamic cuckoo filter [46]. The DVT leverages the PK Vacuum table as the building block and consists of a number of s linked homogeneous PK Vacuum tables. Initially, a DVT consists of only one PK vacuum table. The DVT will extend its capacity by linking a new PK Vacuum table. Each PK Vacuum table in the D-VT has the same number of buckets. We also adopt a dynamic Vacuum filter (DVF), which consists of several linked Vacuum filters, to store fingerprints of keystreams in the dynamic environment with an expansion set of keystreams. We maintain consistency between the DVT and DVF, in which the fingerprints of keystreams that are stored in the i-th linked PK Vacuum table are stored in the i-th linked Vacuum filter. To insert a keystream, its identifier and the keystream will first need to be queried in the DVT and the DVF, respectively, to check duplication. If both results are negative, this key-value item can be inserted into the DVT, and the keystream is inserted into the DVF. The insertion algorithm of the DVT and DVF is similar to that in the dynamic cuckoo filter.

Lookup of keystreams. The lookup of a keystream by its identifier from the DVT requires to probe every PK Vacuum table in the DVT. If a matched fingerprint of the identifier is found, the corresponding keystream is returned. Since keystreams are consumed, and each keystream is only used once, we also maintain a consumption rate for each PK vacuum table in the DVT. Once a keystream is used by querying it with the identifier, the PK Vacuum table's consumption rate that stores this key-value pair is updated. If the consumption rate of one PK Vacuum table reaches 100%, this PK Vacuum table is deleted from the DVT. Simultaneously, the corresponding Vacuum filter storing fingerprints of keystreams is deleted from the DVT.

Storage. Since LOIS-CI serves many LOIS-DIs simultaneously, LOIS-CI maintains much more keystreams than each LOIS-DI. After agreeing on the same seed and the nonce between a LOIS pair to generate a number of keystreams for both request packets and reply packets, LOIS-CI may detect that some keystreams cannot be used due to duplication. Thus LOIS-CI will send duplicate keystreams to LOIS-DI to notify LOIS-DI not to choose these keystreams, making consistency of keystreams without duplication on LOIS-DI and LOIS-CI. Fig. 4.9 shows the storage of keystreams for the LOIS. LOIS-CI stores mappings from identifiers to <keystream, SID> pairs for the u-pload traffic. LOIS-DI stores unused <identifier, keystream> pairs of each service for the upload traffic. Besides, LOIS-DI stores mappings from identifiers to <keystream, SID> pairs in a DVT for the download traffic of all required services.

Each LOIS-SI builds a smaller dynamic PK Vacuum table that only contains

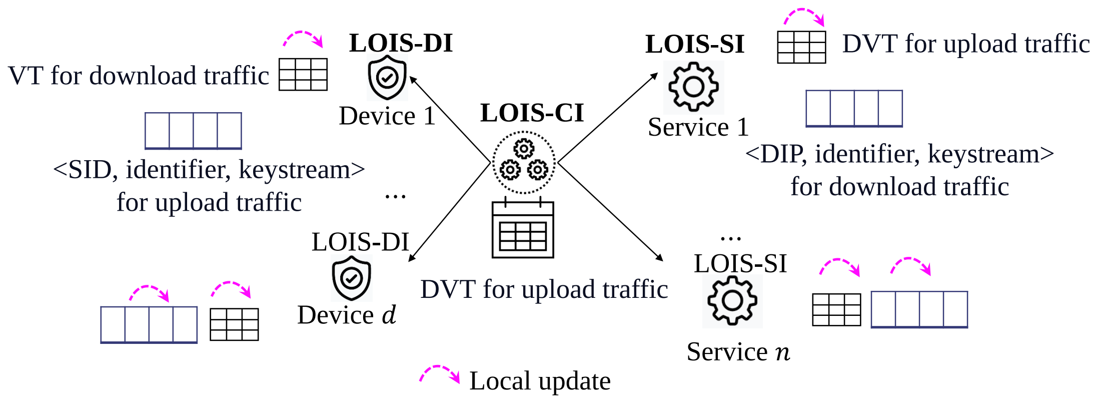


Figure 4.9: LOIS keystreams management

mappings from identifiers to <keystream, SID> pairs of its provided services for the upload traffic for all ongoing served LOIS-DIs. To construct the smaller DVT for LOIS-SI, LOIS-CI sends metadata including seed, the identity of each LOIS-DI called DIP, number of keystreams, and duplicated keystreams to LOIS-SI. Then, LOIS-SI generates the same list of keystreams as LOIS-CI and locally updates the DVT. Additionally, LOIS-SI keeps unused <identifier, keystream> pairs for the download traffic of different LOIS-DIs.

4.4.3 Obtaining the keystream for each packet

LOIS-DI needs to hide packet headers for the upload traffic, and LOIS-SI needs to hide those of the download traffic. Since LOIS-DI and LOIS-SI store unused <identifier, keystream> pairs of requested services for the upload traffic and the download traffic, respectively, they directly extract an unused pair for an incoming packet.

LOIS-CI and LOIS-SI need to recover hidden fields for the upload traffic in packet headers. They query their DVT to get the keystream using the extracted identifier from the packet header, which will be illustrated in Section 4.5. LOIS-DI performs the same way for the download traffic. Querying a DVT using the identifier does not leak information because attackers cannot infer the keystream from the identifier.

4.5 Design of the packet header modification module

To hide sensitive information in the packet header, LOIS first classifies the header's fields into four groups.

(1) **Endpoint-related fields** contain the source address, the destination address, source port, and destination port fields. Destination address and destination port number expose the service type requested by IoT devices. Importantly, passive adversaries can group traffic for different endpoints according to endpoint-related fields. Thus, these fields need to be hidden.

(2) **Flow-related fields** contain the sequence number, the acknowledgment number, URG, ACK, PSH, RST, SYN, and FIN fields. Sequence numbers are related in each flow, leaking flow information. SYN and FIN fields indicate the start and end of a TCP flow, respectively. After adversaries identify the flow by combining endpoint-related fields and flow-related fields, they can get flow volume, flow duration, and pattern information further to infer devices' identity and the requested service. Thus, LOIS needs to hide these flow-related fields to leak no flow information.

(3) **Affected fields** contains total length, header checksum, data offset, checksum, and TCP options fields. Although these fields reveal no sensitive information, they need to be changed due to the modification of other fields.

(4) **Unchanged fields** contains all the remaining fields. They do not need to

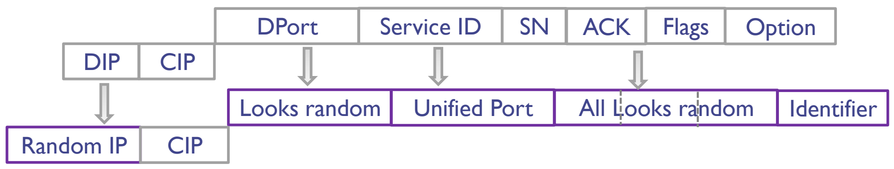


Figure 4.10: Packet header modification for upload traffic

be modified in the LOIS framework.

Unlike the VPN technique that encrypts all the fields in packet headers, LOIS carefully hides sensitive fields by applying two different schemes: replacement and XOR encryption. The replacement scheme is to replace the field with a unified one that cannot be distinguished. The XOR encryption scheme uses the XOR operator to encrypt the plaintext P by a keystream Q to get ciphertext C, which means that $C = P \oplus Q$, where P and Q have the same length. Specifically, P can be easily recovered by $C \oplus Q$. The keystream is generated and managed by the keystream management module. In this section, we assume that the keystream and its identifier are successfully generated for each packet.

Packet header modification for upload traffic. This part describes step 1 and step 2 presented in Section 4.3 and Fig. 4.6. **Step 1.** Fig. 4.10 shows the comparison of the main fields in the packet header before modification and after modification. In our design, the destination IP address and the destination port number of the packet are the unified cloud IP (CIP) and the service ID, respectively. However, the destination port number will expose the service information. Thus, it is then replaced by a unified port number to hide the service information. Therefore, adversaries cannot get the target service information by only directly viewing packet headers. Dealing with source addresses and port numbers faces two cases. 1) The device is located in a smart community and sits behind NAT. In this situation, LOIS-DI also integrates the NAT function by building a NAT translation table. LOIS-DI will negotiate with the gateway router to assign a unique port for this device. Therefore, LOIS-DI will first rewrite the source IP address of all the traffic from the device to a public IP of the smart community and use the assigned port number to the flow. The rewritten IP address and the port number are DIP and DPort in Figure 4.10. Then, these two fields will be encrypted by the keystream. 2) The device is assigned a public IP to connect to the Internet directly. Then, LOIS-DI encrypts the source IP and source port number of the upload traffic to hide these sensitive fields. LOIS-DI also encrypts flow-related fields and modifies affected fields by the keystream. Here, LOIS-DI assigns an unused keystream for the requested service as the target keystream. To notify LOIS-CI of the keystream without exposing it, we insert the keystream's identifier in the TCP option field. Finally, the total length field and the data offset field are adjusted due to the option field's insertion. Due to the modification of packet headers, LOIS-DI recomputes IP header checksum and checksum in the TCP header and then sends the packet to the cloud. **Step 2.** LOIS-CI only needs to set the IP address and the port number of the service server. After LOIS-CI receives the packet, it extracts the identifier and gets the keystream and the service ID by querying the DVT. The LOIS framework should also be compatible with the function of the load balancer. We assume the load balancer uses 5-tuple to select the SIP, which is a typical design for the load balancer [99]. Thus, LOIS-CI needs to decrypt the source IP and the source port number using the keystream, while these two fields remain hidden in the packet. LOIS-CI utilizes 5-tuple and the service ID information to translate CIP and the service ID to the target server IP address and the

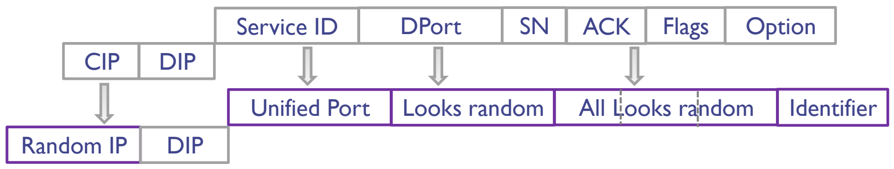


Figure 4.11: Packet header modification for download traffic

port number, then LOIS-CI modifies the destination IP address and the port number of the packet and sends the packet to its target server. After the server receives the packet, it queries local DVT to get the keystream after extracting the identifier and recovers all the hidden fields of the packet.

Packet header modification for download traffic. This part describes step 3. As we have introduced, the traffic between IoT devices and remote servers typically contains a request packet and a reply packet. Besides, remote servers may proactively communicate with IoT devices, such as sending the command. Thus, packet headers for both bidirectional traffic need to be hidden. Otherwise, passive adversaries can infer sensitive information if we only hide packets in one direction. **Step 3.** LOIS-SI gets an unused identifier and the corresponding keystream by looking at the list that stores unused <identifier, keystream> pairs of the target service for DIP. Fig. 4.11 shows how to hide endpoint-related fields and flow-related fields. DIP, which serves as the destination IP address, remains unchanged. Other fields, including source IP address, destination port number, and the flow-related fields, are encrypted by the keystream. A unified port replaces service ID to hide the service information. Then the identifier of the keystream is inserted in the TCP option field. Similar to step 1, other affected fields will be adjusted. Packets with modified packet headers will be directly sent to the LOIS-DI, not passing through the load balancer. Passive adversaries only view that packets are sent to the device with DIP. When LOIS-DI receives the packet, it extracts the identifier and queries the local DVT to get the keystream and the service ID. Then, LOIS-DI uses the keystream to recover all encrypted fields. The service ID then replaces the unified port number. If LOIS-DI integrates the NAT function, LOIS-DI also rewrites the DIP and DPort to the private IP address and the port number.

In the practical deployment, each IoT device may connect to several clouds to request services. To support multi-cloud scenarios, each client independently maintains keystreams for different clouds, which are generated by different seeds using the LOIS handshaking protocol. More connected clouds consume more memory on LOIS-DI to store keystreams.

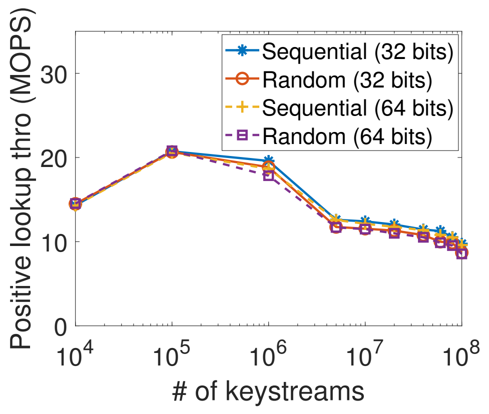
4.6 Implementation and evaluation

We implement the LOIS framework using Intel Data Plane Development Kit (DPDK) [11] in a public cloud experimental environment, CloudLab [7]. DPDK bypasses the complex network stack in the Linux kernel and processes packets in the userspace. CloudLab is a testbed for researchers running experiments with cloud architectures [7]. We use c220g2 nodes in the Wisconsin cluster to evaluate the performance of the LOIS framework. Each node is equipped with one Dual-port Intel x520 10Gbps NIC, with 8 lanes of PCIe V3.0 connections between the CPU and the NIC. And each node has two Intel E5-2660 v3 10-core CPUs at 2.60 GHz. The Ethernet connection between every two nodes is 2x10 Gbps. Logically, node 1 uses the DPDK official packet generator Pktgen-DPDK [15] to generate packets or get packets from a real IoT traffic dataset. Node 2 and node 3 work as the LOIS-DI and the LOIS-CI, respectively. Node 4 works as the LOIS-SI. In addition, we implement LOIS-DI on a Raspberry Pi 3 with one single 1.4 GHz processor and 1 GB RAM, which works as an example of a wide spectrum of devices that can use LOIS. LOIS can be easily implemented on less powerful IoT devices if they have available memory to store keystreams and the corresponding DVT, as shown in Fig. 4.21(a) (E.g., the device requires about 0.15 MB for 10^4 keystreams).

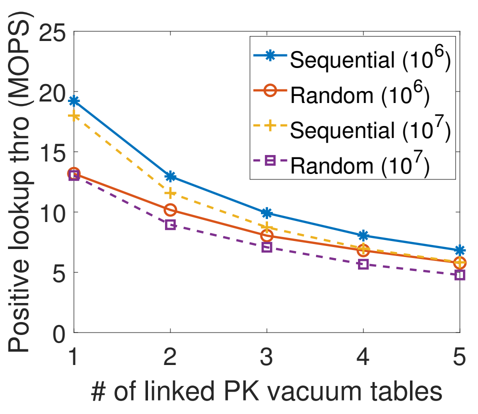
We compare our proposed LOIS with IPsec [13] – a protocol used in most VPNs – implemented using DPDK, and pure forwarding algorithm (hereinafter called Vanilla). We choose AES-CBC-128/SHA1-HMAC for the IPsec algorithm. The following two metrics are used to evaluate performance. (1) **Average latency** measures the average time caused by operations of LOIS-DI, LOIS-CI, and LOIS-SI. (2) **Throughput** measures the number of processed bits per second. Unless otherwise mentioned, we conduct five production runs, LOIS handles more than one million packets on each run.

4.6.1 Keystream benchmark

In this section, we evaluate the positive lookup performance. We measure the throughput in millions of operations per second (**MOPS**). Fig. 4.12(a) shows the positive lookup performance varies with the number of keystreams in one PK Vacuum table. Results show that the PK Vacuum table has good positive lookup throughput – a positive lookup means the keystream does exist in the table. Different access pattern of identifiers on one Vacuum table has little influence on the result. We further evaluate the performance with a different number of linked PK Vacuum tables. Fig. 4.12(b) shows the results. In this set of experiments, we adopt 64 bits identifiers, and the



(a) Lookup thro. varies with # of keystreams



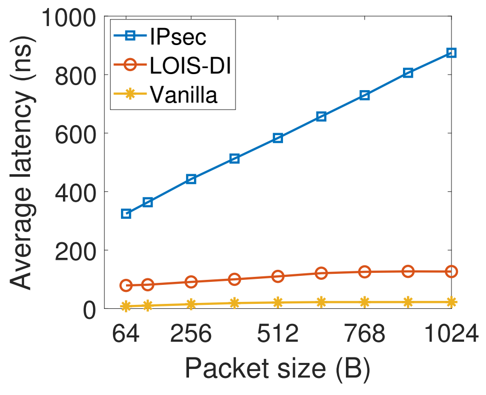
(b) Lookup thro. varies with # of linked PK Vacuum Tables

Figure 4.12: Performance of Keystream Benchmark

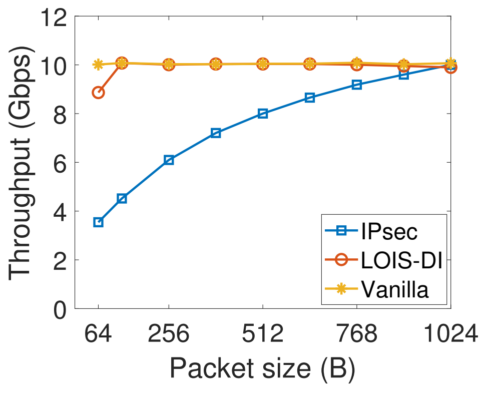
number of keystreams in one Vacuum table is set to be 10^6 and 10^7, respectively. We find that the positive lookup throughput decreases with more linked Vacuum tables due to more possible memory access. Sequential access achieves better lookup throughput than random access.

4.6.2 Evaluation of LOIS

Performance of DI for the upload traffic. We evaluate the performance varying with the packet size of the LOIS-DI for the upload traffic. Smaller packet size means more packets generated per second under 10 Gbps bandwidth. For the upload traffic, the node running LOIS and IPsec hides sensitive information on packet headers. Fig. 4.13 shows the results. In this experiment, the service for each packet is uniformly chosen from 50 services. We can find that LOIS brings small overhead compared to the pure forwarding algorithm, which shows the efficiency of LOIS. LOIS only accesses packet headers and modifies headers either by replacement or XOR encryption; both op-



(a) Latency



(b) Throughput

Figure 4.13: Performance of DI varying with packet size

erations are efficient. IPsec encrypts original packets, causing larger latency with larger packets. The result shows that LOIS-DI outperforms IPsec, only causing 0.14x-0.25x latency compared to IPsec. Furthermore, latency for LOIS-DI with different packet sizes is relatively stable, while packets with larger sizes require larger processing time for IPsec. Fig. 4.13(b) shows that LOIS-DI achieves 1.0x-2.5x throughput compared to IPsec.

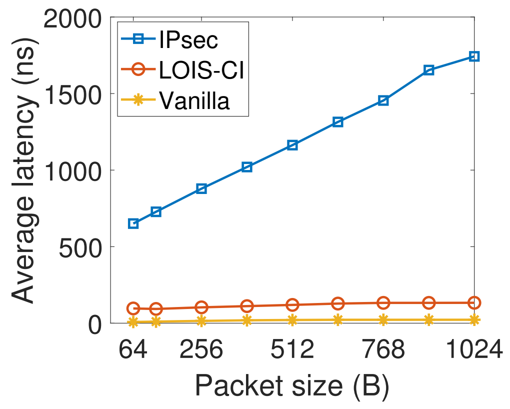


Figure 4.14: Performance of CI varying with packet size

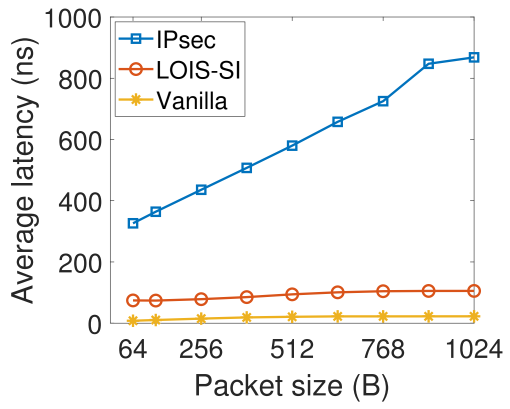
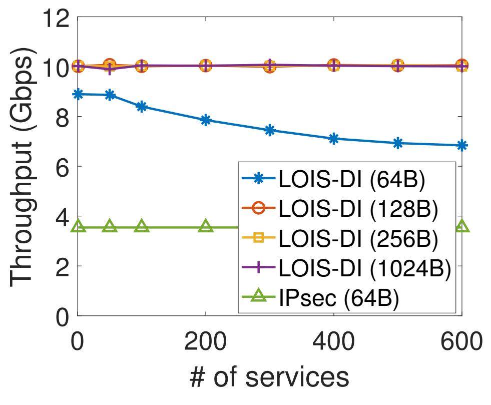
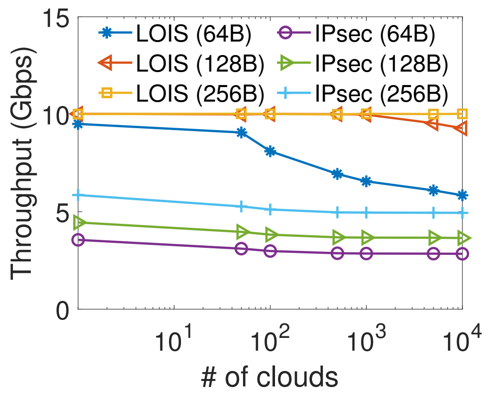


Figure 4.15: Performance of SI varying with packet size

Performance of CI for the upload traffic. We evaluate the performance



(a) Thro. varying with # of services



(b) Thro. varying with # of clouds

Figure 4.16: Performance of DI

varying with the packet size on the load balancer for the upload traffic, which is shown in Fig. 4.14. LOIS-CI extracts the identifier and queries the keystream from the DVT, but LOIS-CI only needs to recover the service ID and then set the destination IP address and the port number according to the service ID. For IPsec, it needs to first decrypt the packets, set the destination IP address and the port number, and then encrypt the whole packet, causing higher overhead. Results show that LOIS-CI only incurs about 0.1x overhead compared to IPsec.

Performance of SI for the upload traffic. We evaluate the performance varying with the packet size on the server for the upload traffic. Fig. 4.15 shows the average latency. We can find that the LOIS-SI brings low overhead on the server, while IPsec incurs higher overhead because the server needs to decrypt the whole received packet. The result shows that LOIS-SI only takes 0.12x-0.22x of IPsec's time for the operation on the server. Combining the total latency on LOIS-DI, LOIS-CI, and LOIS-SI, LOIS reduces the overhead to 10%-20% of IPsec's time for the upload traffic.

Number of services. We evaluate the performance of LOIS-DI for the upload traffic varying with the number of services. We generate packets with target services using a uniform distribution. The packet size is set to be 64 bytes, 128 bytes, 256 bytes, and 1024 bytes. Fig. 4.16(a) shows the results. We can find that the throughput is stable for packets with 128 bytes, 256 bytes, and 1024 bytes, showing that LOIS scales well with the number of services for larger packets. For small packets with 64 bytes, the performance slightly decreases with more services. Because more packets need to be processed per second on LOIS-DI, accessing identifiers and keystreams with more services increases the processing time. LOIS-DI still outperforms IPsec on small packets with 64 bytes.

Number of clouds. This part evaluates the performance for the upload traffic varying with the number of clouds. For LOIS, the client requests a total eight services from each cloud. Each service is assigned 1024 periodically updated keystreams. For IPsec, the client creates sessions with every cloud. Fig. 4.16(b) shows the performance for packets with 64 bytes, 128 bytes, and 256 bytes. Results show that LOIS outperforms IPsec, because IPsec needs to encrypt packet headers and payload, bringing considerable computation overhead. LOIS with 64 bytes has a performance degradation when the number of clouds is larger than 100, because obtaining keystreams with more clouds will influence the performance when the number of packets to be processed is large.

Number of clients. Since each cloud serves many clients concurrently, this part evaluates the performance varying with different number of clients. For LOIS, each client requests a total of eight services. Each service is assigned 1024 periodically updated keystreams. As the packet size for IoT traffic is small, we show packets' performance with 64 bytes, 128 bytes, and 256 bytes. Fig. 4.17 shows the result. We can find that

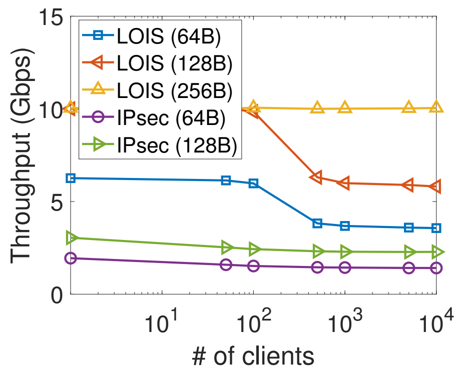


Figure 4.17: Performance of LOIS-CI varying with # of clients

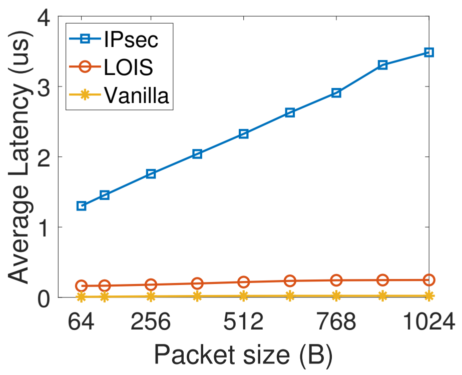


Figure 4.18: Average latency for the download traffic

LOIS outperforms IPsec, achieving >2x throughput. More clients influence the table size in LOIS-CI. The lookup throughput decreases with more keystreams in the table, as shown in Fig. 4.12(a). Therefore, for small packets (64 bytes and 128 bytes), the performance will decrease with more clients (>100). Because querying the table to get the corresponding keystreams to recover packets will dominate the performance with a larger table. Although the performance of LOIS on 64-byte packets and 128-byte packets has an oblivious decrease with a larger table, it still outperforms IPsec by a big margin.

Average latency for the download traffic. This part evaluates the total latency for the download traffic. For LOIS, the server directly sends packets to the target device. LOIS-SI hides sensitive fields, and LOIS-DI recovers packet headers. However, packets need to pass through the load balancer to get protection from VPN tunnel 1 and VPN tunnel 2 for IPsec, incurring larger overhead. Results are shown in Fig. 4.18. We can find that LOIS only incurs about 0.10x of IPsec's time.

Performance on IoT traffic. This part evaluates the performance on the

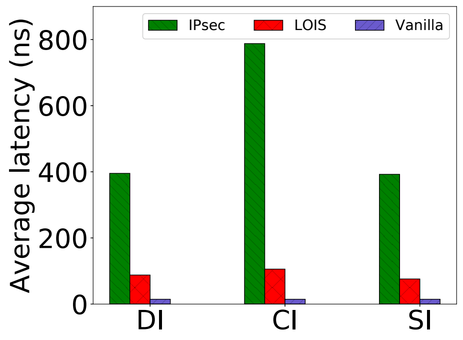


Figure 4.19: Performance on IoT traffic

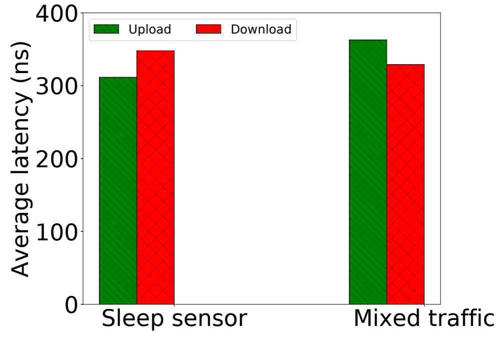
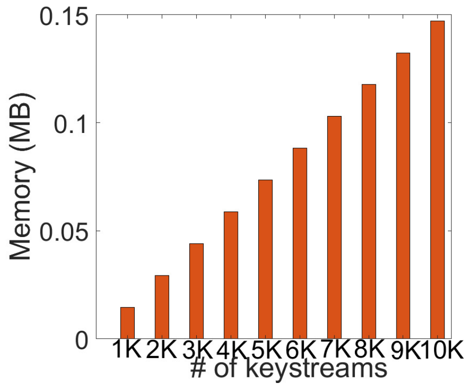
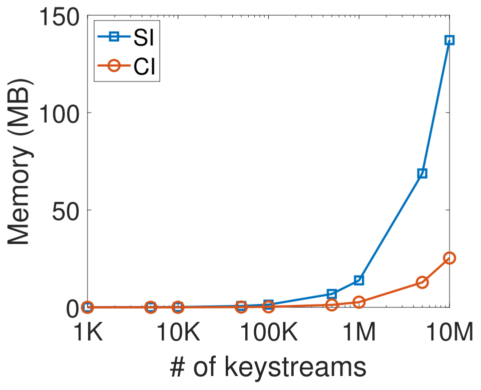


Figure 4.20: Average latency on Raspberry Pi 3



(a) Memory cost on LOIS-DI



(b) Memory cost on CI and SI

Figure 4.21: Memory cost

real IoT traffic, of which the packet size is small. We adopt one-day IoT traffic data from [131], which contains network traffic of 28 unique IoT devices (including sleep sensors). Then we utilize the packet size information from this mixed traffic to evaluate the performance, as shown in Fig. 4.19. Results show that LOIS brings small computational overhead and outperforms IPsec on real IoT traffic. In addition, we test the average latency of LOIS-DI for the upload traffic and the download traffic using the real IoT traffic dataset on a Raspberry Pi 3 testbed. Fig. 4.20 shows that LOIS incurs about

300 ns - 365 ns average latency on the device for one packet.

Memory cost. Fig. 4.21 shows the memory cost. We find that LOIS introduces a low memory overhead on the LOIS-DI, requiring about 0.15MB for 10^4 keystreams, as shown in Fig. 4.21(a). LOIS-CI stores mappings from identifiers to keystreams and services in the DVT, its memory cost is shown in Fig. 4.21(b). Since each keystream is only used once, one solution to further reduce the memory cost on LOIS-CI is to generate a small number of keystreams for each client and periodically update keystreams. LOIS-SI stores identifier-to-keystream mappings for the upload traffic and unused <identifier, keystream> pairs for the download traffic. LOIS-SI only stores a small number of keystreams for its provided services, decreasing the memory cost in practice.

4.7 Security analysis

We discuss the passive adversaries who intend to perform following behaviors.

(1) *Passive adversaries separate packets for different devices.* Since passive adversaries try to infer sensitive information from the traffic, the first step is to separate the traffic for different IoT devices to conduct the device identification and user activity inference tasks. LOIS hides the source IP and the packet header's target service information, preventing on-path adversaries from separating the traffic directly from packet headers. Simultaneously, flow information is hidden. Thus, on-path adversaries cannot map packets into different flows according to packet headers. For the local adversaries, packets are contained in an inner layer of an encrypted 802.11 frame. They can group

the traffic for devices but cannot view more detailed information, such as the service type of the traffic. LOIS can also prevent cloud adversaries from separating packets. VPNs cannot defend against this attack.

(2) *Passive adversaries infer device type or user activities by collecting IoT traffic.* Since we modify packet headers to hide the requested service information for passive adversaries. Thus, they can not infer the device type of the traffic through the destination IPs and ports. Apthorpe *et al.* [29] claimed that passive adversaries could infer the device type through the traffic rate metadata using a simple supervised machine learning method. In this work, the traffic pattern is hidden by integrating the traffic analysis defense module. Therefore, all passive adversaries cannot infer the device type from the traffic rate metadata. For user activities, passive adversaries first needs to obtain the traffic related to the target device and then analyze the traffic pattern. As we have discussed that device information and traffic patterns are hidden by LOIS, adversaries cannot infer user activities through traffic patterns.

(4) *Passive adversaries try to recover encrypted fields in the packet header.* To recover the encrypted field, adversaries need to get the corresponding keystream for the packet. We use an identifier to transfer the keystream information between different entities. Adversaries cannot infer the target keystream from the identifier unless they get identifiers-to-keystreams mappings. Mappings are stored in the PK Vacuum table and are considered secret unless the cloud load balancer is compromised. One concern is the key reuse attack by which adversaries can further recover encrypted fields when they sense two packets use the same keystreams. However, we adopt PK Vacuum tables to avoid duplicate keystreams during a sufficiently long period of time to defend against

this attack.

4.8 Related work

IoT device classification and identification. Researchers classify and identify IoT devices by analyzing the IoT traffic [34, 101, 103, 113, 126, 131, 136]. Paper [131] develops a multi-stage machine learning based classification algorithm to identify specific IoT devices based on the IoT traffic from a smart environment containing 28 different IoT devices. Paper [34] presents a methodology to perform device behavioral fingerprinting that can further infer device type. Paper [113] introduces a probabilistic framework for device identification.

Traffic analysis attack and defense. The analysis of IoT traffic brings the possibility of attacks to IoT devices [2, 28, 51]. Paper [2] finds that IoT traffic rates can reveal potentially sensitive user interactions even when the traffic is encrypted, leading attackers to detect user behaviors. Paper [28] presents a user activity inference attack by which a passive adversary can infer user behaviors from analyzing traffic metadata. To defend against traffic analysis attack, paper [29] proposes four strategies to protect smart home privacy from passive adversaries. One strategy is to tunnel all smart home traffic through a virtual private network (VPN) to prevent device identification and user behavior inference attack. Another strategy is to apply traffic shaping to hide the traffic pattern. For example, the independent link padding algorithm (ILP) [63, 139] and the stochastic traffic padding (STP) algorithm [28] shape bidirectional traffic rates to a predetermined rate.

4.9 Summary

We present a keystream-based LOIS framework to protect user privacy by hiding IoT packet headers. LOIS includes the keystream management module and the packet header modification module. We implement the LOIS framework on commodity servers running in a public cloud. Results show that LOIS achieves a better throughput compared with IPsec, and brings small overhead for every packet. In addition, we implement LOIS-DI on a Raspberry Pi 3 to evaluate the computation overhead on LOIS-DI for bidirectional traffic. Results show that LOIS-DI incurs about 300 ns–365 ns latency on the device for one packet.

Chapter 5

HyperMerger: An Automatic Tool to Generate Consolidated Algorithms for Co-located Network Functions

Network functions (NFs), such as forwarding information bases (FIBs), load balancers, NAT, packet measurement, and firewalls, are running on heterogeneous network devices that could be special hardware middleboxes, programmable router-s/switches, or general-purpose computers. It has been a trend that one modern/future network device supports multiple network functions simultaneously. For example, e-merging programmable switch ASICs and data plane programming languages such as P4 enable implementing different network functions, such as forwarding, measurement, and load balacing, on one programmable switch [70, 99, 145]. Two challenges are widely known in designing algorithms and systems for these NFs: 1) limited memory resource; and 2) achieving high processing speed (preferably at line rate) [99, 127, 128, 160].

Recent studies have been focusing on designing memory-efficient and fast data structures and algorithms for individual network functions. For example, we have witnessed a line of research proposing efficient sketches for various measurement tasks [70, 93, 94, 145, 147]. In addition, another line of research applies space-compact data structures and algorithms for packet forwarding [54,67,87,107,127,128,146,149,159,160] and load balancing tasks [56, 99, 150]. These algorithms maintain various data structures, including variants of Bloom filters [35,67,107,146], Cuckoo hashing [99, 114, 160], Bloomier filters [44,129,149], and others [127,159]. According to our observation, none of these studies considers designing data structures and algorithms for multiple co-existed network functions, although supporting multiple network functions on the same device is widely accepted due to recent advances of programmable networks [93, 99, 145, 147]. We also notice that recent research work studies supporting multiple programmable data planes (PDP) programs on one programmable switch [14, 69, 153, 157]. However, these studies do not consider optimizing co-located NF algorithms based on their characteristics, leaving optimization space.

We argue that if we design consolidated algorithms, each of which serves multiple co-located NFs running on the same device, **performance improvements (such as higher throughput, smaller memory, or both) can be achieved without extra hardware cost**. Hence algorithmic efforts can gain performance for free! It is mainly because:

1. Data structures for different NFs share similar computation steps. One typical common operation of the above algorithms is computing of multiple universal hash functions. It has been reported that hash computations are the largest per-packet overhead that affects packet processing rates [93]. These hash functions must be independent within one algorithm to satisfy algorithm properties, but they do not need to be independent *across different NFs*. Therefore, we can reuse hash results for different NFs to reduce processing overheads.
2. Algorithms and data structures for different NFs can be co-located to reduce memory cost and/or reduce the number of memory accesses per packet, which is second-largest overhead of packet processing [93]. For example, it is possible to store the measurement counter and packet processing actions of the same flow together in one memory unit to reduce the number of random memory accesses as well as to avoid storing the flow ID repeatly.

The opportunities of performance gains by designing consolidated algorithms to support multiple co-located NFs bring us an important yet under-explored problem: **is there a way to automatically design consolidated data structures and algorithms that optimize multiple co-located NF algorithms?** Such a tool is important to lower the barrier of designing consolidated algorithms.

Hence, we present HyperMerger, the first automatic tool that takes multiple NF algorithms as inputs and generates a consolidated and resource-efficient data plane algorithm to serve these co-located NFs. The main idea of HyperMerger is to analyze the processing phases of the input algorithms, find the candidate phases that can be merged, and utilize dependency graphs to complete the merging tasks and ensure correctness.

Our current implementation of HyperMerger takes NF algorithms represented in P4 programs [37] as input and outputs P4 and C codes that can be run on hardware and software network devices. However, the methods in HyperMerger can be extended with data plane representations other than P4.

Our contributions can be summarized as follows:

1. We demonstrate that performance gains can be achieved by designing consolidated algorithms for co-located NFs, which do not increase hardware cost. This contribution is first introduced in the workshop version (referenced in the submission form visible to the PC chairs).

2. We are the first to design and implement an automatic tool to generate consolidated algorithms for co-located NFs. The outputs of HyperMerger can be used in both hardware and software platforms.

3. We conduct experiments on a rich set of co-located NF algorithms. The results show that the algorithms generated by HyperMerger can achieve one or more of the following advantages: reduce SRAM cost (by up to 70%), reduced hash calls (by 31% to 80%), reduced number of pipeline stages (by 2 to 6), and increased throughput (by $> 50\%$).

5.1 Consolidated NF Algorithms

5.1.1 Benefits of Consolidated Algorithms

Space-compact and fast data structures and algorithms have been widely used to achieve the task of various NFs. We have studied many data plane algorithms with their applications to different NFs. Most of the algorithms are variants of classic hash tables, Bloom filters [35, 39], Cuckoo hashing [114] and Cuckoo filters [59], dynamic minimal perfect hashing [43, 44, 127, 148], and sketches [42, 52, 94, 145]. Our observation is that many of these algorithms share similar computation steps and/or similar organizations of data structures. Hence there is a big space of optimization *on the algorithm level* if we design consolidated algorithms and data structures that serve multiple NFs, compared to first designing them individually and then running them as separate programs.

We summarize our observations as follows.

1) Algorithms and data structures for different NFs may share similar computation steps. A common and computation-intensive operation is that they need to compute multiple universal hash functions. These hash functions *should be independent within one NF* to satisfy properties of these algorithms. However, hash values *can be reused for algorithms of different NFs*, because hash functions of different NFs do not need to be independent. For example, one hash value of a flow can be used in a forwarding table to access the forwarding rule of the flow while also being used in a sketch to locate the counter of the flow.

Reducing hash computation time is crucial to resolve the performance bottleneck of many NF algorithms. Table 5.1 shows the percentage of CPU time for hash computation and memory access operations for different algorithms, profiled by the Intel Vtune Amplifier [12]. We use public traffic traces collected in Equinix-Chicago monitor from CAIDA [4] for the analysis, which contains 1.8 million flows and 5.6 million packets. Results show that hash computation plays a considerable time cost for these

	Hashing	Memory accesses
Bloom Filter [35]	28.6%	7.4%
Cuckoo Hashing [114]	12.9%	4.1%
Count-Min Sketch [52]	19.7%	16.7%
UnivMon [94]	45.5%	15.5%

Table 5.1: Percentage of CPU time for hash computations and memory accesses in different algorithms, profiled by Intel VTune [12]. They are the top-two types of overhead.

α (S_a/S_b)	Thrpt-separate	Thrpt-merged
1	44.61 Mops	90.97 Mops
2	46.62 Mops	87.69 Mops
5	40.26 Mops	70.93 Mops

Table 5.2: Throughput comparison of querying two separate tables with sizes S_a and S_b and the merged table. Mops: millions of operations per second.

data structures. Therefore, reusing results of hash computations for multiple NFs is a potential optimization to reduce the time complexity.

2) Algorithms for different NFs need to access different data structures according to the indices computed by hash functions. We can co-locate the fetched contents from these data structures into one memory unit to reduce the number of memory access per packet and/or reduce the total space cost, to further improve the performance.

Hash tables are the basic building blocks for many NF algorithms such as Cuckoo hashing and sketches. A specific value (forwarding port, flow size counter, etc.) is stored, updated, and queried in a slot of a hash table, which can be localized by computing a universal hash function with a key, e.g., certain fields of a packet header.

A consolidated data structure can co-locate individual values from multiple hash tables into one big slot and construct a merged table with these slots. Any obvious advantage is that fetching these values can be completed by one (or fewer) memory accesses, rather than accessing the slots of different tables. Let S_a be the size (number of slots) of Table a, S_b be the size of Table b, and $S_a > S_b$. An ideal case to merge the two tables is $S_a \approx \alpha S_b$ for an integer α, so that α slots from a can be co-located with 1 slot from b. One more observation of existing NF algorithms is that the size of a table can be flexible, as long as the memory cost is no larger than the budget and some properties are preserved, e.g., all values can be put into the table or the accuracy or false positive rate of a structure is below a threshold. Hence it is always possible to set table sizes to satisfy $S_a \approx \alpha S_b$. Table 5.2 shows the throughput improvement of querying a merged table compared to querying two tables separately by varying $\alpha = S_a/S_b$, on a server. In this set of experiments, we use 100 million 32-bit data that is randomly generated as the set of keys and store them into two Cuckoo hash tables. Results show that merging hash tables increases the query throughput by 60% to 100%, due to the reduced number of memory access.

5.1.2 Example of consolidated NF algorithms

We present a detailed consolidated NF algorithm, called CuckooCM, which serves the functions of a forwarding information base (FIB) using a Cuckoo hashing table [128,160] and a flow size estimator using a Count-Min Sketch (CM-Sketch) [52,147]. Both FIBs and flow size estimators are common network functions running on network devices such as routers and switches, serving the packet forwarding task and network

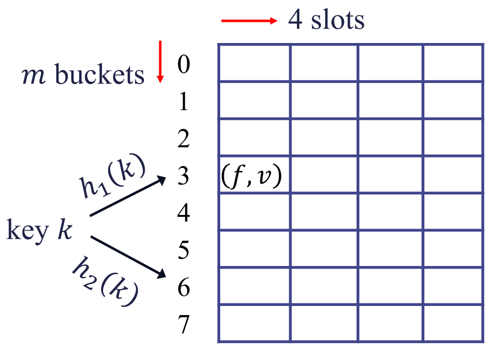


Figure 5.1: An example of a Cuckoo hash table

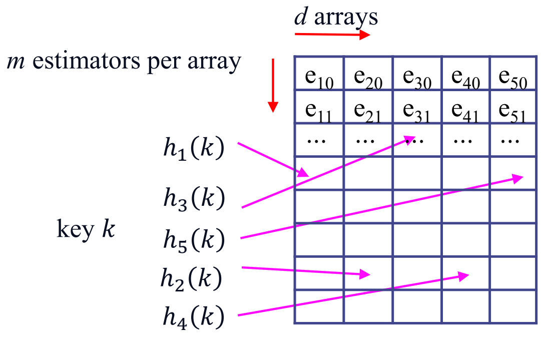


Figure 5.2: An example of a Count-Min sketch

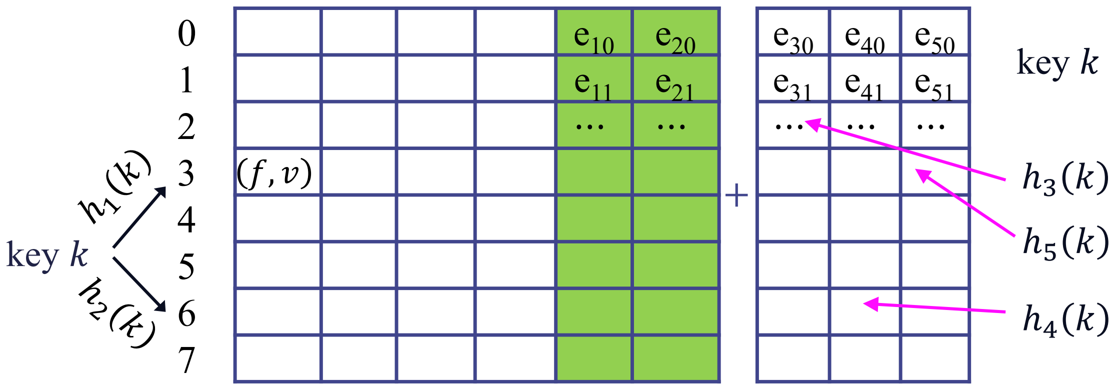


Figure 5.3: CuckooCM

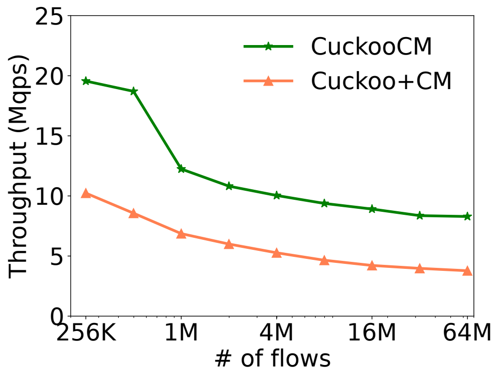
measurement task respectively.

To the ease of illustration, we use a simplified version of Cuckoo hashing based FIB compared to existing algorithms with similar ideas like CuckooSwitch [160] and Cuckoo Forwarder [128]. The data structure of the FIB is shown in Fig. 5.1. It is a table of m buckets and each bucket contains 4 slots. Given a packet, the key k can be the destination IP address for destination-based routing or flow ID (e.g., four tuples) for flow-based routing. The corresponding value v returned by the table by querying the key k is the index of the switch port to forward the packet. Here, each slot stores a tuple

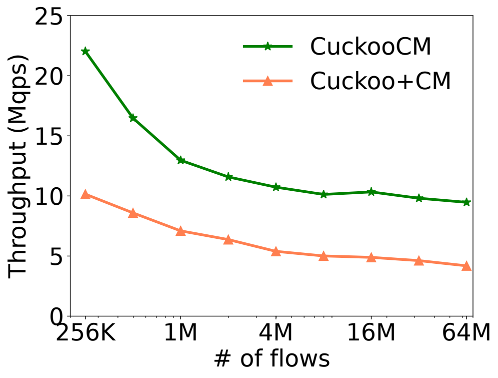
$< f, v >$, where f is the fingerprint of the key k by calculating $f = h(k)$ using a hash function h (different from h_1 and h_2 used later) and v is the index of the forwarding port. Compared to directly storing $< k, v >$, using the fingerprint f usually reduces much space cost. For example, a flow ID represented by 4-tuple needs 96 bits but a fingerprint can be as short as 20 bits. Each flow ID k has two candidate buckets by computing $h_1(k)$ and $h_2(k)$ and the tuple $< f, v >$ is stored in one of the two buckets. When querying k, the two buckets will be fetched and $< f, v >$ can be found from them.

The data structure of the CM-Sketch is shown in Fig. 5.2, which can be used in various packet measurement tasks, such as flow size estimation and heavy hitter detection. The CM-Sketch consists of m arrays, each array has d estimators. For each incoming packet, the flow ID (4 tuples) is used as the key k. The CM-Sketch updates one estimator in each array computed by a universal hash function $h_i(k)$, $i \in [1, d]$. To query the sketch with k, e.g., getting the estimated size of the flow, all d estimators of k are read from the table, one from each array. The d estimators are then processed, e.g., by computing the minimum, to get the final query result.

We present the consolidated data structure and algorithm, called CuckooCM, which is designed to serve both FIB and packet measurement tasks by combining the above two algorithms. The data structure of CuckooCM is shown in Fig. 5.3. For each bucket b of the table, $b \in [0, m]$, it also stores the two estimators e_{b1} and e_{b2}. In this way, when a packet of flow ID k is processed, by computing $h_1(k)$ and $h_2(k)$, these two hash values can be reused for both the FIB lookup and updates of two estimators, because the two estimators of k are stored in the two buckets respectively. If $d > 2$, the remaining estimator can be stored in another table, which does not affect the correctness



(a) Uniform traffic



(b) Zipfian traffic

Figure 5.4: Throughput comparison

or accuracy of the sketch.

We show the performance improvement of CuckooCM compared to running the Cuckoo hash table and CM-Sketch as separate programs using experiments. Fig. 5.4 shows the throughput of CuckooCM and running the two algorithm separately on a software switch, using generated network traffic with uniform and Zipfian distributions respectively. We use 5 arrays for the CM-Sketch. Result show that CuckooCM achieves about 2x throughput compared to running the two algorithms separately with both traffic distributions. The memory costs are the same and accuracy of the CM-Sketch does not change.

We discuss another case of consolidation. Suppose two NFs use the same data structures, such as Cuckoo hash tables. Each table needs to store the keys and the values and the NFs have the same type of keys but different types of values. For example, a layer-4 load balancer [56] and a FIB both use 5-tuples as the keys but one returns the internal destination IP and the other returns the port number to forward. We run a set of experiments on a hardware Tofino programmable switch. We create $m \in \{1, 2, 3\}$

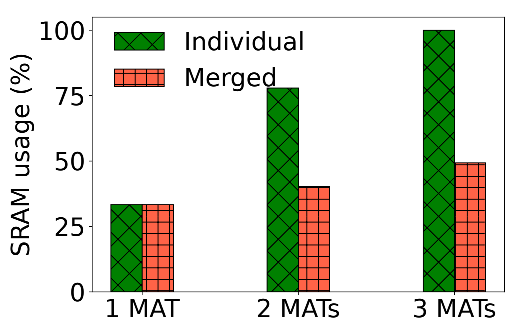
Cuckoo hash tables that use the same type of keys (48 bits each) but return different types values (16 bits each). We run them separately compared to running a consolidated table by merging them. Fig. 5.5(a) shows the SRAM cost reported by the Tofino switch. We find that the consolidated algorithm can save around 30% SRAM by merging two tables or 50% SRAM by merging three. Fig. 5.5(b) shows the number stages of packet pipeline of running them separately and running a consolidated one. A consolidated algorithm can also evidently reduce the number of stages and hence reduce the packet processing time.

The above results show that a consolidated NF algorithm can improve packet processing throughput without spending more memory cost, possibly even reducing the memory cost. Hence **consolidated NF algorithms can gain extra performance without extra hardware cost, i.e., for free**.

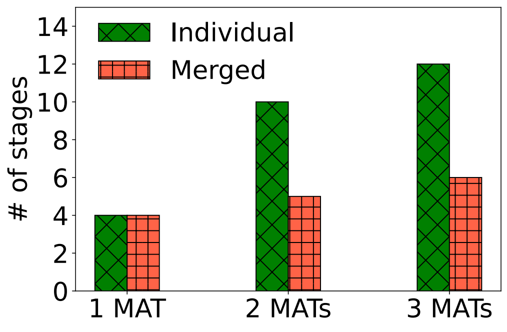
	keys	benefit	possible algorithms
FIB+ flow-size sketch	flow ID	faster processing	Ludo+cSketch or ElasticSketch [145]
FIB+ flow-spread sketch	destination	faster processing	(Ludo + cSketch) or (Buffalo [146] + bSketch [161])
LB+NAT	five-tuple	smaller memory, faster processing	Maglev [56] or SilkRoad [99] + NAT
LB+sketches	five-tuple	faster processing	Maglev [56] or SilkRoad [99] + cSketch
FIB+LB+NAT	five-tuple	smaller memory, faster processing	all use hash tables

Table 5.3: Examples of possible consolidation of NF algorithms. LB: load balancer; NAT: network address translation;

We list other combination of NFs to gain extra performance or resource efficiency in Table 5.3. For example, a FIB can be merged with a network address translator (NAT), which both use the flow ID as the key and hash tables as their data structures. Such combination can reduce a significant amount of memory and lookup time. A layer-4 load balancer [56, 99, 129] can be merged with a FIB or NAT if they all use the 5-tuple



(a) SRAM usage



(b) # of stages

Figure 5.5: Resource savings on Tofino

of a packet as the key. Some FIBs use the destination instead of 4-tuple or 5-tuple as the key, and they can be combined with flow-spread sketches [161], e.g., those to count how many flows , to improve the throughput.

5.2 Background of Automatic NF Consolidation

Designing and implementing consolidated NF algorithms require the network operator to be familiar with the details of individual NF algorithms and have strong analytical skills to find out possible consolidating strategies. In addition, manually designing consolidated algorithms for individual network devices take time even for skilled network operators. Hence it is highly desired to have an automatic tool to produce a consolidated algorithm by inputting a number of individual NF algorithms. The output should serve all functions of the original NFs while providing possible resource saving or performance gain.

This work designs and implements HyperMerger, a tool to automatically merge co-existed NF algorithms into a consolidated one. HyperMerger utilizes the framework of P4 [37] to represent data plane algorithms. P4 is a programmable language to express how packets are to be processed in a switch and how a switch is to be configured. A P4 program can be run on specific hardware switches or software switches. Note that *the main idea and method of HyperMerger do not rely on P4 and can be built on other data plane representations.* Using P4 is to simplify the implementation of HyperMerger and make it easier for the networking community to understand and use HyperMerger if they have existing experience of P4.

We briefly introduce the concepts of P4 that will be used in HyperMerger. A P4 program is a representation of a data-plane algorithm, which mainly contains the following key components:

Headers. The definition of a header describes the structure of a list of packet header fields. Example headers can be Ethernet headers, IP headers, TCP headers, and UDP headers.

Parsers. The definition of a parser shows how to identify headers within a packet, and specifies valid header sequences. For example, a parser `parse_IP` can be defined as first extracting the IP header, then moving to the parser `parse_TCP` if the extracted 'protocol' field is 6 or moving to the parser `parse_UDP` parser if the extracted 'protocol' field is 17.

Registers. A set of registers are a data structure to store data or states and allow the data to be queried. For example, the data structure of a sketch can be implemented by registers.

Match-action Tables (MATs). MATs are special mechanisms in the P4 program to specify how to perform packet processing. Different from tables that are commonly used in algorithms, such as hash tables, a MAT specifies conditions denoted by using different match fields, and corresponding actions it may execute when meeting the conditions. Examples of actions can be hash computations and returning queried values from a structure with certain match fields. MATs with no matching field are called default-only MATs, such as hashing. MATs returning queried values from a data structure by matching certain indices are called query MATs. For example, querying a hashing table includes two MATs: the first MAT is to compute the hash function, which is a default-only MAT, and the second one is to return the value from the registers by matching the computed hash value, which is a query MAT.

Actions. P4 can support a number of actions from simple protocol-independent primitives. Actions are accessible through MATs.

Control logic. The control logic in a P4 program defines the execution order of MATs to guide how to process a packet. For example, the control logic of a CM-Sketch with five arrays needs to assign MATs to perform five hash computations followed by 5 MATs to access 5 estimators according to the hash indices.

P4 compilers use a two-layer model including a front-end compiler and a back-end compiler. The front-end compiler generates an intermediate representation (IR) to represent the original P4 program, which could be a JSON file (.json). In the JSON IR file, a P4 program can be represented by two directed acyclic graphs (DAGs): a parser graph and a phase dependency graph (PDG).[1] In a parser graph, vertices are parsers, each of which could be parsing a certain type of packet headers, and edges are conditions that transfer from one parser to another parser. For a PDG, a vertex (phase)

[1]Some documents use “table dependency graph” and each vertex may not be a real table but a processing phase. Hence we use ‘phase’ instead of ‘table’ to avoid confusion for readers with no experience of P4.

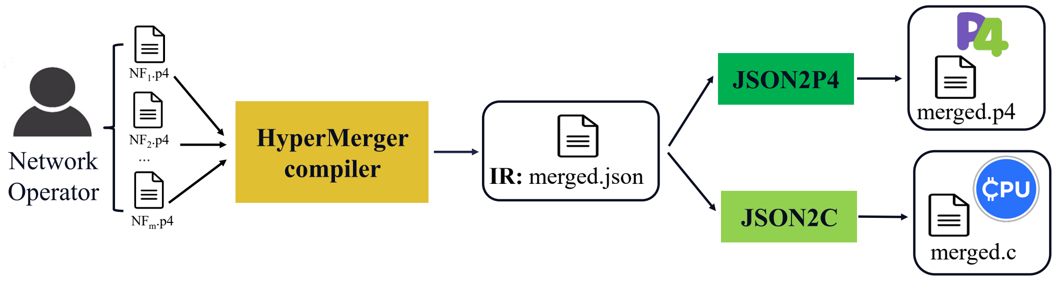


Figure 5.6: The workflow of HyperMerger

is a MAT, denoting a phase of processing the packet. An edge shows the dependency relation between two MATs that specifies how to move from one MAT to another MAT by performing certain actions.

Insights. The IR of a P4 program uses two graphs to represent data plane data structures and algorithms. Hence consolidating NF algorithms is equivalent to merge the graphs from individual algorithms into two unified graphs. During the merging process, if multiple algorithms share same vertices in their graphs, these vertices can be merged into one if no violation of dependency occurs.

5.3 Design of HyperMerger

We present the detailed design of HyperMerger that automatically generates a consolidated algorithm from multiple separate NF algorithms.

5.3.1 Overview

Fig. 5.6 shows the components and the workflow of HyperMerger. The inputs of HyperMerger are multiple P4 programs for different NFs, denoted as $\texttt{NF}_1\texttt{.p4}$, $\texttt{NF}_2\texttt{.p4}$, ..., $\texttt{NF}_\texttt{m}\texttt{.p4}$ respectively. The output is a representation of the consolidated algorithm in either P4 or C. Our current implementation support both languages and it is possible

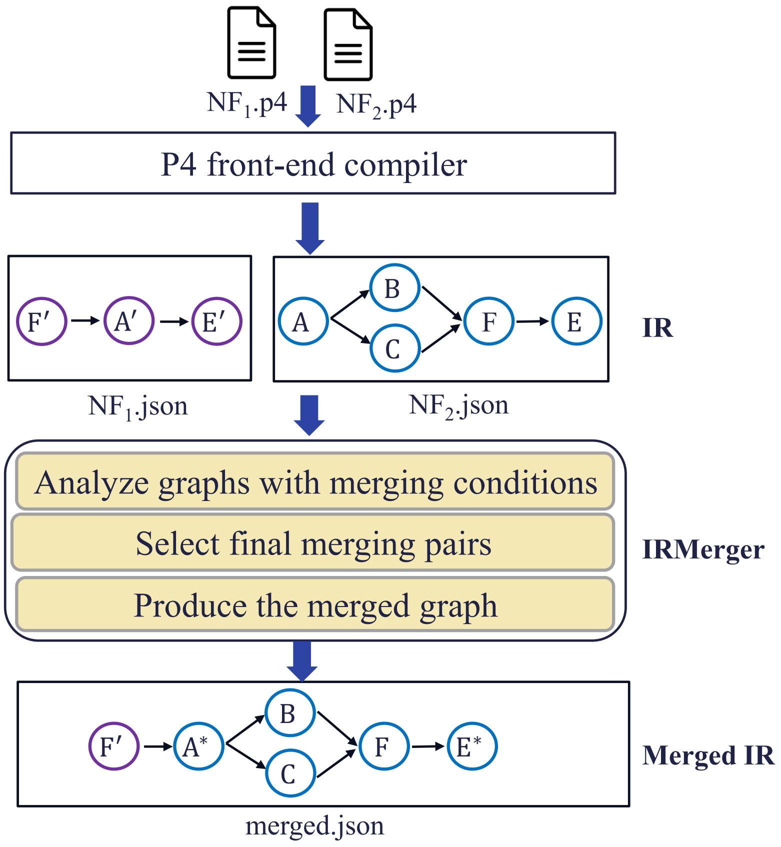


Figure 5.7: The workflow of the HyperMerger compiler. (A, A'), (F, F'), (E, E') are candidate merging pairs. HyperMerger finally merges A and A' to A^* and E and E' to E^*.

to extend the support to other languages such as Python. HyperMerger automatically outputs a P4 program `merged.p4` and a C program `merged.c`. The output P4 program can be directly run on a hardware or software target that supports P4. The output C code can be run on arbitrary software platforms.

Our design includes three main components: a HyperMerger compiler, a JSON to P4 translator (JSON2P4), and a JSON to C translator (JSON2C). Network operator provides HyperMerger individual P4 programs of the NFs they want to run on the network device. Then, the HyperMerger compiler analyzes these programs and generates one unified intermediate representation (IR), which is a JSON file `merged.json`. Recall that the JSON file represents two graphs: (1) a parser graph that specifies the sequence of extracting packet header fields and (2) a phase dependency graph (PDG) that specifies the sequence of match-actions of processing a packet represented by MATs. Depending on which language the network device (e.g., hardware or software switch) supports, HyperMerger translates the IR `merged.json` to a P4 program `merged.p4` by JSON2P4 or C program `merged.c` by JSON2C.

Fig. 5.7 shows the workflow of the HyperMerger compiler. The HyperMerger compiler first utilizes the P4 front-end compiler to generate an IR $\texttt{NF}_\texttt{i}\texttt{.json}$ of every P4 program $\texttt{NF}_\texttt{i}\texttt{.p4}$. We design a function called IRMerger to process the IRs of all separate NF algorithms and generate a consolidated IR `merged.json` containing the operations of all these algorithms. As depicted in Fig. 5.7, HyperMerger uses the P4 front-end compiler to convert the P4 programs to the IRs $\texttt{NF}_\texttt{1}\texttt{.json}$ and $\texttt{NF}_\texttt{2}\texttt{.json}$. in which each PDG includes a number of MATs as its vertices and the dependencies among MATs as the directed edges. Each JSON file is abstracted as a PDG and the parser graph is omitted here due to space limit. In the two PDGs, there are three pair of MATs that can be potentially merged: (A, A'), (F, F'), and (E, E'). Applying the dependency constraints, (A, A') and (E, E') can be merged but (F, F') cannot – F' should be processed before A' but F is after A. After processing by IRMerger, the consolidated IR `merged.json` is abstracted as a new PDG shown in the bottom of Fig. 5.7, where A and A' are merged to A^* and E and E' are merged to E^*.

IRMerger includes (1) a module to analyze the input parser graphs and PDGs with pre-defined merging conditions and find candidate merging pairs (e.g., the three pairs in the example); (2) a module to select final merging pairs (e.g., (A, A') and (E,

E')) and exclude those violating the dependency constraints; (3) a module to produce merged graphs based on the selected merging pairs. Then IRMerger generates the consolidated IR `merged.json` as the output of HyperMerger compiler and the input of JSON2P4 or JSON2C. We detail these modules in the following subsections. Note the discussion below assumes only two NF algorithm to consolidate. **For more than two algorithms, consolidation can be done sequentially.**

5.3.2 Analyze graphs with merging conditions

This section introduces merging conditions applied to parser graphs and PDGs to the input IRs, based on our insights from the characteristics of multiple NF algorithms. We begin with giving detailed definitions and descriptions of parser graphs and PDGs.

A parser graph $G_p = (S, E_p)$ is a directed acyclic graph (DAG), where $S = \{s_1, s_2, ..., s_m\}$ denotes all the packet-header parsers in the P4 program and the edges $E_p = \{(s_i, s_j, c_{ij}), (s_i, s_j \in S)\}$ shows the packet goes from the parser s_i to the parser s_j when meeting the condition c_{ij}.

A PDG $G = (T, E)$ is also a DAG, where $T = \{t_1, t_2, ..., t_n\}$ represents the packet processing phases (MATs) and and $E = \{(t_i, t_j), (t_i, t_j \in T)\}$ represents the dependency between two MATs: i.e., a packet should be processed by one MAT t_i and then move to another MAT t_j. A vertex $t (t \in T)$ includes the following attributes to represent a MAT.

• `t.pid`: the ID of the MAT.

• `t.name`: the name of the MAT.

- `t.match_field`: the match key of the MAT. Default-only MATs such as hash computations have no match key.
- `t.match_type`: the type of this matching, e.g., exact or prefix.
- `t.actions`: the actions of the MAT.
- `t.size`: the max number of entries of the MAT.
- `t.tag`: 0 or 1 to indicate whether this MAT is default-only.

An edge $e = (t_i, t_j)$ of the PDG contains the corresponding action information that makes a packet traverse from t_i to t_j. Given the PDG G, we can compute a *dependency matrix* D for G: $D[t_i, t_j] = 1$ if there is a directed path from t_i to t_j, and $D[t_i, t_j] = 0$ otherwise.

We introduce three merging conditions to analyze the parser graphs and PDGs.

Condition 1: Two equivalent parsers are selected.

This condition is to avoid duplicate extractions of packet header fields by different NFs. Two parsers s_i and s_j from two parser graphs are considered equivalent if they meet three requirements: 1) The fields extracted from the packet headers are the same; 2) Their match fields are the same; 3) Their next parsers should have the same matching fields.

Condition 2: Default-only MATs triggering equivalent hash computations from different NFs are selected.

Recall that a default-only MAT is just an action without matching. This condition is to reuse hash computations among different NF algorithms. Two hash computations are equivalent if they use the same fields as the input keys and their hash

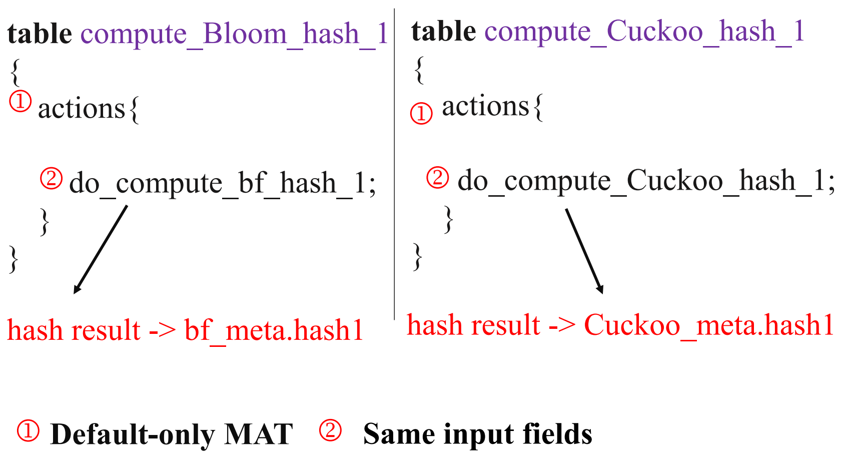


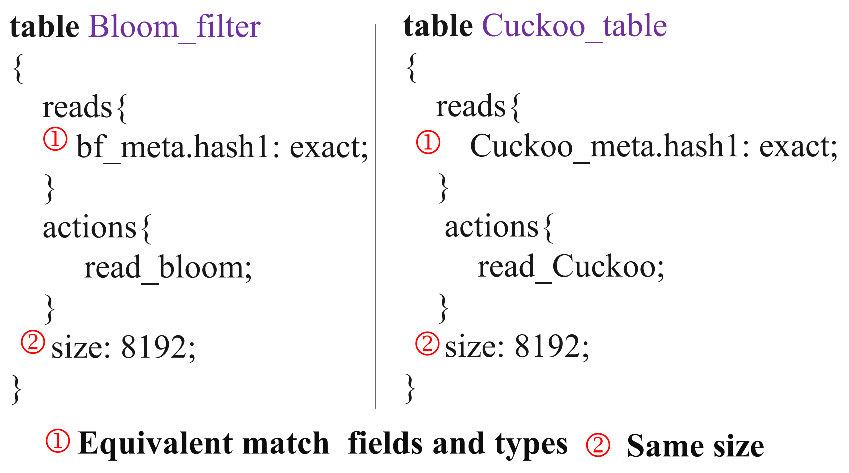
table compute_Bloom_hash_1
{
① actions{
② do_compute_bf_hash_1;
}
}
hash result -> bf_meta.hash1
table compute_Cuckoo_hash_1
{
① actions{
② do_compute_Cuckoo_hash_1;
}
}
hash result -> Cuckoo_meta.hash1
① Default-only MAT ② Same input fields

Figure 5.8: Two MATs that meets Condition 2

functions are in an equivalent group. For example, we can put different uniform hash functions that produce results no shorter than 32 bit in a '32b' equivalent group. Fig. 5.8 shows an example including a MAT `compute_Bloom_hash_1` from the algorithm Bloom filter [35] and a MAT `compute_Cuckoo_hash_1` from the algorithm Cuckoo hashing [114]. By detecting that both MATs are default-only MATs and `do_compute_bf_hash_1` and `do_compute_cuckoo_hash_1` have same fields as the keys, the two MATs are selected as a candidate merging pair.

Condition 3: Query MATs with the same match fields are selected.

Recall that a query MAT returns queried values from a data structure by matching certain fields or indices computed by hash functions. This condition is to merge two similar data structures from different NF algorithms to reduce the total number of memory accesses. In particular, merging query MATs on the P4 platform can save the memory cost, because matching fields are reused after merging. Two query MATs $t_i \in T_1$ and $t_j \in T_2$ can be merged if their match fields are the same: $\mathtt{t_i.match\_field} = \mathtt{t_j.match\_field}$, e.g., they both use 4-tuple, and their match types



```
table Bloom_filter
{
   reads{
   ① bf_meta.hash1: exact;
   }
   actions{
       read_bloom;
   }
 ②size: 8192;
}
```

```
table Cuckoo_table
{
   reads{
 ①   Cuckoo_meta.hash1: exact;
   }
   actions{
       read_Cuckoo;
   }
 ②size: 8192;
}
```

Figure 5.9: Two MATs that meets Condition 3

are the same: $\mathtt{t_i.match\_type} = \mathtt{t_j.match\_type}$, e.g., they both use exact matching. Additional processing is needed to align two tables. Let S_a be the size (number of slots) of Array a, S_b be the size of Array b, and $S_a > S_b$. $(\alpha - 1)S_b < S_a \leq \alpha S_b$ for an integer α. α slots from a should be co-located with one slot from b. An example is shown in Fig. 5.8 where a Bloom filter and a Cuckoo table both use arrays to store data. They use the same match fields and have the same size. Hence their slots can be easily merged.

Condition 4: Parallel MATs are not selected. If two MATs t_i, t_j in a PDG G satisfy that t_i is ordered before t_j in a valid topological order of G and there is no path from t_i to t_j, they are called parallel MATs. If a MAT is parallel with any other MAT, it will be not selected to any candidate pair. For example B and C in Fig. 5.7 are parallel MATs hence none of them should be merged with other MATs.

After the selection using the three conditions, HyperMerger obtains a few pairs of parsers and MATs as candidate pairs. Not all of them can be merged, hence further analysis based on constraints such as dependencies is necessary.

5.3.3 Select final merging pairs

HyperMerger selects final merging pairs from the candidate pairs by applying some constraints. We do not need to apply any constraint on the selected pairs of parsers because the dependencies have been considered in Condition 1 of analyzing parsers. For candidate MAT pairs, we need to select those satisfying the following requirement.

Recall D is a dependency matrix defined in Sec. 5.3.2: $D[x, y] = 1$ if and only if there exist a directed path connected by dependency edges from x to y. The following property must hold for D_1 and D_2, which are the dependency matrices of G_1 and G_2 respectively:

$$D_1[t_i, t_j] \cdot D_2[t'_j, t'_i] = 0, \qquad \forall t_i, t_j \in T_m$$

where (t_i, t'_i) and (t_j, t'_j) are two candidate pairs and $t_i, t_j \in T_1$ and $t'_i, t'_j \in T_2$. This property indicates that there should be no dependency loop in the merged graph.

For example, in Fig. 5.7 merging two candidate pairs $(E,\ E')$ and $(F,\ F')$ will violate the property and introduce a loop in the merged graph. Since we cannot merging all pairs, which pairs of them shall IRMerger select? The proposed selection algorithm of IRMerger is based on two intuitive ideas. 1) It maximizes the number of selected final pairs, because typically more selected pairs represents more resources to save, either on hash computations or memory cost. 2) It prefers to select "heavier" MATs, which have more number of entries or larger sizes of match fields. For example, merging two equivalent MATs with 2048 entries saves more space costs than merging two equivalent MATs with 1024 entries.

We design a hierarchical selection algorithm in IRMerger to select final merging

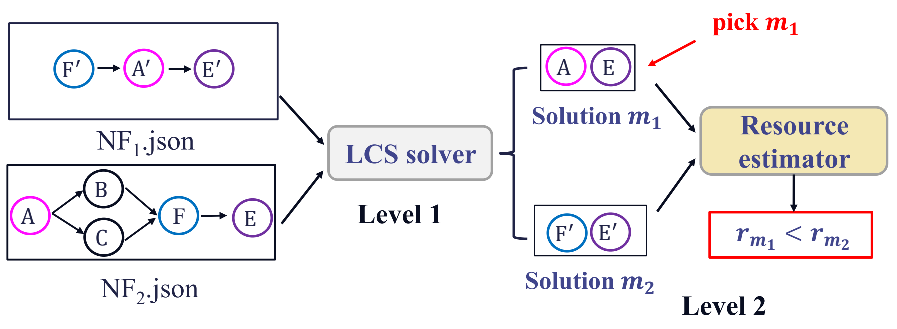


Figure 5.10: Hierarchical MATs selection

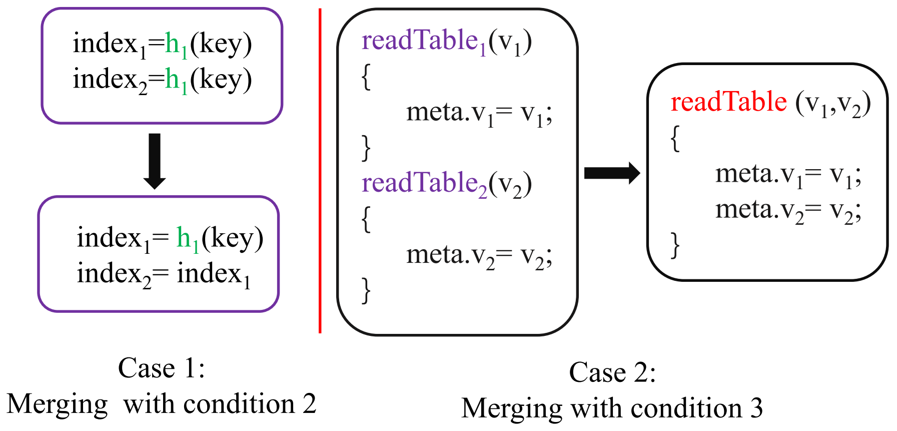


Figure 5.11: Merging operations for different cases

nodes as shown in Fig. 5.10. At the first level, we utilize a longest common subsequence (LCS) algorithm of dynamic programming to find the maximum number of merged nodes. Since the input candidate pairs would not be with a large number, dynamic programming can solve it in a short time [3, 48]. The output of the LCS algorithm can be multiple sets with the longest length of merged MATs. We then introduce the second level design to choose a desired set when there are multiple sets of LCS solutions.

The second level is a resource estimator to estimate which set of merged MATs can result in small resource cost among the outputs of the first level. For a MAT t_1 with n_1 entries, w_1 bits for each action, and m_1 bits for the match field, we use $n_1 * (w_1 + m_1)$

bits to approximate its memory cost. With the estimation of resource usage of MATs, HyperMerger can compute the required memory cost for each set of LCS output. Then HyperMerger chooses the LCS with minimum resource usage as the MATs to merge. In Fig. 5.10, two sets of LCS outputs are produced from level one: m_1 including A and E and m_2 including F' and E'. Since $r_{m_1} < r_{m_2}$, HyperMerger chooses to merge (A, A') pair and (E, E') pair.

5.3.4 Produce the merged graph

We present the algorithm to produce the merged graphs including the parser graph and PDGs.

For two PDGs $G_1 = (T_1, E_1)$ and $G_2 = (T_2, E_2)$, we merge G_1 and G_2 to build a merged graph $G_m = (T_m, E_m)$. G_m is first initialized to G_2. Then we merge G_1 to G_m. Let the set of the final merging pairs output by the previous step be F. Vertices in G_1 are divided into two disjoint sets: S including all vertices (MATs) that are included in F and $W = T_1 - S$. HyperMerger finds a valid topological order O_1 of T_1. The vertices in S separate O_1 into multiple segments. For example, in Fig. 5.6, a valid topological order of the PDG in $\texttt{NF}_1\texttt{,json}$ is $A - B - C - F - E$ and $S = \{A, E\}$. Hence it has only one segment: the one between A and E.

Each pair $(t_i, t'_i) \in F$ is merged as a new MAT t^*_i. Fig. 5.11 shows examples of merging two MATs when they meet condition 2 or condition 3. Then IRMerger adds the associated edges of t_i in G_1 to t^*_i. Every segment between two vertices t_i and t_j in G_2 ($t_i \in S$ and $t_j \in S$) needs to be added to the segment between t'_i and t'_j in G_2 to form a longer segment. In Fig. 5.6, the segment $B - C - F$ between A and E should be added to the segment between A' and E' – no vertex in this case. Note in this step we must check if there is any data dependency among the MATs in the two segments. One typical data dependency is write-after-read. Assume vertex $t_i \in T_1$ performs the read operation on a field extracted from the packet header, and vertex $t'_j \in T_2$ performs the write operation on the same field, we need to put t_i before t'_j in G_m to maintain the correctness. If there is no data dependency in the two segments, IRMerger directly links the segment of G_1 after that of G_2. Otherwise, we find vertices that cause data dependencies and generate a topological order of vertices of the two segments in a way to preserve the dependencies. The final merged graph is then generated from these merged MATs and the segments, stored in `merged.json`.

Merge parser graphs. IRMerger merges the parser graphs using the similar steps as those of merging PDGs.

Correctness. After merging, the following two properties hold. D_1, D_2, and D_2 are the dependency matrices of G_1, G_2, and G_m respectively.

- **Dependency preserving**: If u and v are a pair of merging nodes and are merged to a new node t, we have:

$$\begin{cases} D_m[u,a] = D_1[t,a], & \forall a \in T_1 \\ D_m[b,u] = D_1[b,t], & \forall b \in T_1 \\ D_m[v,c] = D_2[t,c], & \forall c \in T_2 \\ D_m[d,v] = D_2[d,t], & \forall d \in T_2 \end{cases}$$

  which means all dependencies related to u and v must be preserved by t.

- **Loop free**.

$$D_m[t_i, t_j] \cdot D_m[t_j, t_i] = 0, \qquad \forall t_i, t_j \in T_m$$

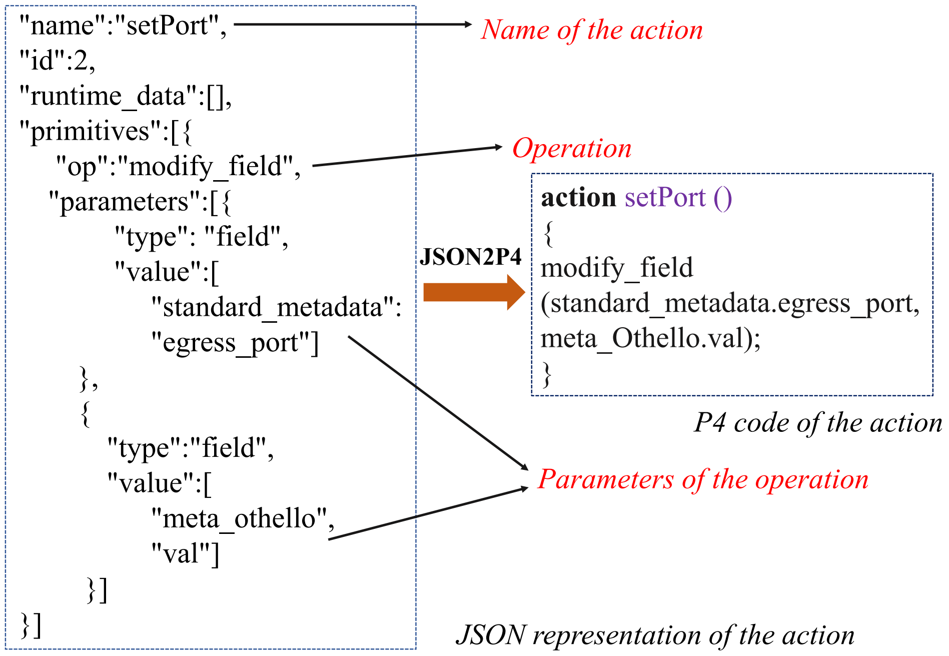
Merge multiple NF algorithms. Multiple NF algorithms can be merged sequentially to a consolidated one. The merging order has little influence to the eventual consolidated algorithm. To reduce the time cost of the merging process (not the time cost of NF algorithms), we order the NF algorithms by the descending order of the number of tables in their PDGs and then merge algorithms one by one.

5.3.5 Translate IR to P4 and C

The intermediate representation (IR) `merged.json` can be translated to code to run on network devices that support this type of code. Our current implementation includes two translators: JSON2P4 that translates the IR to a P4 program `merged.p4` running by P4 hardware or software switches and JSON2C that translates the IR to C code `merged.c` running by software network functions that support C. P4Merger is not limited to P4 and C and can easily be extended to support other languages and network devices.

JSON2P4

Since the IR `merged.json` follows the standard JSON structures and interpretation of P4 programs specified by the P4 specification document, we can directly translate the IR into its P4 code according to the interpretation. Fig. 5.12 shows an example of translating an action called setPort from the JSON representation to P4 code. JSON2P4 simply scans the fields with key words in JSON, such as `name`, `op`, and `value`, and translates the contents directly to P4 code.



```
"name":"setPort",
"id":2,
"runtime_data":[],
"primitives":[{
   "op":"modify_field",
   "parameters":[{
        "type": "field",
        "value":[
            "standard_metadata":
            "egress_port"]
      },
      {
         "type":"field",
         "value":[
            "meta_othello",
            "val"]
      }]
}]
```

```
action setPort ()
{
modify_field
(standard_metadata.egress_port,
meta_Othello.val);
}
```

Name of the action
Operation
JSON2P4
P4 code of the action
Parameters of the operation
JSON representation of the action

Figure 5.12: An example of the JSON2P4 translation

JSON2C

We use JSON2C to show the method that translates the IR `merged.json` to code in a general-purpose language C. If the network operator needs to use other languages such as Python, the translator can be design in the similar way.

The main steps of JSON2C is to translate each component in `merged.json` to the corresponding component in C, as follows.

Headers. Header types in the JSON file can be easily represented by structs in C. Each header field is mapped to a struct member, with bit fields to specify its type and size. Then headers and metadata are modeled as an instance of the corresponding struct, serving as global variables.

Parsers. Parsers are modeled as multiple functions in C to guide the extraction of contents from packet headers.

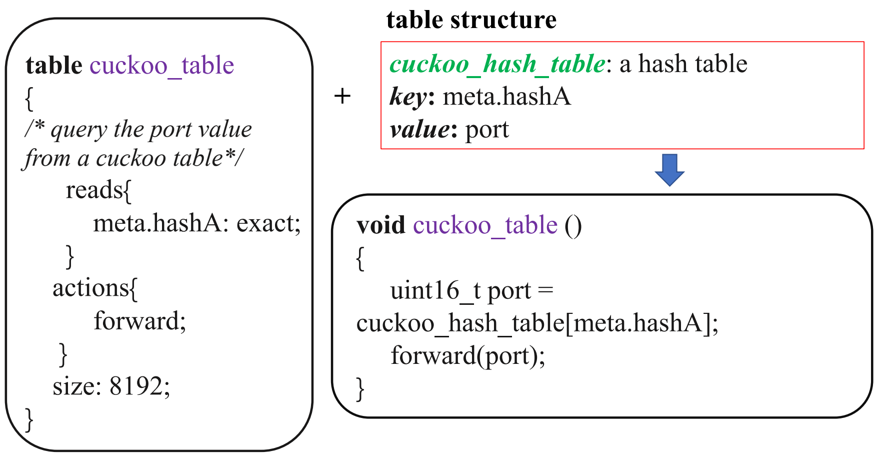


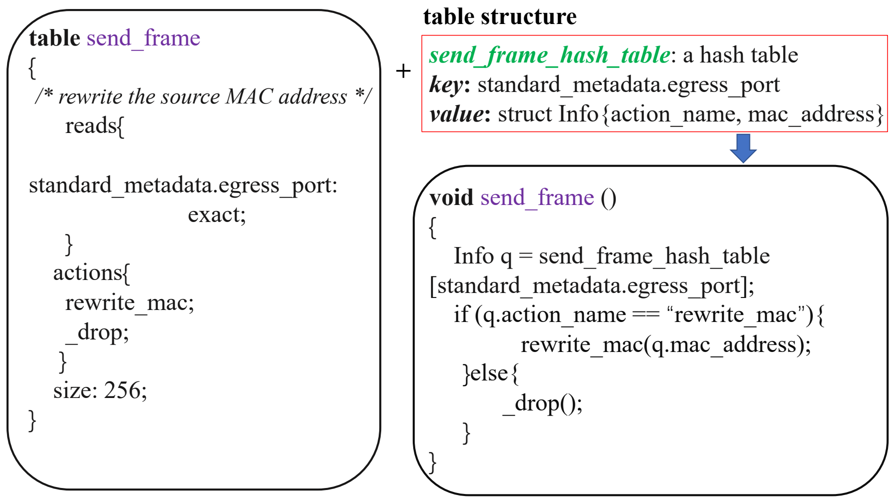
Figure 5.13: Translate a MAT to C: lookup a Cuckoo hash table

Registers. Registers are modeled as arrays in C. Operations on registers are then translated into read and write operations to the arrays.

Actions. Each action is modeled as a function in C. Operations contained in actions are then translated into corresponding C representations.

MATs. MATs are modeled as functions in C. For default-only MATs, JSON2C directly call the function of the action. For other MATs, JSON2C require the operator to provide *table structures*. A table structure defines the how the table is organized. Fig. 5.13 shows an example, where the operator provides a table structure whose name is `cuckoo_hash_table`, the lookup index is *meta_hashA* and the returned value is *cuckoo_v*. In P4, this table structure is provided by the control plane and for other languages the operator is also necessary to provide it. Fig. 5.13 shows an example of translating a MAT that serves to lookup a Cuckoo hash table to C code. Fig. 5.14 shows an examples of translating a MAT that sends an Ethernet frame by writing the source MAC address to the frame to C code.

Hash modules. JSON2C provides a set of hash functions of P4 and C that


```
table send_frame
{
 /* rewrite the source MAC address */
    reads{

standard_metadata.egress_port:
                    exact;
    }
   actions{
    rewrite_mac;
    _drop;
   }
   size: 256;
}
```
+
table structure

send_frame_hash_table: a hash table
key: standard_metadata.egress_port
value: struct Info{action_name, mac_address}

```
void send_frame ()
{
   Info q = send_frame_hash_table
[standard_metadata.egress_port];
   if (q.action_name == "rewrite_mac"){
           rewrite_mac(q.mac_address);
    }else{
         _drop();
    }
}
```

Figure 5.14: Translate a MAT to C: send an Ethernet frame after writing the source MAC address

are considered equivalent: e.g., they are all uniform and generate results no shorter than 32 bits. Hash functions in a JSON file can be directly translated to an equivalent hash function in C code.

5.4 Evaluation

The purposes of this section are two-fold: 1) we use the implemented HyperMerger program to demonstrate that it can automatically and effectively consolidate multiple NFs, and 2) we test the consolidated algorithms on both a hardware device (a Tofino switch) and a general-purpose server to show the advantages of consolidated NF algorithms.

5.4.1 Experimental Setup

Hardware. We evaluate the consolidated algorithms generated by HyperMerger with JSON2P4 on a testbed with an EdgeCore 100BF Tofino-based programmable switch connected to two servers with 100Gb Ethernet. We use the Tofino SDE version of 9.6.0 in our experiments. The consolidated algorithms translated by JSON2C are run on a general-purpose server with Intel i7-7700, 3.60GHz and 8 MB L3 Cache.

Algorithms for network functions. We list a number of algorithms for various network functions in Table 5.4. We generate five consolidated algorithms using HyperMerger by making various combinations of these algorithms, as described in Table 5.5. They are labeled with Case 1 to Case 5. Each case serves multiple NFs as listed in Table 5.5. In addition, Cases 4 and 5 include Buffalo [146] that uses multiple sets of Bloom filters for FIBs: each BF for each port. Hence the total numbers of algorithms are 7 for Case 4 and 9 for Case 5.

Comparison methods. We compare the algorithms generated by HyperMerger with the following two methods. 1) Vanilla, which is simply running multiple P4 programs simultaneously. 2) P4Visor [157], which merges multiple P4 programs into one program using code merging techniques, but it does not consider any optimization on algorithms and data structures.

Traffic workload. We use CAIDA backbone traces collected from the 'equinix-chicago' high-speed monitor [4] as the workloads. We use four different traces labeled with Trace 1 to Trace 4, whose characteristics are introduced in Table 5.6.

5.4.2 Evaluation metrics

We care about three main types of metrics of the generated consolidated algorithms: resource cost on the hardware switch, processing throughput of algorithms, and accuracy of the sketches. Each of them further includes multiple details performance metrics.

Network function	**Algorithms & data structures**
Forwarding (FIB)	Cuckoo hash table (FIB-Cu) [160]
	Concise (FIB-Co) [149]
	Buffalo (FIB-Buf) [146]
Firewall (FW)	Bloom filters (WF-BF) [39]
NAT	Cuckoo hash table (NAT-Cu)
	Concise (NAT-Co) [149]
Load balancer (LB)	Cuckoo hash table (LB-Cu) [99]
	Concury (Co) [129]
Packet measurement	Count-Min sketch (CM) [52]
	Count sketch (CS) [42]
	HyperLogLog (HLL) [62]
	MRB [58]
	MRAC [82]
	UnivMon [94]

Table 5.4: Algorithms and data structures of NFs to consolidate

Case ID	Algs to consolidate	# algs
Case 1	FIB-Cu, NAT-Cu, LB-Co	3
Case 2	FW-BF, CS, MRAC, MRB, HLL	5
Case 3	FW-BF, CS,CM, MRAC, MRB, HLL	6
Case 4	FIB-Buf (multiple BFs), NAT-Cu, LB-Co	7
Case 5	FIB-Buf (multiple BFs), NAT-Cu, LB-Co, CM, UnivMon, MRAC, MRB, HLL	9

Table 5.5: Descriptions of five cases of consolidations

Name	Loc	Date	# of flows	# of packets
Trace 1	Equinix-Chicago	02/19/2015	0.6M	15.8M
Trace 2	Equinix-Chicago	05/21/2015	0.4M	14.3M
Trace 3	Equinix-Chicago	01/21/2016	1.1M	29M
Trace 4	Equinix-Chicago	03/17/2016	0.8M	24.2M

Table 5.6: CAIDA traces as evaluation workloads

Resource cost on the hardware switch. We focus on the following types of resource cost when running the P4 programs on the Tofino switch.

- **Number of pipeline stages.** Each P4 program consumes a certain number of pipeline stages on the switch. When no remaining stage is available, the switch cannot support any more program.
- **SRAM.** SRAM shows the memory cost that is used to run P4 programs, including the part to support MATs and the part to maintain states, such as registers.
- **Hash calls.** Hash calls indicates the number of hash computations of a program.

The switch can only support a limited number of hash calls.

The amount of all these types of hardware resources on a switch is fixed and limited. For example, a commercial programmable switch, such as an EdgeCore 100BF Tofino-based programmable switch, support maximum 12 stages and each pipeline stage is equipped with 6 hash calls.

Throughput of packet processing. Throughput is another key performance metric. The algorithms of all cases running on the hardware switch can achieve full bandwidth. Hence we do not plot the corresponding figures. For software implementations, the throughput results are reported in million packet processing per second (MPPS).

Accuracy. Network measurements using sketches include errors. Hence we report the average relative error (ARE) and relative error (RE) of the evaluated algorithms. They are defined as follows.

- **Average relative error (ARE)**: This metric is to evaluate the accuracy of size estimation for elephant flows. It is defined as $\frac{1}{K}\sum_{i=1}^{K}|f_i - \hat{f}_i|/f_i$, where K denotes the top k elephant flows. f_i represents the actual flow size for flow i and $\hat{f}_i$ is the estimated flow size from the sketch algorithms.
- **Relative error (RE)**: This metric is to evaluate the accuracy of cardinality estimation of flows. It is defined as $|R - E|/R$, where R is the ground truth value of cardinality and E is the estimated cardinality from sketches.

5.4.3 Evaluation on the hardware switch

We run the consolidated algorithms generated by HyperMerger, the program generated by P4Visor, and the vanilla versions on the Tofino switch. All these programs can achieve the full bandwidth. The resource costs shown in this subsection are reported by a tool, Intel P4 Insight, which examines hardware costs for running programs on the Tofino switch.

Fig. 5.15 shows the SRAM usage of the five cases of consolidated algorithms generated by HyperMerger, compared to P4Visor and Vanilla. For Cases 1, 4, and 5, HyperMerger can merge multiple query MATs into a consolidated one that avoids storing the contents of the index field and hence they reduce SRAM cost. They result in reduced SRAM usage (by 42% for Case 1, 23% for Case 4, 71% for Case 5), compared to P4Visor. For Cases 2 and 3, the SRAM saving is very limited, because the space of their MATs are difficulty to reuse. Note in the experiments, we set the size of the data structures such that running the vanilla versions will not overwhelm the SRAM on the switch, otherwise they cannot be run. However from the results we may expect that even if the SRAM cost of the vanilla versions exceeds the resource limit, some consolidated algorithms by HyperMerger can still fit in the SRAM space.

Fig. 5.16 shows the cost of hash distribution units, considered as a metric for the number of hash calls, for the five cases. We find that in all cases, HyperMerger can significantly reduce the amount of hash distribution units (by 66% for Case 1, 31% for Case 2, 35% for Case 3, 50% for Case 4, 80% for Case 5), compared to P4Visor. We find that when more MATs use the same data structures (in Cases 1, 4, and 5), the saving of hash calls is bigger.

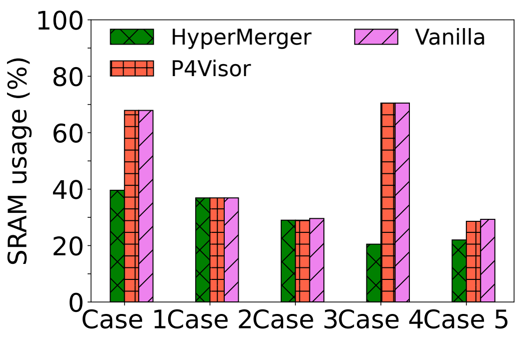


Figure 5.15: Usage of SRAM

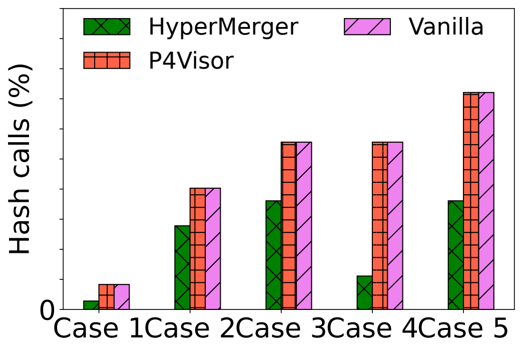


Figure 5.16: Usage of hash calls

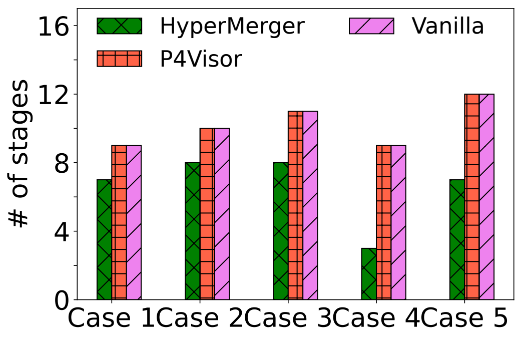


Figure 5.17: # of pipeline stages

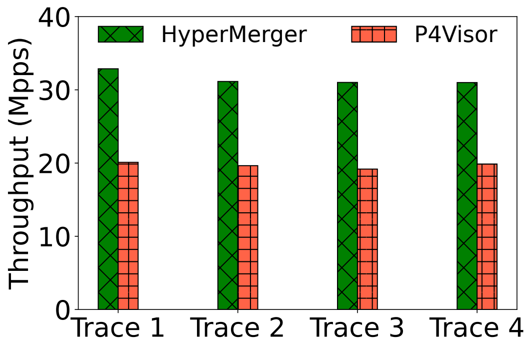


Figure 5.18: Thro. of case 1

Fig. 5.17 shows the total number of pipeline stages for the five cases. Again in all cases, HyperMerger reduces the number of pipeline stages (by 2 for Case 1, 2 for Case 2, 3 for Case 3, 5 for Case 4, 6 for Case 5), compared to P4Visor and Vanilla method. Considering there are only 12 stages in total, these savings are significant.

5.4.4 Evaluation on the software platform

In this subsection, we show the evaluation results of the throughput and measurement accuracy of the consolidated algorithms on a software platform implemented by us. The packet processing functions are running on the CPU. The consolidated algorithms are generated by HyperMerger in C code using JSON2C.

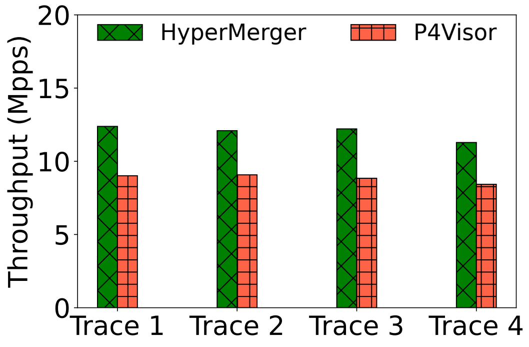


Figure 5.19: Thro. of case 2

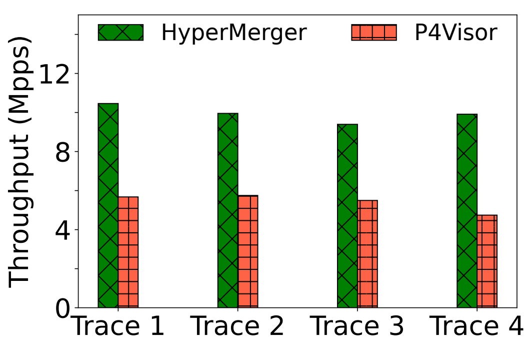


Figure 5.20: Thro. of case 3

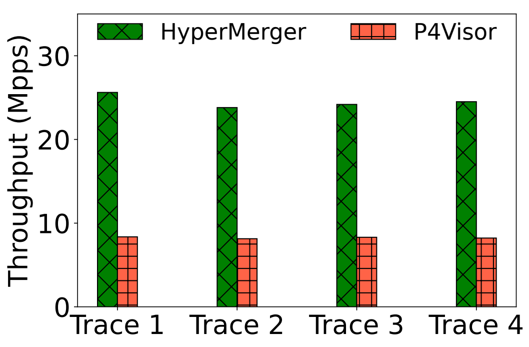


Figure 5.21: Thro. of case 4

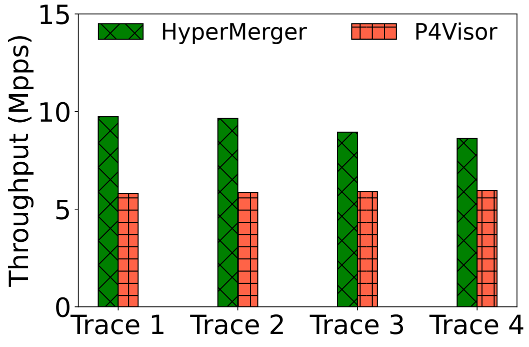


Figure 5.22: Thro. of case 5

Throughput evaluation. We test the packet processing throughput of all five cases compared to P4Visor and Vanilla. From Fig. 5.18 to Fig. 5.22, we show the throughput of all five cases of consolidated algorithms by HyperMerger and compare them to P4Visor. In each figure we show the results of four traffic workload labeled by Traces 1 to 4. We find that in every case and every workload, the algorithm generated by HyperMerger has higher throughput compared to P4Visor: 56%-64% higher for Case 1, 34%-38% higher for Case 2, 71%-109% higher for Case 3, around 3x for Case 4, and 45%-68% more for Case 5. The performance improvements mainly come from reusing

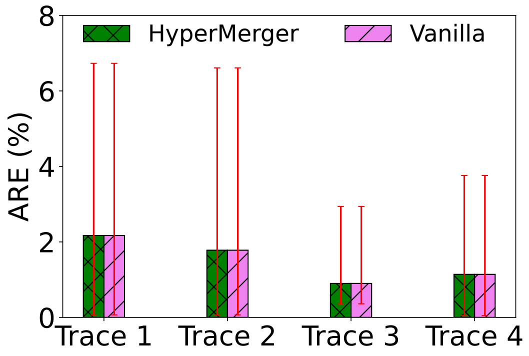


Figure 5.23: ARE of flow size estimation by Count sketch

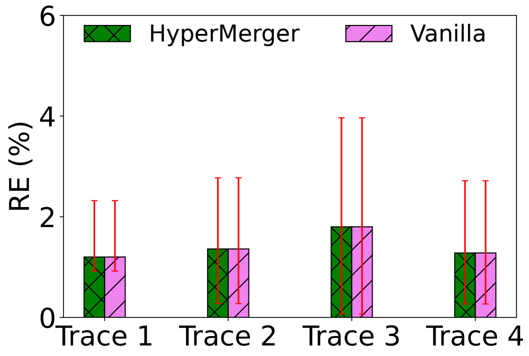


Figure 5.24: RE of flow cardinality by MRB

hash computations and merging tables to put query results in a same memory unit to reduce the number of memory accesses. Case 4 shows significant improvement by the consolidated algorithm because most hash computations can be reused and slots in different tables can be put in one memory unit.

Accuracy evaluation. We evaluate the accuracy of consolidated algorithms for packet measurement tasks to show that HyperMerger does not influence the accuracy of individual algorithms after merging them. We show the accuracy in the average relative error (ARE) of flow size estimation of the Count sketch and the relative error (RE) of flow cardinality estimation by MRB in Case 3, by comparing the consolidated algorithm with the Vanilla version. The Count Sketch consists of 5 arrays, each array has 8192 estimators. And each estimator uses 4 byte counters. MRB employs 16 bitmaps, the size of each bitmap is 8192. Results are shown in Fig. 5.23 and Fig. 5.24. We can find the consolidated algorithm by HyperMerger shows no difference in ARE and RE for all workloads compared to the Vanilla version. Hence consolidation will not affect the accuracy of measurement algorithms.

5.5 Related Work

5.5.1 Compact algorithms for NFs

Memory-efficiency and processing throughput are two key performance metrics of NFs because memory on network devices are precious resource and achieving line rate is essential. Therefore many NFs rely on space-compact data structures to achieve their tasks.

Compact and fast FIBs. Forwarding information bases (FIBs) are fundamental network functions which output the forwarding action for each packet, based on the lookup result of certain packet header fields (e.g., 5-tuple or destination IP/MAC address). BUFFALO [146] is a layer-two FIB design for enterprise and data center networks which leverages the Bloom filter [39] to store the destination addresses of each outgoing port. CuckooSwitch [160] is a software switch built on cuckoo hash tables [114] for fast and compact FIB lookups. Concise [149] further reduces memory cost of fast FIBs using Bloomier filters [43]. Ludo hashing [127] is a recent key-value lookup design that costs the least space among known FIB solutions and supports fast lookups and instant updates.

Sketching algorithms for network measurement. Sketching algorithms are widely used for various network measurement tasks [32,42,49,52,58,62,71,82,91,93, 94,145,147,155,156,161], such as flow size estimation, heavy hitter detection, and estimation of flow size distribution. Count sketch [42] and Count-Min sketch [52] are two fundamental sketches for flow size estimation and heavy hitter detection. MRB [58] and HyperLogLog [62] are sketches for cardinality estimation. OpenSketch [147] provides a simple three-stage pipeline to support many measurement tasks on programmable switches. FlowRadar [91] provides per-flow counters for all the flows for data center networks based on a encoding method. R-HHH [32] supports detection of hierarchical heavy hitters. UnivMon [94] is a general sketch-based solution based on the universal streaming theory that supports multiple measurement tasks. Elastic Sketch [145] is adaptive to traffic characteristics and generic to measurement tasks by separating elephant flows and mice flows. Zhou et al. [161] proposes generalized sketch families for network traffic measurement.

Compact data structures for other NFs. Cuckoo hash tables and its variants [125] are important structures for NAT. Many recent designs of layer-4 cloud load balancers also utilizes space-compact algorithms [99, 127]. Bloom filters have been used to implement cloud firewalls [9] and network measurement [91].

5.5.2 Programmable data plane

There have been research on supporting multiple network function programs on programmable switches [14, 69, 153, 157]. Hyper4 [69] and HyperV [153] propose virtualization of programmable data plane to execute multiple P4 programs at a time on a same switch. To reduce the resource overhead of Hyper4 and HyperV, P4Visor [157] and P4Bricks [14] utilize code merging technique to combine multiple P4 programs to a single P4 program. However, none of them utilizes the characteristics of algorithms and data structures to optimize multiple co-located network functions.

5.6 Summary

We demonstrate that consolidated algorithms for co-located NFs bring tremendous performance improvements in packet processing complexity and memory-efficiency compared to running these NFs as separate programs. More importantly, these performance gains do not cost extra hardware resources. To lower the barrier of designing consolidated NF algorithms, we present HyperMerger, an automatic tool to generate consolidated algorithms that can be run on both hardware and software network devices. The current implementation of HyperMerger can produce both P4 and C code. We study various cases of co-located NF algorithms and evaluated the consolidated ones by HyperMerger on a Tofino programmable switch and a software platform. The results show that HyperMerger can significantly optimize the NF algorithms by reducing SRAM cost, reducing hash calls, reducing the number of pipeline stages, and increasing throughput.

Chapter 6

Discussions

Designing low-cost and scalable network services is necessary and challenging as you need to handle large volume of data and consider resource capabilities on deployed devices. This dissertation is motivated by the needs of low-cost and scalable network services to meet network requirements under 5G. One key observation is that space-efficient data structures play an important role, leveraging space to save resources and gaining improvements in throughput. Thus, we utilized space-efficient data structures to optimize algorithms for efficient network services. In this chapter, we briefly summarize key contributions of the thesis work and discuss open issues for the future work.

6.1 Summary of Contributions

Designing a general space-efficient AMQ data structure. In Chapter 1, we discussed that space-efficient data structure plays an important role in designing network services. We designed Vacuum filters which is a space-efficient AMQ data structures. And we successfully utilized Vacuum filters to develop security and privacy services for IoT devices, showing the benefits of adopting space-efficient data structures. Vacuum filters can be widely used in other services, such as data management and the design of memory-efficient network functions. We will explore more practical services that can use Vacuum filters as a component in the future.

Optimizing memory-efficient algorithms for different network services. In this thesis, we choose several essential network services and build new designs by optimizing memory-efficient algorithms to make them adapt to today's network with more connected heterogeneous devices and a large volume of data. Specifically, we designed CCV to provide fast certificate validation for IoT devices, designed LOIS to protect sensitive information on packet headers to protect user privacy, and built the HyperMerger tool to generate consolidated algorithms for co-existed NFs automatically. The thesis work explores and validates the design principle that we can utilize efficient space-efficient data structures to optimize algorithms for network services when they need to handle a large volume of data and are deployed on network devices with limited resources.

Prototype and deployment. To validate designed algorithms and system design, we built the prototype of CCV by using the discrete event simulation model, implemented the LOIS framework using Intel Data Plane Development Kit (DPDK) [11], and validated the consolidated algorithm generated by the HyperMerger tool on a Tofino switch and a software switch.

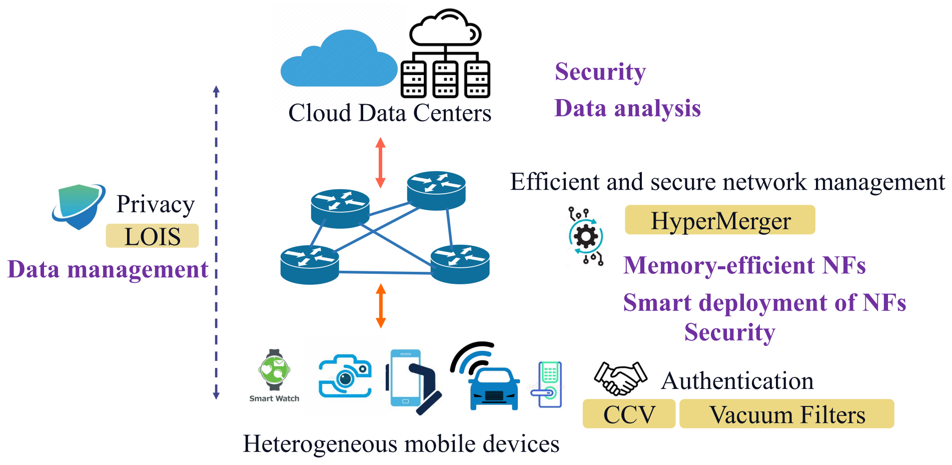


Figure 6.1: Big picture of research problems

6.2 Future Work

In the future, we will keep developing memory-efficient algorithms to provide efficient, scalable, and reliable network services in the 5G ecosystem. Our research objectives include: 1) new system design with better performance to support fundamental functions required by heterogeneous mobile devices in 5G, such as data management and network functions. 2) design of security and privacy services under 5G. We will focus on security and privacy problems related to applications. Fig. 6.1 shows the big picture of the research problems for low-cost, scalable, and reliable network services. We have done some work including Vacuum Filters, CCV, LOIS and HyperMerger, and here we list some potential research projects.

1) **Distributed data management.** A distributed data management system that provides data integrity, authenticity, access control, and fast data retrieval will be designed.

2) **NFs for high traffic rate.** We will focus on the design and implementation of algorithms and compact data structures for NFs to meet the requirement of super

high data rate and high mobility.

3) **Design of the smart SDN controller.** The deployment of network functions on network devices is a fundamental and important task. We believe that a smart controller that collects information of its managed devices and dynamically assigns NFs on network devices is required, especially for the 5G network with the network slicing technology.

4) **Security and privacy services.** The first part is security and privacy services for end devices. Several research topics are listed as follows: a) Design of privacy-preserving large-scale data analysis systems. b) Access control design for data retrieval to protect data privacy. c) Troubleshooting based on the collected traffic data and network data. For example, the detection of abnormal mobile devices and the problem locating when end devices experience performance degradation or cannot access services. The second part is security and privacy services for the core part of 5G. Some examples of research topics are shown as follows: a) Efficient defense of DoS and DDoS attacks on the SDN controller. b) Design of a controlling mechanism for a secure inter-network slices communication between functioning and network operators.

6.3 Concluding Remarks

It's a wonderful time to redesign memory-efficient algorithms for various network services to improve users' experience in the 5G era. In this section, I want to share my experiences in designing different network services.

1) **Bottleneck analysis for network services.** Analyzing the requirements

of network services and then learning the bottleneck are essential steps to design for efficient network services. When we developed the certificate validation method for low-power IoT devices, we found that the computational resource is limited and influences the throughput performance of validation results. Thus, we targeted to reuse validation results to save computation power. For the work HyperMerger to design resource-efficient algorithms for multiple co-exited network functions, we utilized Intel Vtune profiler to know that hash computation and memory access consume considerable cost on the algorithm. Thus we tried to reduce the number of hash computations and memory access to improve the performance when designing consolidated algorithms. Therefore, understanding bottlenecks gives us the direction to optimize memory-efficient algorithms for network services.

2) **Selecting proper platforms for network services.** Network services are deployed on a certain type of network device. Before designing the specific algorithm, we first need to choose the proper platforms to run network services. Then our algorithm also needs to consider the resource capacities of the platform. For example, suppose we design an algorithm for a network function on a hardware switch. In that case, we need to carefully design the algorithm to meet the requirements of the switch, such as limitations of the memory access model and limited support of complex operations.

3) **Prototype implementation and evaluation.** After designing core algorithms, it's better to implement the prototype and do a comprehensive evaluation to view the feasibility and effectiveness of our proposed method. There are many useful public resources, such as CloudLab [7] which is a testbed to do experiment with cloud architectures. Those resources are great tools for implementations and evaluations.

Bibliography

[1] A Fast Alternative to the Modulo Reduction. Accessed: 2019-04-10.

[2] A Smart Home is No Castle: Privacy Vulnerabilities of Encrypted IoT Traffic, author=Apthorpe, Noah and Reisman, Dillon and Feamster, Nick, journal=arXiv preprint arXiv:1705.06805, year=2017.

[3] Algorithms on Stings, Trees, and Sequences: Computer Science and Computational Biology.

[4] The caida ucsd anonymized internet traces 2013 - 2014. mar. `http://www.caida.org/data/passive/passive_2013_dataset.xml`.

[5] Campus LAN SDN Strategies: North American Enterprise Survey. `http://www.infonetics.com/pr/2015/Enterprise-LAN-SDN-Survey-Highlights.asp`.

[6] CBSTM-IoT: Context-based Social Trust Model for the Internet of Things, author=Rafey, Sherif Emad Abdel and Abdel-Hamid, Ayman and El-Nasr, Mohamad Abou, booktitle=Proc. of IEEE MoWNeT, pages=1–8, year=2016,.

[7] CloudLab. `https://www.cloudlab.us/`.

[8] Dynamic Trust Management for Internet of Things Applications, author=Bao, Fenye and Chen, Ing-Ray, booktitle=Proc. of ACM Self-IoT, pages=1–6, year=2012,.

[9] First Step toward Cloud-based Firewalling, author=Khakpour, Amir R and Liu, Alex X, booktitle=Proc. of IEEE SRDS, pages=41–50, year=2012,.

[10] Google Admits Its New Smart Speaker Was Eavesdropping on Users. `https://money.cnn.com/2017/10/11/technology/google-home-mini-security-flaw/index.html`.

[11] Intel DPDK: Data Plane Development Kit. `https://www.dpdk.org`.

[12] Intel VTune Amplifier. `https://software.intel.com/en-us/vtune`.

[13] Ipsec. `https://doc.dpdk.org/guides-16.04/sample_app_ug/ipsec_secgw.html`.

[14] P4Bricks: Enabling Multiprocessing using Linker-based Network Data Plane Architecture, author=Soni, Hardik and Turletti, Thierry and Dabbous, Walid, year=2018.

[15] Pktgen-DPDK. `https://github.com/pktgen/Pktgen-DPDK`.

[16] Cuckoo Filter Code. `https://github.com/efficient/cuckoofilter`, 2017.

[17] Implementation of Counting Quotient Filter. `https://github.com/splatlab/cqf`, 2017.

[18] Bloom Filter in Bitcoin. `https://bitcoin.org/en/developer-guide#bloom-filters`, 2019.

[19] Censys. `https://censys.io/certificates`, 2019.

[20] Implementation of Morton Filter. `https://github.com/AMDComputeLibraries/Morton_filter`, 2019.

[21] Implementation of Vacuum Filters. `https://github.com/wuwuz/Vacuum-Filter`, 2019.

[22] The Ethereum Project. `https://www.ethereum.org/`, 2019.

[23] Cisco Annual Internet Report (2018-2023) White Paper. *Cisco White Paper*, 2020.

[24] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. Peek-a-Boo: I See Your Smart Home Activities, Even Encrypted! In *Proc. of ACM WiSec*, pages 207–218, 2020.

[25] Kemal Akkaya, Nico Saputro, Samet Tonyali, Mumin Cebe, and Mohamed Mahmoud. Efficient Certificate Verification for Vehicle-to-Grid Communications. Technical report, Florida Intl Univ., Miami, FL (United States), 2017.

[26] Arwa Alrawais, Abdulrahman Alhothaily, Jiguo Yu, Chunqiang Hu, and Xiuzhen Cheng. Secureguard: A Certificate Validation System in Public Key Infrastructure. *Proc. of IEEE TVT*, 67(6):5399–5408, 2018.

[27] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet Caches on

Routers: the Implications of Universal Redundant Traffic Elimination. In *Proc. of ACM SIGCOMM*, 2008.

[28] Noah Apthorpe, Danny Yuxing Huang, Dillon Reisman, Arvind Narayanan, and Nick Feamster. Keeping the smart home private with smart (er) iot traffic shaping. *Proc. of PoPETs*, 2019(3):128–148, 2019.

[29] Noah Apthorpe, Dillon Reisman, and Nick Feamster. Closing the Blinds: Four Strategies for Protecting Smart Home Privacy from Network Observers. *arXiv preprint arXiv:1705.06809*, 2017.

[30] Manos Athanassoulis and Anastasia Ailamaki. BF-tree: Approximate Tree Indexing. *PVLDB*, 7(14):1881–1892, 2014.

[31] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer networks*, 54(15):2787–2805, 2010.

[32] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. Constant Time Updates in Hierarchical Heavy Hitters. In *Proc. of ACM SIGCOMM*, pages 127–140, 2017.

[33] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, , and E. Zadok. Don't Thrash: How to Cache Your Hash on Flash. *PVDLB*, 5(11):1627–1637, 2012.

[34] Bruhadeshwar Bezawada, Maalvika Bachani, Jordan Peterson, Hossein Shirazi, Indrakshi Ray, and Indrajit Ray. Iotsense: Behavioral Fingerprinting of IoT Devices. *arXiv preprint arXiv:1804.03852*, 2018.

[35] B. H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[36] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An Improved Construction for Counting Bloom Filters. In *Proc. of ESA*, 2006.

[37] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[38] Alex D Breslow and Nuwan S Jayasena. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *PVLDB*, 11(9):1041–1055, 2018.

[39] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet mathematics*, 1(4):485–509, 2004.

[40] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. *Proc. of ACM SIGOPS Operating Systems Review*, 36(SI):299–314, 2002.

[41] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *Proc. of ACM TOCS*, 26(2):4, 2008.

[42] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding Frequent Items in Data Streams. In *Proc. of EATCS ICALP*, pages 693–703. Springer, 2002.

[43] Denis Charles and Kumar Chellapilla. Bloomier Filters: A Second Look. In *Proc. of Springer ESA*, 2008.

[44] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proc. of ACM SODA*, pages 30–39, 2004.

[45] Dong Chen, Guiran Chang, Dawei Sun, Jiajia Li, Jie Jia, and Xingwei Wang. TRM-IoT: A Trust Management Model Based on Fuzzy Reputation for Internet of Things. *Computer Science and Information Systems*, 8(4):1207–1228, 2011.

[46] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. The Dynamic Cuckoo Filter. In *Proc. of IEEE ICNP*, 2017.

[47] Ray Chen, Jia Guo, and Fenye Bao. Trust Management for SOA-based IoT and Its Application to Service Composition. *Proc. of IEEE TSC*, 9(3):482–495, 2014.

[48] Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. SPEED: Resource-Efficient and High-Performance Deployment for Data Plane Programs. In *Proc. of IEEE ICNP*, pages 1–12, 2020.

[49] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *Proc. of ACM SIGCOMM*, pages 226–239, 2020.

[50] K. Christidis and M. Devetsikiotis. Blockchains and Smart Contracts for the Internet of Things. *Proc. of IEEE Access*, 2016.

[51] Bogdan Copos, Karl Levitt, Matt Bishop, and Jeff Rowe. Is Anybody Home? Inferring Activity from Smart Home Network Traffic. In *Proc. of IEEE SPW*, pages 245–251, 2016.

[52] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: the Count-Min Sketch and Its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[53] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *Proc. of ACM TODS*, 43(4):16, 2018.

[54] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching Using Bloom Filters. In *Proc. of ACM SIGCOMM*, 2003.

[55] Junqi Duan, Deyun Gao, Dong Yang, Chuan Heng Foh, and Hsiao-Hwa Chen. An Energy-Aware Trust Derivation Scheme with Game Theoretic Approach in Wireless Sensor Networks for IoT Applications. *Proc. of IEEE Internet of Things Journal*, 1(1):58–69, 2014.

[56] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proc. of USENIX NSDI*, 2016.

[57] David Eppstein. Cuckoo Filter: Simplification and Analysis. *arXiv preprint arXiv:1604.06067*, 2016.

[58] C. Estan, G. Varghese, and M. Fisk. Bitmap Algorithms for Counting Active Flows on High Speed Links. In *Proc. of the IMC*, 2003.

[59] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proc. of ACM CoNEXT*, pages 75–88, 2014.

[60] Li Fan, Pei Cao, Jussara Almeida, and Andrei Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *Proc. of IEEE/ACM ToN*, 8(3):281–293, 2000.

[61] Renjian Feng, Xiaofeng Xu, Xiang Zhou, and Jiangwen Wan. A Trust Evaluation Algorithm for Wireless Sensor Networks Based on Node Behaviors and Ds Evidence Theory. *Sensors*, 11(2):1345–1360, 2011.

[62] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the Analysis of a Near-optimal Cardinality Estimation Algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156, 2007.

[63] Xinwen Fu, Bryan Graham, Riccardo Bettati, Wei Zhao, and Dong Xuan. Analytical and Empirical Analysis of Countermeasures to Traffic Analysis Attacks. In *Proc. of IEEE ICPP*, pages 483–492, 2003.

[64] Carlos Gañán, Jose L Muñoz, Oscar Esparza, Jorge Mata-Díaz, Juan Hernández-Serrano, and Juanjo Alins. COACH: COllaborative Certificate StAtus CHecking Mechanism for VANETs. *Journal of network and computer applications*, 36(5):1337–1351, 2013.

[65] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud Scale Load Balancing with Hardware and Software. In *Proc. of ACM SIGCOMM*, 2014.

[66] Arthur Gervais, Srdjan Capkun, Ghassan O Karame, and Damian Gruber. On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients. In *Proc. of the ACM ACSAC*, 2014.

[67] A. Goel and P. Gupta. Small Subset Queries and Bloom Filters Using Ternary Associative Memories, with Applications. In *Proc. of ACM SIGMETRICS*, 2010.

[68] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[69] David Hancock and Jacobus Van der Merwe. Hyper4: Using P4 to Virtualize the Programmable Data Plane. In *Proc. of ACM CoNEXT*, pages 35–49, 2016.

[70] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust Network Measurement for Software Packet Processing. In *Proc. of ACM SIGCOMM*, pages 113–126, 2017.

[71] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Proc. of ACM SIGCOMM*, pages 576–590, 2018.

[72] René Hummen, Hossein Shafagh, Shahid Raza, Thiemo Voig, and Klaus Wehrle.

Delegation-based Authentication and Authorization for the IP-based Internet of Things. In *Proc. of IEEE SECON*, 2014.

[73] René Hummen, Jan H Ziegeldorf, Hossein Shafagh, Shahid Raza, and Klaus Wehrle. Towards Viable Certificate-based Authentication for the Internet of Things. In *Proc. of ACM workshop on HotWiSec*, 2013.

[74] Jinfang Jiang, Guangjie Han, Feng Wang, Lei Shu, and Mohsen Guizani. An Efficient Distributed Trust Model for Wireless Sensor Networks. *Proc. of IEEE TPDS*, 26(5):1228–1237, 2015.

[75] Audun Jøsang. A Logic for Uncertain Probabilities. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(03):279–311, 2001.

[76] Audun Jøsang and Touhid Bhuiyan. Optimal Trust Network Analysis with Subjective Logic. In *Proc. of IEEE SECURWARE*, 2008.

[77] Aman Khalid and Flavio Esposito. Optimized Cuckoo Filters for Efficient Distributed SDN and NFV Applications. In *Proc. of IEEE NFV-SDN*, pages 77–83, 2020.

[78] Minhaj Ahmad Khan and Khaled Salah. IoT Security: Review, Blockchain Solutions, and Open Challenges. *Future Generation Computer Systems*, 82:395–411, 2018.

[79] Hasan Ali Khattak, Munam Ali Shah, Sangeen Khan, Ihsan Ali, and Muhammad Imran. Perception Layer Security in Internet of Things. *Future Generation Computer Systems*, 100:144–164, 2019.

[80] Anna N Kim, Fredrik Hekland, Stig Petersen, and Paula Doyle. When HART goes Wireless: Understanding and Implementing the WirelessHART Standard. In *Proc. of IEEE ETFA*, 2008.

[81] Shogo Kitajima and Masahiro Mambo. Verifying the Validity of Public Key Certificates Using Edge Computing. In *Proc. of Springer SICBS*, 2017.

[82] Abhishek Kumar, Minho Sung, Jun Xu, and Jia Wang. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):177–188, 2004.

[83] Minseok Kwon, Pedro Reviriego, and Salvatore Pontarelli. A Length-Aware Cuckoo Filter for Faster IP Lookup. In *Proc. of IEEE INFOCOM WKSHPS*, 2016.

[84] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[85] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput. *PVLDB*, 12(5):502–515, 2019.

[86] James Larisch, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers. In *Proc. of IEEE SP*, 2017.

[87] D. Li, H. Cui, Y. Hu, Y. Xia, and X. Wang. Scalable Data Center Multicast using Multi-Class Bloom Filter. In *Proc. of IEEE ICNP*, 2011.

[88] Dan Li, Yuanjie Li, Jianping Wu, Sen Su, and Jiangwei Yu. ESM: Efficient and Scalable Data Center Multicast Routing. *Proc. of IEEE/ACM ToN*, 2012.

[89] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proc. of the ACM EuroSys*, 2014.

[90] Xin Li, Huazhe Wang, Ye Yu, and Chen Qian. An IoT Data Communication Framework for Authenticity and Integrity. In *Proc. of IEEE/ACM IoTDI*, 2017.

[91] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A Better Netflow for Data Centers. In *Proc. of USENIX NSDI*, pages 311–324, 2016.

[92] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. Flock-Lab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems. In *Proc. of ACM/IEEE IPSN*, 2013.

[93] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-based Monitoring in Software Switches. In *Proc. of ACM SIGCOMM*, pages 334–350. 2019.

[94] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with Univmon. In *Proceedings of ACM SIGCOMM*, pages 101–114, 2016.

[95] Javier Lopez, Ruben Rios, Feng Bao, and Guilin Wang. Evolving Privacy: From

Sensors to the Internet of Things. *Future Generation Computer Systems*, 75:46–57, 2017.

[96] Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic Nuggets in Content Delivery. *SIGCOMM Computer Communication Review*, 2015.

[97] Mohamed Mahmoud. Efficient Certificate Verification for Vehicle-to-Grid Communications. In *Proc. of Springer FNSS*, 2017.

[98] Tobias Maier, Peter Sanders, and Stefan Walzer. Dynamic Space Efficient Hashing. *Algorithmica*, 81(8):3162–3185, 2019.

[99] Rui Mao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*, 2017.

[100] M Hammad Mazhar and Zubair Shafiq. Characterizing Smart Home IoT Traffic in the Wild. *arXiv preprint arXiv:2001.08288*, 2020.

[101] Yair Meidan, Michael Bohadana, Asaf Shabtai, Juan David Guarnizo, Martín Ochoa, Nils Ole Tippenhauer, and Yuval Elovici. ProfilIoT: A Machine Learning Approach for IoT Device Identification based on Network Traffic Analysis. In *Proc. of ACM SAC*, pages 506–509, 2017.

[102] Diego M Mendez, Ioannis Papapanagiotou, and Baijian Yang. Internet of Things: Survey on Security and Privacy. *arXiv preprint arXiv:1707.01879*, 2017.

[103] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, N Asokan, Ahmad-Reza Sadeghi, and Sasu Tarkoma. IoT Sentinel: Automated Device-Type Identifica-

tion for Security Enforcement in IoT. In *Proc. of IEEE ICDCS*, pages 2177–2184, 2017.

[104] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive Cuckoo Filters. In *Proc. of SIAM ALENEX*, 2018.

[105] Sanaz Rahimi Moosavi, Tuan Nguyen Gia, Ethiopia Nigussie, Amir-Mohammad Rahmani, Seppo Virtanen, Hannu Tenhunen, and Jouni Isoaho. Session Resumption-Based End-to-End Security for Healthcare Internet-of-Things. In *Proc. of IEEE CIT/IUCC/DASC/PICOM*, 2015.

[106] Sanaz Rahimi Moosavi, Tuan Nguyen Gia, Amir-Mohammad Rahmani, Ethiopia Nigussie, Seppo Virtanen, Jouni Isoaho, and Hannu Tenhunen. SEA: A Secure and Efficient Authentication and Authorization Architecture for IoT-Based Healthcare using Smart Gateways. *Procedia Computer Science*, 52:452–459, 2015.

[107] M. Moradi, F. Qian, Q. Xu, Z. M. Mao, D. Bethea, and M. K. Reiter. Caesar: High-Speed and Memory-Efficient Forwarding Engine for Future Internet Architecture. In *Proc. of ACM/IEEE ANCS*, 2015.

[108] R Moskowitz and R Hummen. Hip Diet Exchange (dex). *draft-moskowitz-hip-dex-00 (WiP), IETF*, 2012.

[109] James K. Mullin. Optimal Semijoins for Distributed Database Systems. *Proc. of IEEE TSE*, 16(5):558–560, 1990.

[110] Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle

Adams. X. 509 Internet public key infrastructure online certificate status protocol-OCSP. Technical report, 1999.

[111] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks . *Proc. of IEEE Communications Surveys and Tutorials*, 2014.

[112] Alma Oracevic, Selma Dilek, and Suat Ozdemir. Security in Internet of Things: A Survey. In *Proc. of IEEE ISNCC*, 2017.

[113] Jorge Ortiz, Catherine Crawford, and Franck Le. DeviceMien: Network Device Behavior Modeling for Identifying Unknown IoT Devices. In *Proc. of ACM/IEEE IoTDI*, pages 106–117, 2019.

[114] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[115] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A General-Purpose Counting Filter: Making Every Bit Count. In *Proc. of ACM SIGMOD*, 2017.

[116] Francesco Paolucci, Filippo Cugini, Piero Castoldi, and Tomasz Osiński. Enhancing 5g sdn/nfv edge with p4 data plane programmability. *IEEE Network*, 35(3):154–160, 2021.

[117] Namje Park and Namhi Kang. Mutual Authentication Scheme in Secure Internet of Things Technology for Comfortable Lifestyle. *Sensors*, 16(1):20, 2016.

[118] Changhua Pei, Youjian Zhao, Guo Chen, Ruming Tang, Yuan Meng, Minghua Ma, Ken Ling, and Dan Pei. WiFi Can Be the Weakest Link of Round Trip Network Latency in the Wild. In *Proc. of IEEE INFOCOM*, 2016.

[119] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, Hash-and Space-Efficient Bloom Filters. In *Proc. of Springer International Workshop on Experimental and Efficient Algorithms*, pages 108–121, 2007.

[120] Martin Raab and Angelika Steger. Balls into BinsA Simple and Tight Analysis. *Randomization and Approximation Techniques in Computer Science*, pages 159–170, 1998.

[121] Jingjing Ren, Daniel J Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. Information Exposure from Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *Proc. of ACM IMC*, pages 267–279, 2019.

[122] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. 2012.

[123] Christian Esteve Rothenberg, Carlos AB Macapuna, Fábio L Verdi, and Mauricio F Magalhaes. The Deletable Bloom Filter: A New Member of the Bloom Family. *IEEE Communications Letters*, 14(6):557–559, 2010.

[124] Danilo FS Santos, Angelo Perkusich, and Hyggo O Almeida. Standard-based and Distributed Health Information Sharing for MHealth IoT Systems. In *Proc. of IEEE Healthcom*, pages 94–98, 2014.

[125] Nicolas Le Scouarnec. Cuckoo++ Hash Tables: High-Performance Hash Tables for Networking Applications. In *Proc. of ACM/IEEE ANCS*, pages 41–54, 2018.

[126] Mustafizur R Shahid, Gregory Blanc, Zonghua Zhang, and Hervé Debar. IoT Devices Recognition through Network Traffic Analysis. In *Proc. of IEEE BigData*, pages 5187–5192. IEEE, 2018.

[127] S. Shi and C. Qian. Ludo Hashing: Compact, Fast, and Dynamic Key-value Lookups for Practical Network Systems. In *Proc. of ACM SIGMETRICS*, 2020.

[128] Shouqian Shi, Chen Qian, and Minmei Wang. Re-designing Compact-Structure based Forwarding for Programmable Networks. In *Proc. of IEEE ICNP*, pages 1–11, 2019.

[129] Shouqian Shi, Ye Yu, Minghao Xie, Xin Li, Xiaozhou Li, Ying Zhang, and Chen Qian. Concury: A Fast and Light-weight Software Cloud Load Balancer. In *Proc. of ACM SoCC*, pages 179–192, 2020.

[130] Kuldeep Singh and Anil Kumar Verma. A Fuzzy-based Trust Model for Flying Ad Hoc Networks (FANETs). *International Journal of Communication Systems*, 31(6):e3517, 2018.

[131] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT Devices in Smart Environments using Network Traffic Characteristics. *Proc. of IEEE TMC*, 2018.

[132] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast

Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *Proc. of ACM SIGCOMM*, 2005.

[133] Tianyi Song, Ruinian Li, Bo Mei, Jiguo Yu, Xiaoshuang Xing, and Xiuzhen Cheng. A Privacy Preserving Communication Protocol for IoT Applications in Smart Homes. *IEEE Internet of Things Journal*, 4(6):1844–1852, 2017.

[134] Emily Stark, Lin-Shung Huang, Dinesh Israni, Collin Jackson, and Dan Boneh. The Case for Prefetching and Prevalidating TLS Server Certificates. In *Proc. of NDSS*, 2012.

[135] Yu Tang, Sathian Dananjayan, Chaojun Hou, Qiwei Guo, Shaoming Luo, and Yong He. A Survey on the 5G Network and Its Impact on Agriculture: Challenges and Opportunities. *Computers and Electronics in Agriculture*, 180:105895, 2021.

[136] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. Packet-Level Signatures for Smart Home Devices. *Signature*, 10(13):54.

[137] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. PingPong: Packet-Level Signatures for Smart Home Device Events. *arXiv preprint arXiv:1907.11797*, 2019.

[138] Floris Van den Abeele, Tom Vandewinckele, Jeroen Hoebeke, Ingrid Moerman, and Piet Demeester. Secure Communication in IP-based Wireless Sensor Networks via a Trusted Gateway. In *Proc. of IEEE ISSNIP*, pages 1–6, 2015.

[139] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vu-

vuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proc. of ACM SOSP*, pages 137–152, 2015.

[140] Liang Wang and Jussi Kangasharju. Real-world Sybil Attacks in BitTorrent Mainline DHT. In *Proc. of IEEE GLOBECOM*, pages 826–832, 2012.

[141] Minmei Wang, Chen Qian, Xin Li, Shouqian Shi, and Shigang Chen. Collaborative Validation of Public-Key Certificates for IoT by Distributed Caching. *Proc. of IEEE/ACM ToN*, 29(1):92–105, 2020.

[142] Minmei Wang and Mingxun Zhou. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. *Proc. of the VLDB Endowment*, 2019.

[143] Albert Wasef and Xuemin Shen. EMAP: Expedite Message Authentication Protocol for Vehicular Ad Hoc Networks. *Proc. of IEEE TMC*, 12(1):78–89, 2013.

[144] Zhuohan Xie, Wencheng Ding, Hongya Wang, Yingyuan Xiao, and Zhenyu Liu. D-Ary Cuckoo Filter: A Space Efficient Data Structure for Set Membership Lookup. In *Proc. of IEEE ICPADS*, 2017.

[145] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proc. of ACM SIGCOMM*, pages 561–575, 2018.

[146] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: Bloom Filter Forwarding Architecture for Large Organizations. In *Proc. of ACM CoNEXT*, 2009.

[147] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *Proc. of USENIX NSDI*, pages 29–42, 2013.

[148] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. A Concise Forwarding Information Base for Scalable and Fast Name Lookups. In *Proc. of IEEE ICNP*, 2017.

[149] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *Proc. of IEEE/ACM ToN*, 2018.

[150] Ye Yu, Xin Li, and Chen Qian. SDLB: A Scalable and Dynamic Software Load Balancer for Fog and Mobile Edge Computing. In *Proc. of ACM MECCOM*, 2017.

[151] Ye Yu, Chen Qian, and Xin Li. Distributed Collaborative Monitoring in Software Defined Networks. In *Proc. of ACM HotSDN*, 2014.

[152] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of Things for Smart Cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[153] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Hyperv: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. In *Proc. of IEEE ICCCN*, pages 1–9, 2017.

[154] Tong Zhang, Lisha Yan, and Yuan Yang. Trust Evaluation Method for Clustered Wireless Sensor Networks Based on Cloud Model. *Wireless Networks*, 24(3):777–797, 2018.

[155] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. CocoSketch: High-performance Sketch-based Measurement over Arbitrary Partial Key Query. In *Proc. of ACM SIGCOMM*, pages 207–222, 2021.

[156] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets. In *Proc. of USENIX NSDI*, pages 991–1010, 2021.

[157] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *Proc. of ACM CoNEXT*, pages 98–111, 2018.

[158] Serena Zheng, Noah Apthorpe, Marshini Chetty, and Nick Feamster. User Perceptions of Smart Home IoT Privacy. *Proc. of ACM HCI*, 2(CSCW):200, 2018.

[159] D. Zhou, B. Fan, H. Lim, D. G. Andersen, M. Kaminsky, M. Mitzenmacher, R. Wang, and A. Singh. Scaling Up Clustered Network Appliances with ScaleBricks. In *Proc. of ACM SIGCOMM*, 2015.

[160] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. of ACM CoNEXT*, 2013.

[161] You Zhou, Youlin Zhang, Chaoyi Ma, Shigang Chen, and Olufemi O Odegbile.

Generalized Sketch Families for Network Traffic Measurement. In *Proc. of ACM POMACS*, pages 1–34, 2019.