

Studies on Secure and Efficient Systems for Cloud-Independent Edge Computing

Naoki Matsumoto

Kyoto University

Graduate School of Informatics
Course of Communication and Computer Engineering
Multimedia and Secure Networking Group

Studies on Secure and Efficient Systems for Cloud-Independent Edge Computing

Naoki Matsumoto

- | | |
|--------------------|---|
| <i>1. Reviewer</i> | Takeshi Iwashita
Graduate School of Informatics
Kyoto University |
| <i>2. Reviewer</i> | Kazuyuki Shudo
Graduate School of Informatics
Kyoto University |
| <i>Supervisor</i> | Yasuo Okabe |

Naoki Matsumoto

Studies on Secure and Efficient Systems for Cloud-Independent Edge Computing

Reviewers: Takeshi Iwashita and Kazuyuki Shudo

Supervisor: Yasuo Okabe

Kyoto University

Multimedia and Secure Networking Group

Course of Communication and Computer Engineering

Graduate School of Informatics

List of Included Publications

Refereed Journal Articles

1. Naoki Matsumoto, Daisuke Kotani and Yasuo Okabe, *CREBAS: Enabling Network Access Control in a Home with One Click*, Journal of Information Processing, Volume 31, pp. 174-184, 2023, <https://doi.org/10.2197/ipsjjip.31.174>

Notice for the use of this material: The copyright of this material presented in Chapter 3 is retained by the Information Processing Society of Japan (IPSJ). This material is published on this web site with the agreement of the author (s) and the IPSJ. Please be complied with Copyright Law of Japan and the Code of Ethics of the IPSJ if any users wish to reproduce, make derivative work, distribute or make available to the public any part or whole thereof. All Rights Reserved, Copyright (C) Information Processing Society of Japan.

2. Naoki Matsumoto, Daisuke Kotani and Yasuo Okabe, *D4C: Fast and Efficient Container Image Updating for Edge with Delta Encodings*, IEICE TRANSACTIONS on Communications, vol. E108-B, no. 7, pp. 780-795, July 2025, <https://doi.org/10.23919/tra nscom.2024EBT0008>

The material of Chapter 5 was presented on "D4C: Fast and Efficient Container Image Updating for Edge with Delta Encodings", which appeared in IEICE TRANSACTIONS on Communications, Copyright(C) 2025 IEICE, and figures of the chapter are reused from it under the permission of the IEICE.

Refereed International Conference Articles

1. Naoki Matsumoto and Akihiro Suda, *bypass4netns: Accelerating TCP/IP Communications in Rootless Containers*, In Proceedings of the Asian Internet Engineering Conference 2024 (AINTEC '24), Association for Computing Machinery, pp. 67-75, 2024, <https://doi.org/10.1145/3674213.3674221>
2. Naoki Matsumoto, Tomoya Kawanishi, Kenji Ohira and Masahiro Kozuka, *yak: WebAssembly-based Data Analysis Platform with Data Providers Driven Analysis Control*, In 2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, pp. 1116-1125, 2025, <https://doi.org/10.1109/COMPSAC65507.2025.00144>

The material of Chapter 6 was Copyright(C) 2025 IEEE. Reprinted, with permission, from "yak: WebAssembly-based Data Analysis Platform with Data Providers Driven Analysis Control", which appeared in 2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC). In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Kyoto University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Abstract

Edge computing (EC) processes data close to its source. Cloud service providers have introduced cloud-based EC platforms that let developers deploy and operate applications on edge nodes, including IoT devices, while offloading node management to the provider. Although convenient, cloud-based EC introduces significant challenges in availability and privacy. Platform outages can disrupt dependent services and prevent users from operating critical applications and devices. Moreover, many cloud-based EC designs require edge-produced data to be transferred to the cloud for integration with other services, creating privacy risks because users must trust the platform operator to handle their data appropriately.

To address these challenges, I leverage a cloud-independent architecture called Region-based EC. In this architecture, edge nodes and devices within a local region continue to operate and serve applications independently of the cloud platform. Region-based EC improves availability and enhances privacy by reducing reliance on the cloud platform and keeping sensitive data within the region. Despite these benefits, Region-based EC presents challenges: (1) devices within a region may be untrusted, necessitating secure intra-region communication, (2) deploying securely and updating applications on resource-constrained edge nodes is difficult, and (3) maintaining data privacy when integrating across regions or with external services is non-trivial.

This dissertation addresses these challenges with four contributions. First, I present CREBAS, a capability-based access control system that enforces communication control within a region, mitigating threats from untrusted devices. Using a capability-based authorization architecture, CREBAS enables non-expert users to configure fine-grained network access control. Second, I leverage lightweight container technologies for resource- and bandwidth-constrained edge nodes and introduce two enhancements: `bypass4netns` and `D4C`. Rootless containers is a more secure containerization approach that runs without elevated privileges, but it suffers from networking performance overhead. `bypass4netns` addresses this bottleneck with a socket-switching technique and significantly improves throughput and application performance. `D4C` reduces the time and bandwidth required to update images via merge-based delta generation and a dedicated filesystem that applies deltas lazily. `D4C` provides faster and more efficient updates than existing methods, making it

feasible to keep edge nodes up to date. Third, I introduce yak, a data analysis platform that allows data providers to specify prohibited operations and require anonymization when desired. With yak, data providers can collaborate securely with external analysts without leaking sensitive information.

These contributions advance the practicality of Region-based EC by addressing key challenges in security, application deployment, and data privacy. Independence from cloud platforms creates opportunities for developing and deploying a wide range of applications and services. This dissertation paves the way toward realizing my vision of Region-based EC.

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Prof. Yasuo Okabe, for his excellent guidance. His mentorship was not limited to the research itself but also covered essential aspects of my life.

I am deeply grateful to Prof. Takeshi Iwashita and Prof. Kazuyuki Shudo for their insightful comments and helpful advice during my dissertation defense.

This work would not have been possible without the support of my co-authors: Assoc. Prof. Daisuke Kotani, Mr. Akihiro Suda, and Prof. Masahiro Kozuka. I truly appreciate their valuable discussions and contributions to advancing this research.

I was also fortunate to spend my daily research life with the members of Okabe Laboratory. Discussions with them provided me with inspiration, insightful ideas, and the motivation to pursue my work.

My sincere thanks also go to my friends and colleagues: Yuki Takezawa, Ryotaro Banno, Kotaro Matsuoka, Naoyuki Okamoto, Tatsuya Baba, and Kosuke Umeki. Their encouragement and daily discussions were indispensable for completing this dissertation. Our deep friendship is one of the most precious treasures I gained during my PhD life, and I will cherish it forever.

Last but not least, I would like to thank my family for their continuous support, patience, and understanding throughout my PhD journey.

Contents

1	Introduction	1
2	Background and Challenges in Region-based Edge Computing	7
2.1	Definition of Edge Computing	7
2.2	Benefits and Issues of Cloud-based Edge Computing	8
2.3	Region-based Edge Computing for Cloud-Independent Architecture	9
2.4	Issues for Region-based Edge Computing	11
2.4.1	Intra-Region Security	11
2.4.2	Management and Integration by Platform Providers	13
2.4.3	Computation at the Resource Constrained Edge Nodes	13
2.4.4	Issues for Container-based Edge Computing	16
2.4.5	Collaboration While Privacy-Preserving Data Analysis	17
3	Network Access Control with User-friendly Authorization	19
3.1	Introduction	19
3.2	Related Work	22
3.2.1	Network Access Control in a Home	22
3.2.2	Capability-based Access Control	23
3.3	Authorization Architecture	24
3.3.1	The Guidelines for Access Control at Home	24
3.3.2	Overview of the Authorization Architecture	24
3.3.3	Automatic Authorization	26
3.3.4	Flows of Authorization	27
3.4	Network Access Control	28
3.4.1	Overview of Network Access Control	28
3.4.2	A Twin Device	29
3.4.3	Access Control Methods	31
3.4.4	The Flow of Access Control	32
3.4.5	A Trick for Forwarding Packets	32
3.5	Implementation and Evaluation	34
3.5.1	Prototype Implementation	34
3.5.2	Performance Evaluation	34
3.5.3	Resource Consumption in PEP	35

3.5.4	PoC with a Real Protocol	38
3.6	Discussion	39
3.6.1	The Validity of the Authorization Architecture	39
3.6.2	The Effectiveness of Automatic Authorization	42
3.6.3	Comparison with Prior Work	42
3.6.4	The Limitations of CREBAS	43
3.7	Conclusion	44
4	Acceleration of TCP/IP Communications in Rootless Containers	45
4.1	Introduction	45
4.2	Background	47
4.2.1	Rootless Containers	47
4.2.2	Seccomp Notify	48
4.2.3	Socket Switching	49
4.3	The Design of bypass4netns	50
4.3.1	Overview of bypass4netns	50
4.3.2	Analyzing Socket Behavior with eBPF	51
4.3.3	Socket Switching With Seccomp Notify	52
4.3.4	Publishing Ports with Socket Switching	53
4.3.5	Fast Multi-node Communication	54
4.4	Implementation and Insights	56
4.4.1	Implemented Components	56
4.4.2	Insights in Implementation	56
4.5	Evaluation	57
4.5.1	Environment	57
4.5.2	Switching Overhead	58
4.5.3	Application Compatibility	58
4.5.4	Application Performance	59
4.6	Discussion	62
4.6.1	Security Considerations	62
4.6.2	Kubernetes Support	63
4.6.3	As a Versatile Socket Switching Method	63
4.7	Conclusion	63
5	Efficient Container Image Updating with Delta Encodings	65
5.1	Introduction	65
5.2	Background	67
5.2.1	Leveraging Containers at the Edge	67
5.2.2	Container Image Update Issues	67

5.2.3	Issues in Applying Delta Encoding to Image Updating	68
5.3	Related Work	69
5.3.1	Reduction in Data Size and Time Required for Pull	69
5.3.2	Content Distribution Using Delta Encoding	71
5.4	Design of D4C	72
5.4.1	Approaches to Apply Delta Encodings	72
5.4.2	Overview of D4C	73
5.4.3	Generating Delta Bundles	74
5.4.4	Merge-based Delta Bundle Generation Strategy	76
5.4.5	Merging bsdiff's Delta Files	78
5.4.6	Di3FS	81
5.5	Implementation	82
5.5.1	Di3FS	82
5.5.2	Dedicated Snapshotter Plugin	83
5.6	Evaluation	83
5.6.1	Evaluation Environment	83
5.6.2	Generating Delta Bundles	84
5.6.3	Merging Delta Bundles	87
5.6.4	Breakdown of Merge and DeltaMerging	89
5.6.5	Applying Delta Bundle	90
5.6.6	Image Pulling with D4C	91
5.6.7	Di3FS Performance	92
5.6.8	Impact on Application Performance	94
5.7	Discussion	96
5.7.1	Delta Bundle Generation with Merge Strategy	96
5.7.2	Overhead and Application Performance with Di3FS	97
5.7.3	Lower CPU or Storage Environment	98
5.7.4	Practicality in Daily Operations	99
5.8	Conclusion	100
6	Data Analysis Platform with Data Provider's Control	101
6.1	Introduction	101
6.2	Related Work	103
6.2.1	TEE-based Data Management and Analysis System	103
6.2.2	Personal Data Management System	104
6.3	Overview of Data Analysis Platform	104
6.4	Analysis Control with Policy	105
6.4.1	Requirements and Approaches	105
6.4.2	Fundamental Design	106

6.4.3	Formulation for Policy Format and Decision Making	107
6.4.4	Flow of Analysis Control	109
6.5	yak: WebAssembly-based Analysis Platform	111
6.5.1	Towards Implementing Analysis Control	111
6.5.2	Design of yak	111
6.5.3	WebAssembly-based Module System	112
6.5.4	Defining Analysis Procedure with DAG	114
6.5.5	Providing Policies and Data From Remote with AuthN	115
6.5.6	Volatile Platform without Persistent Storage	116
6.5.7	Remote Attestation Protocol for Data Providers	116
6.6	Threat model	118
6.7	Evaluation	118
6.7.1	Performance Evaluation with Synthetic Data	118
6.7.2	Use Case: Anonymization based on Policy	121
6.7.3	Case Study: Analysis of Financial Products	122
6.8	Discussion	123
6.8.1	Validity and Risk of Analysis Control	123
6.8.2	Usability for Data Providers	124
6.8.3	Analysis Performance	124
6.9	Conclusion	125
7	Conclusion	127
	Bibliography	129
	List of Figures	147
	List of Tables	151

Introduction

Edge Computing (EC) is a paradigm that places computing resources closer to data sources, such as Internet of Things (IoT) devices, rather than relying on centralized cloud data centers. It enables low-latency communication and supports applications that require real-time processing. Fig. 1.1 shows an overview of the current cloud-oriented EC architecture for IoT. Cloud Service Providers (CSPs) provide services that manage edge nodes, including IoT devices, and encourage customers to deploy applications on those nodes. Mobile carriers also provide a similar concept known as Multi-access Edge Computing (MEC) [1]. MEC deploys applications on edge nodes located near base stations and enables low-latency access to these applications. Both aim to provide low-latency services by placing computing resources closer to users instead of relying on centralized data centers.

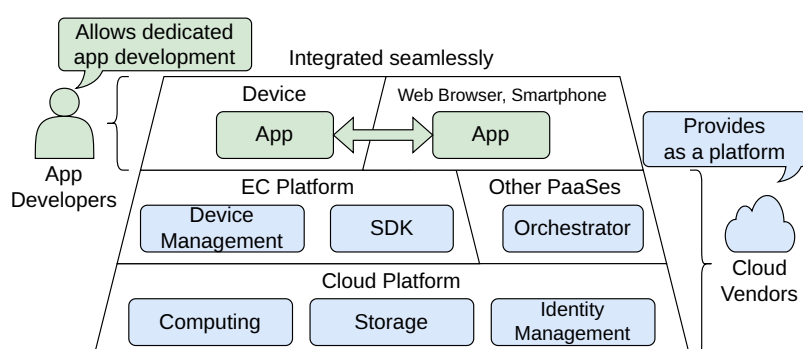


Fig. 1.1: Cloud-based Edge Computing architecture

However, CSP- and mobile-carrier-oriented EC architectures have several issues regarding availability and privacy. CSPs provide an all-in-one platform that includes edge device management, application deployment, and integration with web services running on the platform. As shown in Fig. 1.1, many users access services from their devices, but the entire platform relies on a single CSP. The current EC architecture is **tightly coupled**: edge nodes depend heavily on the cloud platform. This architecture represents an extreme form of centralization. The more functionality is aggregated into the platform, the more responsibility it bears and the less autonomy users retain. If the platform goes down due to failures or attacks, many users are affected simultaneously. This problem goes beyond service disruptions such as e-commerce outages. It poses a serious threat that can make even basic household operations,

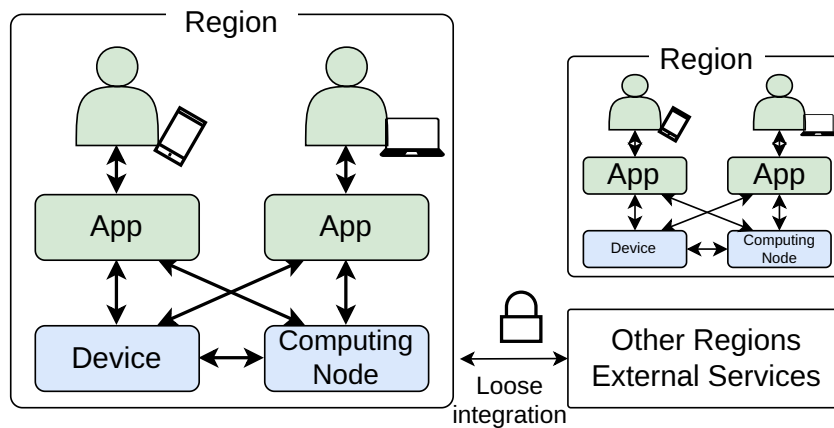


Fig. 1.2: Region-based Edge Computing architecture

such as turning lights on and off, impossible. Users have no option but to wait until the platform recovers. This availability risk is critical and has already materialized in daily life [2, 3].

It also raises privacy concerns because all data generated by users' devices is collected and managed by service providers. Even if the service provider ensures data confidentiality and publishes privacy policies, there have been cases of non-compliance [4, 5]. Users are expected to trust the service provider, but past incidents show that such trust is not always justified. Researchers are actively studying cryptographic approaches [6, 7] to protect data privacy in cloud computing. However, these approaches often require significant computational resources and may apply only to specific algorithms. In principle, users can protect privacy by keeping their data under their own control. The tightly coupled architecture makes this difficult because edge nodes are designed to work with the cloud platform, sharing or aggregating their data.

A loosely coupled EC architecture can mitigate the availability and privacy issues. In this dissertation, I refer to this architecture as Region-based Edge Computing (Region-based EC). Fig. 1.2 shows an overview of the architecture. In this architecture, edge nodes in a local region process and consume data locally. Each region is self-sufficient and can operate independently of the cloud platform even when the platform is unavailable. It also avoids sending data to the cloud whenever possible. Privacy-sensitive data is retained within the local region, and only non-sensitive data is sent to the cloud platform. This is similar to the Region-Based Trust-Aware (RBTA) model [8]. The RBTA model clearly defines the boundary for private data by distinguishing a local region from external regions.

A Region-based EC architecture can improve availability and privacy, but it still presents several challenges. First, the RBTA model assumes that all devices in a local region are trustworthy, which is not always the case. Prior work has shown that IoT devices are often compromised or contain backdoors [9, 10]. Malicious devices can sniff network traffic or attack other devices in the same local region. Cloud platforms can leverage near-unlimited resources to detect and mitigate such attacks, but edge nodes and devices have limited resources. For a more realistic architecture, I relax this assumption and consider potential malicious behavior within the local region. Second, deploying and maintaining applications on resource-constrained edge nodes is challenging. Edge nodes need to accommodate diverse applications while ensuring the security of both the node and the region. Moreover, both the bandwidth and stability of cloud-edge links are often limited. This prevents developers from deploying applications frequently. To leverage this EC architecture, it is essential to develop secure and efficient application deployment and maintenance methods. Third, some services still require collaboration with external services, including cloud services. A method is required to maintain data privacy during such collaborations.

In this dissertation, I propose improvements toward realizing a Region-based EC architecture by addressing the above challenges. The central goal is to enable self-sufficient and secure edge regions that operate with looser coupling to the cloud. To tackle the security challenge in local regions, I propose CREBAS, a capability-based network access control system that provides fine-grained control over intra-region communications. For running applications on resource-constrained edge nodes, I assume container-based approaches. To address the challenges of deploying and maintaining containerized applications at resource-constrained edge nodes, I propose `bypass4netns` and `D4C`. `bypass4netns` improves network performance for rootless containers by bypassing bottlenecks in their networking stack. `D4C` enables fast and efficient container image updates using delta encoding, significantly reducing the data transferred over constrained networks. With these methods, edge nodes can securely and efficiently run various applications. Some services still require collaboration with external services, including cloud services. Maintaining data privacy during such collaborations is challenging. To enable privacy-preserving collaboration with external services, I propose `yak`, a data-analysis platform that implements an analysis-control mechanism. Analysis control enables data providers (users in a region) to enforce fine-grained policies governing how their data is analyzed within a Trusted Execution Environment. This allows data providers to maintain control over their data even when collaborating with external services, preventing data leakage.

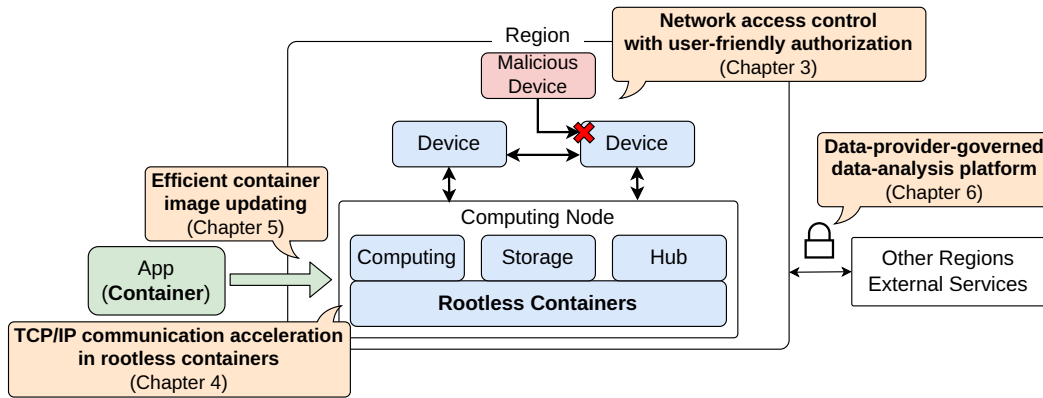


Fig. 1.3: Improvements toward a region-based edge computing architecture

The primary contributions of this work are summarized in Fig. 1.3 and listed below:

1. **Network access control with user-friendly authorization (CREBAS):** CREBAS provides a fine-grained, capability-based network access control system for IoT devices. It enables users to enforce communication policies, preventing malicious or compromised devices from launching attacks against other devices in the same local region.
2. **Efficient and secure application deployment at the edge (bypass4netns and D4C):** Deploying applications on resource-constrained edge nodes presents practical challenges. This dissertation addresses two key issues:
 - **TCP/IP communication acceleration in rootless containers (bypass4netns):** bypass4netns accelerates TCP/IP communication performance for rootless containers, a more secure containerization method, by bypassing bottlenecks in their networking stack.
 - **Efficient container image updating (D4C):** D4C provides a fast and efficient image update mechanism using delta encoding. It significantly reduces data transfer, making updates feasible even over low-bandwidth networks.
3. **Data-provider-governed data-analysis platform (yak):** yak enables data providers to enforce fine-grained policies that govern how their data is analyzed. This gives providers control over the analysis process itself, preventing data leakage through malicious queries and ensuring that results are anonymized according to their specifications.

This dissertation is structured as follows. Chapter 2 provides the necessary background on edge computing, its security challenges, application deployment approaches, and privacy-preserving techniques. Chapter 3 presents the design and evaluation of CREBAS. Chapter 4 introduces the `bypass4netns` method for accelerating rootless container networking. Chapter 5 details the D4C system for efficient container updates. Chapter 6 describes the yak platform and its analysis-control framework. Finally, Chapter 7 concludes the dissertation and discusses potential directions for future research.

Background and Challenges in Region-based Edge Computing

This chapter presents the trends in Edge Computing (EC) and the issues inherent in cloud-based EC. As a solution to these issues, I introduce a Cloud-Independent EC architecture and its design, termed Region-based EC. Finally, I present the challenges that must be addressed to realize Region-based EC and outline the contributions of this dissertation.

2.1 Definition of Edge Computing

Edge Computing (EC) emerged as a complementary paradigm to cloud computing. Its primary objectives are to improve response times and conserve bandwidth by processing data closer to the source rather than transmitting it to centralized cloud infrastructure. With the proliferation of IoT, EC has evolved in tandem to enable real-time applications by processing collected data near the devices where it originates, thereby reducing latency.

EC serves as a broad term encompassing various architectures. It may simply refer to “computing at the edge”, or it may denote a particular standardized architecture. Within this context, I distinguish between two key concepts: MEC (Multi-access Edge Computing) [1] and unstandardized architectures including EC-based IoT [11] and Fog Computing [12].

MEC aims to reduce latency by processing services at base stations and within mobile network infrastructure. Mobile network operators are transitioning traditionally centralized network functions into VNFs (Virtualized Network Functions) deployed on edge nodes, processing data closer to users to deliver low-latency services. This concept is referred to as Network Functions Virtualization (NFV). Within the NFV domain, significant research focuses on methods for optimally deploying VNFs across

networks based on traffic characteristics and load conditions [13]. The MEC architecture has been standardized by ETSI [1], and mobile network operators and vendors are developing and deploying services in accordance with these specifications. MEC is not intended for direct end-user involvement; rather, it is designed as a platform for mobile network operators to deploy their own services—not as a space for end users to freely deploy applications.

EC architectures, which refer to computing near users and data sources, aim to process data generated by IoT devices at edge nodes, reducing latency and minimizing the amount transmitted to the cloud. These architectures do not have standardized specifications like MEC or common implementations; researchers and cloud service providers independently propose and implement their own solutions. Systems adopting this architecture typically feature a three-tier hierarchical structure. The bottom tier consists of resource-constrained IoT devices, the middle tier contains more capable edge nodes such as gateways and on-premise servers, and the top tier comprises the resource-rich cloud. An abstraction of the middle tier is sometimes referred to as Fog Computing. In contrast to MEC, developers and researchers can independently design custom architectures tailored to specific use cases or build software stacks based on their requirements.

This dissertation focuses on EC architectures that are not constrained by standardized specifications such as MEC. While MEC is standardized for mobile network operators and primarily targets mobile environments, the scope of this dissertation extends to other settings, including residential environments. Consequently, to address challenges and solutions at the architectural level, I examine EC architectures that are independent of specific standards like MEC.

2.2 Benefits and Issues of Cloud-based Edge Computing

As illustrated in Fig. 1.1, some EC implementations rely on cloud service provider (CSP) platforms for device management and service integration. Application and service developers benefit from the unified platform design, enabling easy integration with various services provided on the platform. AWS IoT [14] is a representative example of such an approach. Developers can manage IoT devices and deploy applications using AWS IoT services, leveraging computational (AWS Lambda [15]) and storage (AWS S3 [16]) resources provided by AWS. Without such a platform, developers must build and operate infrastructure for device management, computation, and

storage themselves. By utilizing CSP platforms, developers can significantly reduce development and maintenance effort. This is a major advantage of cloud-based EC architectures.

While developers benefit from the merits of cloud-based EC, security risks stemming from CSP dependence also exist. First, the platform can become a single point of failure. Major outages at AWS [3] have caused numerous IoT devices to cease functioning properly [2], demonstrating that reliance on CSP platforms introduces significant availability risks. Users cannot control their devices, even if they have them in their hands, until the platform recovers. Platform failures are infrequent; however, when they occur, the impact tends to be widespread and can result in severe disruption. Furthermore, since IoT devices are managed through the CSP's platform, both device operations and collected data fall under CSP control. This creates potential vulnerabilities: if the CSP acts maliciously, it could compromise device control (an availability issue) and enable unauthorized use of collected data (a privacy concern). In practice, there have been cases where service providers failed to adhere to privacy protections mandated by regulatory bodies [4, 5]. Thus, while relying on CSP platforms can reduce development and maintenance costs, it also increases security risks.

2.3 Region-based Edge Computing for Cloud-Independent Architecture

To address these issues, Region-based Edge Computing (Region-based EC), a cloud-independent edge computing architecture, is crucial for the future evolution of EC. In EC systems built upon CSP-provided platforms, the tight coupling between platform services and user-held devices introduces both availability and privacy concerns. Region-based EC aims to provide services in a maximally decoupled manner, avoiding CSP-centric integrations.

In Region-based EC, a “Region” is the administrative domain in which users directly control their devices and data. The Region-Based Trust-Aware (RBTA) model [8] addresses privacy concerns by distinguishing the sensitivity of data handled within versus outside regions. In the RBTA model, a region is defined as a collection of devices or Fog computing nodes that are geographically close and serve as the boundary of trust. However, users may remotely access devices at home from an office while still retaining control over their devices and data. Therefore, Region-based EC is not bound to geographical proximity.

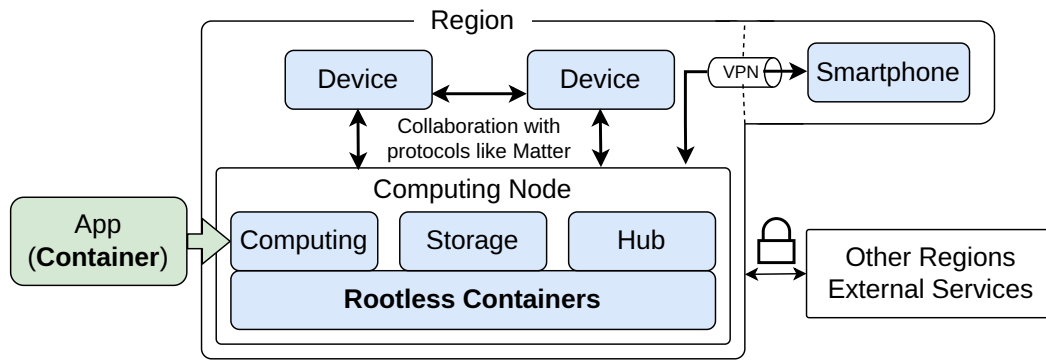


Fig. 2.1: Architecture of Region-based Edge Computing

In Region-based EC, a region is defined not by physical proximity or network locality but by *user control*. *User control* means that users can directly manage their devices and data without relying on external services. Thus, the boundary of a region is the scope within which users can exercise this control. This control includes not only administrative control over devices, such as root privileges, but also independence from specific platforms or contracts. While CSPs offer their own platforms and services, users must rely on the CSPs' infrastructure, software licenses, and policies to utilize them. In contrast, Region-based EC enables users to manage their devices and data without being limited by the constraints of specific platforms or contracts. Consequently, the software stack within Region-based EC is expected to utilize and enhance open-source software (OSS) to eliminate dependence on proprietary platforms. For instance, AWS Outposts [17] and Oracle Alloy [18] are examples of private cloud solutions that provide cloud-like services in an on-premises environment. However, these solutions are designed to integrate tightly with their cloud platforms, meaning users do not have full control over the software stack within the Region. Therefore, they are not considered Region-based EC solutions. Accordingly, “edge computing” in this context refers to computation performed within the region, close to the user, rather than strictly at the network edge. With this definition, Region-based EC does not necessarily guarantee low latency in all scenarios, such as accessing home devices from a remote office. Rather than focusing on latency reduction, Region-based EC prioritizes user control over devices and data, addressing availability and privacy concerns inherent in cloud-based EC architectures.

Fig. 2.1 illustrates the architectural overview. Traditional EC systems have focused on enabling edge nodes and devices to function similarly to cloud services, emphasizing integration with various services and centralized management. Region-based EC instead prioritizes providing independent services to users without dependence on external services like CSPs. Mechanisms for device coordination without reliance

on CSPs or other external entities have already been implemented; Matter [19] is a prominent example for IoT devices. Computing nodes are also deployed within the region to provide computational and storage resources. As such nodes, I consider using users' laptops or tablets, which provide sufficient computational resources to host applications. In Region-based EC, the goal is to provide services entirely independently of external services. However, a fully standalone implementation is impractical for modern services, particularly when they must consume public data or interoperate with external ecosystems. Therefore, Region-based EC should still incorporate mechanisms for necessary external integration. This architecture aims to provide basic functionality without depending on external services, while also enabling flexible integration when needed. I position Region-based Edge Computing as an architectural direction for addressing these availability and privacy concerns while reducing dependence on CSP-centric platforms.

2.4 Issues for Region-based Edge Computing

While Region-based Edge Computing represents an ideal EC architecture, it also presents challenges that must be addressed.

2.4.1 Intra-Region Security

Security is critically important for EC systems that provide services based on network proximity to users. While clearly separating regions addresses availability and confidentiality issues stemming from dependence on external services like CSPs, security concerns within individual regions remain unresolved. According to Kong et al. [20], security challenges in EC systems can be classified as follows. Similar issues have also been identified in Fog Computing [21].

- **DDoS Attacks:** Malicious actors can overwhelm resource-constrained edge nodes or devices with excessive requests, leading to service denial.
- **Eavesdropping and Sniffing:** Attackers can intercept network traffic between devices, the edge, and the cloud, potentially exposing sensitive data.
- **Malicious Hardware/Software Injection:** Compromised devices or software can be introduced into the network to launch further attacks or exfiltrate data.

- **Unauthorized Access:** Lack of strong authentication and authorization mechanisms can allow attackers to gain control of devices or edge nodes.
- **Privacy Leakage:** The personal data collected by IoT devices can be leaked, violating user privacy.

While these threats exist in standard enterprise networks, they become particularly severe in EC environments—especially those containing IoT devices. Unlike standard PCs and servers, IoT devices may not receive continuous firmware updates. Even when vulnerabilities are discovered, users may be unable to perform updates to address them. Furthermore, IoT devices often have severely constrained resources, making it difficult to implement advanced security features such as antivirus software or firewalls. Additionally, some IoT devices may contain built-in backdoors—either unintentionally exposed management consoles or, in some cases, deliberately installed by companies to collect user data without consent [9, 10]. Combined with these vulnerabilities, IoT devices become highly attractive targets for attackers. As a result, they not only become targets for DDoS attacks but also serve as vehicles for injecting malicious software into edge nodes, enabling network eavesdropping or acting as launchpads for attacks on other networks. Unlike cloud services, IoT devices can directly observe users’ private spaces, creating additional risks of sensitive personal information being compromised.

Implementing additional security mechanisms on resource-constrained devices or requiring continuous firmware updates from manufacturers is sometimes impractical. Therefore, I propose enhancing security through Network Access Control, which regulates device-to-device and external communications at the network layer. By controlling device communications via OpenFlow, it is possible to mitigate the impact of vulnerable or malicious devices on others. OpenFlow [22] is a protocol for controlling network devices, enabling dynamic modification of flow processing by adding or removing flow rules. This approach allows for granular flow-level control that was previously impossible with conventional network devices. Various approaches for improving network security using OpenFlow already exist, including Intrusion Detection Systems (IDS) [23], Firewalls (FW) [24], and strict control over device-to-device communications in enterprise networks (Ethane) [25]. However, end users—who act as region administrators—are not network specialists, making direct application of expert-oriented policies (such as Ethane’s) difficult. While outsourcing management to external providers [26] is one option, the primary objective of a Cloud-Independent Edge Computing architecture is to enable device management within regions without reliance on CSPs. Therefore, I propose an authorization mechanism based on a capability model—designed for end-user

manageability—along with corresponding network control methods. Details of this proposal are discussed in Chapter 3.

2.4.2 Management and Integration by Platform Providers

The primary advantages of CSP platforms are centralized IoT device management and seamless integration with cloud services. Even within Region-based EC, the system must provide similar convenience without relying on external services such as CSPs. To achieve this, a platform is required that supports both IoT device management and cloud service integration. However, since target devices and use cases are diverse, a flexible architecture is needed. I define such integration components as software deployed on computational nodes within the region, enabling flexible management and integration. When devices lack the required functionality (e.g., software components necessary for external collaboration), additional software can be deployed to supplement them. In other words, computational nodes serve as gateways to external systems, including other regions; similar approaches have been proposed as middleware frameworks, such as OSGi [27]. However, further discussion is required regarding how to accommodate multiple such software components while ensuring secure deployment. I discuss these issues in Section 2.4.3.

2.4.3 Computation at the Resource Constrained Edge Nodes

The delegation of processing tasks between network nodes has been explored since the early days of distributed operating systems. While distributed OSes primarily targeted high-performance workstations, the 1990s saw the emergence of concepts known as “Mobile Code” [28]. Specialized programming languages like Java [29] and Obliq [30] were developed for this purpose [31], with the goal of enhancing functionality by distributing and executing code on client machines. Language Virtual Machines (Language VMs) represent a lightweight isolation technology, typically implemented through managed runtimes such as the Java Virtual Machine (JVM). However, a significant limitation has been that tasks must be written in a specialized language, and virtual machine execution often incurs performance penalties compared to native code. Another approach, Software Fault Isolation (SFI) [32], utilizes software-based techniques for isolation. This approach isolates untrusted code by inserting runtime checks and statically verifying code correctness to restrict memory access. Systems such as Omniware [33] and Native Client (NaCl) [34] employed SFI to enable the secure execution of native code. However, this method

typically requires code rewriting to limit memory access and necessitates a dedicated toolchain or runtime. WebAssembly [35] leverages SFI to allow programs written in languages such as Rust or Go to be compiled into WebAssembly modules. It achieves near-native performance while enabling portability. However, WebAssembly does not yet fully support all processing requirements, particularly those reliant on standard OS process models.

Meanwhile, the emergence of virtualization technology on x86 PCs introduced an alternative process delegation mechanism. Starting with Xen [36], virtual machine (VM) technology—previously limited to workstations—became established, enabling OS-level isolation by treating entire operating systems as units deployable on separate nodes. In EC environments, approaches like Cloudlet [37] propose utilizing virtualization for processing on edge nodes. Since VMs inherently isolate entire operating systems, they offer strong isolation with minimal host dependencies, though they have been criticized for performance overhead [38]. Advancements in hardware support and dedicated OSes have significantly improved I/O performance [39, 40], allowing VMs to approach the performance of physical machines. However, since each VM runs its own kernel rather than sharing the host OS kernel, high resource consumption remains an inherent drawback [41].

Container technology provides a mechanism for isolating multiple environments on a shared host kernel. Unlike VMs, which require guest OS kernels, container technology achieves lightweight isolation by eliminating the need for separate kernels. Container technology was originally conceptualized as a mechanism for resource isolation at the OS process level [42]. Implementations such as FreeBSD Jail [43], OpenVZ [44], and Solaris Zones [45] emerged, followed by widespread adoption through LXC [46] and Docker [47]. The primary advantage of containers is that they require neither guest OSes nor kernels, resulting in lightweight isolation and minimal performance overhead [48]. Furthermore, by packaging application execution environments as images (e.g., Docker), they enable process portability independent of host-side libraries. One limitation is that, because container-based isolation shares the host kernel, vulnerabilities in the kernel may compromise isolation. Therefore, while container technology does not offer isolation robustness equivalent to VMs, it provides lightweight isolation that enables practical process portability.

Table 2.1 compares these various approaches. Language VM and SFI approaches offer strong and lightweight isolation, but they often mandate dedicated languages or toolchains, limiting compatibility with legacy applications. WebAssembly addresses this by targeting general-purpose languages. This allows programs written in Rust,

Tab. 2.1: Comparison with offloading approaches

	Isolation	Performance	Footprint	Legacy Compatibility
Language VM	Strong	Medium	Small	Low (dedicated languages)
SFI	Strong	High	Small	Low (dedicated toolchains)
Virtual Machine	Strong	High (Comp.) Medium (I/O)	Large	High
Container	Moderate	High	Medium	Medium (depends on Host)

Go, or C to be compiled into modules and executed anywhere a WebAssembly runtime exists. Although thread support [49] is being standardized, WebAssembly still faces challenges in supporting standard OS process models and concurrency. Since leveraging existing applications is critical for reducing development and maintenance costs, approaches supporting legacy applications are preferred in Region-based EC, rendering Language VM and SFI approaches less suitable.

In contrast, VMs offer robust isolation between guest operating systems and high compatibility. However, since each VM runs a complete guest OS, it consumes significant memory and storage resources even for simple operations. While lightweight solutions like MicroVMs [41] and Unikernels [50, 51] have been proposed, they often compromise the benefit of leveraging existing guest operating systems. Despite I/O performance improvements from para-virtualization technologies like virtio [52], eliminating the inherent overhead of virtualization mechanisms remains difficult [41]. Furthermore, regarding device resource sharing, VMs typically require dedicated device assignment (e.g., GPUs via PCI Passthrough [53]), making resource utilization less efficient than in other approaches. Consequently, in Region-based EC, which often operates in resource-constrained environments, high-overhead approaches like VMs are less suitable despite their strong isolation.

Container technology is characterized by process isolation based on the host kernel. While it does not provide isolation as robust as VMs, it incurs lower overhead and offers more efficient resource utilization, particularly in terms of memory. Therefore, deploying processing tasks using container technology is the most suitable approach for Region-based EC. However, host kernel vulnerabilities can compromise isolation. Similarly, vulnerabilities in the container runtime may lead to security breaches. Addressing these security risks requires implementing appropriate countermeasures. Resolving these issues is essential for ensuring secure operation in Region-based EC.

2.4.4 Issues for Container-based Edge Computing

While container-based Edge Computing leverages lightweight characteristics as a key advantage, it also presents several significant challenges.

First, isolation breaches may occur due to vulnerabilities in the kernel or runtime environment. Multiple vulnerabilities [54, 55, 56, 57] have been reported, making container security an ongoing concern. One approach to addressing these challenges is to use rootless containers [58]. This method restricts the privileges of processes running within containers by executing them as non-privileged users, thereby limiting access to the host kernel and minimizing the impact if vulnerabilities are exploited. However, a major drawback of rootless containers is performance degradation in network operations as a trade-off for non-privileged execution. I propose techniques to mitigate this issue and demonstrate performance improvements in real-world applications. Details are provided in Chapter 4.

Next, I address orchestration challenges [59] inherent in container-based Edge Computing. Effective management of multiple containers requires robust orchestration mechanisms. Container orchestration provides automated solutions for deployment, scaling, and maintenance, with Kubernetes [60] being one of the most widely adopted tools. KubeEdge [61], an extension of Kubernetes designed for EC, has also been developed. Within Region-based EC, orchestration mechanisms are necessary to manage containers locally without reliance on external services like CSPs. KubeEdge is also designed to operate standalone at the edge [62]. In this dissertation, I assume management using existing orchestration tools.

Another critical issue involves distribution of container images [63]. Container images package complete runtime environments, often containing all required libraries and dependencies, resulting in substantial size. Consequently, distributing images to edge nodes may be constrained by network bandwidth and storage capacity. Techniques like Lazy Pulling [64] accelerate container startup but do not reduce image size. Machine learning libraries frequently require extremely large images, necessitating special consideration. Thus, the cost associated with image transfer can become a barrier to utilizing updated or feature-rich images. I propose a method that achieves significant reductions in data transfer volume as well as performance improvements specifically for container image update scenarios. Detailed explanations are provided in Chapter 5.

2.4.5 Collaboration While Privacy-Preserving Data Analysis

The fundamental principle of Region-based Edge Computing is to complete all data processing within the defined Region. However, in practical scenarios, there may be unavoidable situations where external services (e.g., LLMs) must be utilized. In such cases, the critical requirement becomes “data processing while preserving privacy.” The RBTA model achieves privacy protection by preventing transmission of sensitive data outside the Region. Therefore, when data must be transmitted externally, protective mechanisms become essential.

EC systems implement various privacy protection measures, which can be broadly categorized into three approaches: Cryptographic Schemes, Secure Data Aggregation, and Trusted Execution Environments (TEEs).

Cryptographic Schemes involve encrypting data and performing operations such as search [6] and arithmetic on the encrypted form. While each specific operation historically required its own cryptographic scheme, fully homomorphic encryption [7] has been proposed to enable general-purpose processing on encrypted data. However, fully homomorphic encryption incurs significant computational overhead, making it impractical for many real-world applications.

Secure Data Aggregation refers to techniques that either obscure individual data points through aggregation or introduce statistical noise to make them unidentifiable. These methods consider data distribution statistically to prevent individual identification. Representative techniques include differential privacy [65]. As with other approaches, these methods often require design tailored to specific processing needs, making general-purpose use challenging.

TEEs provide isolated processing environments using hardware-based security features such as Intel SGX. The isolated environment is inaccessible from outside, and memory is encrypted, preventing access even from the host OS. While TEEs incur overhead for memory encryption, they enable general-purpose computation with performance near standard environments. However, a major limitation is that although TEEs protect the *process* from the host, they do not inherently protect data from malicious *analysis queries*. An analyst could submit queries designed to leak raw data (e.g., `SELECT * FROM users LIMIT 1`). To ensure privacy protection, a mechanism to control such information-leaking queries is necessary.

I propose a privacy-preserving data analysis platform based on TEEs. Furthermore, I implement a mechanism that allows data providers to control analysis queries. Detailed implementation methods are presented in Chapter 6.

Network Access Control with User-friendly Authorization

This chapter is based on a paper “CREBAS: Enabling Network Access Control in a Home with One Click” published in Journal of Information Processing, Vol.31, 2023 [66], ©2023 Information Processing Society of Japan.

3.1 Introduction

Home networks include not only conventional devices such as PCs but also smart home devices such as smart speakers and smart TVs. These smart home devices cooperate over the home network and allow users to control and check the status of home appliances using handheld devices or voice commands. Currently, most devices interoperate with external systems in the cloud, but more devices will cooperate via direct communications [67] within the home network.

Conventional home networks lack mechanisms to control communications between devices or provide only simple, limited security. Devices can communicate with each other without restrictions. Therefore, all devices in a home network are exposed to the risk of attacks or unintended communications from malicious devices. Malicious devices can scan a home network and intercept private information. In addition, such devices can operate other devices and cause unintended behavior from the user’s perspective. As a result, insufficient security in a home network can cause not only leakage of private information but also serious consequences that affect users’ lives.

In the case of cooperation via direct communications, not all devices support access control, and access control mechanisms can be limited by software or resource constraints. In fact, some communication protocols [68, 69] are not designed to perform authentication or access control. As a result, attackers and malicious devices can easily exploit devices and steal information from them. As more devices and protocols are used in modern home networks, these security problems will become more serious.

To protect devices from attacks, network access control methods [70, 71] have been proposed. These methods control the network using OpenFlow [72]. Although they can protect devices from malicious or unintended attacks, they rely on flow-based access control and cannot govern messages transferred within TCP streams. This means such methods can purge completely malicious devices but cannot prevent otherwise legitimate devices from scanning the network and covertly stealing information.

Prior works have shown that network access control can mitigate vulnerabilities, but they have not examined authorization architectures for non-expert users. An authorization architecture is the system by which users configure access control. Without a properly designed authorization architecture, users may struggle to configure each device appropriately or to verify that access control is applied as intended. Users can make incorrect configurations that allow malicious communications. Therefore, no matter how capable a network access control method is, it is ineffective if the authorization architecture is not properly designed. An appropriate authorization architecture is essential, especially in home environments.

Some research on usable access control and authorization architectures for the home [73, 74, 75, 76, 77, 78, 79] has been conducted. This body of work considers requirements for more usable authorization architectures in various home environments. To apply these insights to network access control, I address several issues. Home networks include not only devices that support new access control methods but also existing devices that do not. Network access control can govern both without modifications to devices, but authorization architectures often require device support for their protocols. Furthermore, there is a gap in the granularity of control between network access control and authorization architectures. Authorization architectures provide control per operation or content, but, as I mentioned, flow-based network access control cannot achieve such fine granularity. I address these problems by jointly designing an authorization architecture and network access control methods.

In this chapter, I propose CREBAS, which enables network access control in a home with one click. Fig. 3.1 provides an overview of CREBAS. CREBAS aims to control each communication at the level of operations, such as `Get Temperature` or `Turn on Light`. Providing fine-grained access control increases complexity and the burden on users. To reduce this burden, CREBAS employs an authorization architecture based on the capability model. This enables users to achieve fine-grained authorization easily and allows network access control to manage communications at the operation level within a home using the authorized capability.

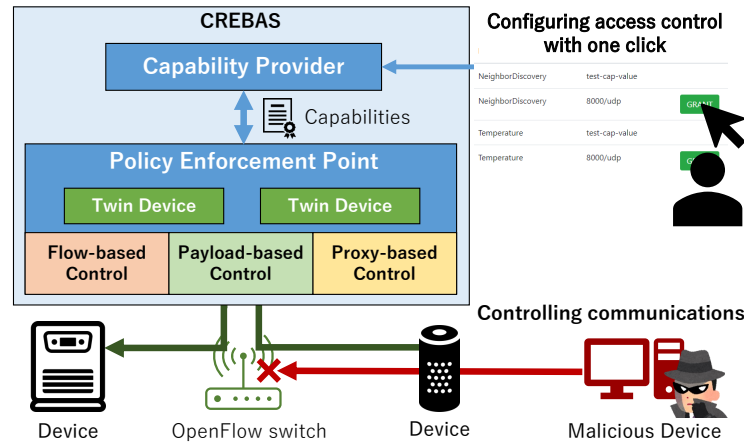


Fig. 3.1: Overview of CREBAS

First, I propose an authorization architecture based on the capability model. I designed the authorization architecture according to the guidelines [74]. The capability model authorizes access to each device's communications or functions and issues a token called a capability. Capability tokens prove rights to use or access the specified resources. This model achieves fine-grained authorization and is suitable for home networks where various devices exist. Capabilities are assigned to each device's communications, and users authorize them to control communications. Users are not required to investigate device specifications or write policies such as ACLs. However, it may require frequent authorization operations, which burdens users. To reduce the number of authorization requests to users, the authorization architecture incorporates Automatic Authorization. Automatic Authorization makes authorization decisions automatically on behalf of the user.

Secondly, I propose network access control methods that control communications according to the authorized capabilities. These methods enable more fine-grained access control through flow-based, payload-based, and proxy-based control. Existing devices cannot handle capabilities and cannot inspect packet payloads. To solve this problem, I designed a twin device that complements such missing functions. The twin device enables the proposed methods to integrate into a home network without major modifications to existing devices.

Then, I implemented a prototype of CREBAS. I confirmed that the performance degradation is acceptable and that it is possible to control communications finely among smart home devices.

Finally, I discuss the usability of the authorization architecture.

Tab. 3.1: Comparison with other network access control methods

	(1) Configured by users	(2) Independent system	(3) Flow-based control	(4) Message -based control	(5) Discussion of usability
Feamster [26]			✓		
Moyano et al. [80]			✓		
Osman et al. [81]			✓		
Mortier et al. [70]	✓	✓	✓		
Lear et al. [71]	✓		✓		
CREBAS	✓	✓	✓	✓	✓

3.2 Related Work

3.2.1 Network Access Control in a Home

Table 3.1 shows the comparison with other network access control methods.

Feamster [26] showed that it is possible to detect spam email and botnets by outsourcing the management of home networks using OpenFlow. OpenFlow can provide flow-based access control from a remote controller. Moyano et al. [80] designed a platform to control home networks with NFV-based applications. Osman [81] proposed an automatic control system. The system analyzes devices in a home network with an analysis system built on a cloud and separates the network automatically using OpenFlow. These studies propose delegating network management from users who lack networking expertise to external experts or automated mechanisms. In these studies, the authors argue that a fully automated system is easy to use because it eliminates user burden. However, according to Yao's work [73], users are reluctant to delegate the management of devices in the home to an external party. A system in which users do not have control is not desirable. A mechanism or interface for (1) Configured by users is necessary, but it is not discussed. Additionally, these methods have to send information about each individual's device and network to the external system, which causes privacy problems. In addition, if the external system that manages the network becomes unavailable, the home network also becomes unavailable.

On the other hand, there are some (2) Independent systems without outsourcing. These systems have no privacy and availability issues. Mortier [70] proposed a system to manage devices in a home network by integrating routers. Manufacturer Usage Description (MUD) [71] enables vendors to define specifications of communications and access control policies for each IoT device and apply them automatically.

MUD controls communications among devices so that MUD can protect devices even if devices have vulnerabilities.

Outsourcing and standalone methods both use only (3) Flow-based control. Flow-based control has a limitation in terms of the granularity of control. If a device performs both authorized and malicious communications in the same TCP stream, flow-based control methods cannot control it properly.

Although these methods demonstrate the technical feasibility of controlling home networks, they did not discuss or experiment with their usability. The discussion about usability is important because the methods are applied to real home networks and users. If users make incorrect configurations, access control will not work. There are some works on access control for ordinary users in a home [73, 74, 75, 76, 77, 78, 79]. (5) Discussion of usability should be performed to design a more usable system and evaluate its validity.

In this chapter, I propose CREBAS. CREBAS is an independent system that does not delegate control to external parties and consists of authorization architecture and network access control methods. The authorization architecture is designed based on the guidelines [74] for access control in a home network, and I discuss the validity in terms of its usability. Users can configure the access control with one click on the CREBAS WebUI. To provide more fine-grained network access control, I propose (4) Message-based control methods called *payload-based* and *proxy-based* access control. These methods allow authorized messages while blocking malicious ones within the same TCP stream, flow-based control cannot achieve.

3.2.2 Capability-based Access Control

The capability model [82] is a kind of access control model. In this model, a device issues a capability to a user, and the user can use the corresponding function by presenting the capability to the device. Gusmeroli [83] proposed access control for IoT devices based on the capability model. In the case of Role-Based Access Control and Attribute-Based Access Control, there are problems such as role explosion and difficulty in ensuring attribute consistency. Capability-Based Access Control (CapBAC) solves these problems and manages access for each device easily. Additionally, by delegating capabilities to others, it is possible to authorize and issue delegated capabilities. However, these CapBAC methods cannot be applied to existing devices because they require devices to process capabilities and control access. Various devices are produced by various vendors and need to support the same CapBAC

method. CREBAS provides CapBAC without modifications to devices, including existing devices, by controlling communications on the network side.

3.3 Authorization Architecture

The authorization architecture provides a mechanism for users to authorize communications among devices.

3.3.1 The Guidelines for Access Control at Home

Mazurek worked out the guidelines [74] shown below for access control systems aimed at home users.

1. Allow fine-grained control
2. Plan for lending devices
3. Include reactive policy creation [75]
4. Include logs
5. Reduce or eliminate up-front complexity
6. Support iterative policy specification

I propose the authorization architecture based on the capability model to meet these guidelines. Conventional policy-based access control makes it difficult for users to achieve these guidelines. I discuss this in more detail in Section 3.6.

3.3.2 Overview of the Authorization Architecture

The authorization architecture is designed based on the capability model. In this architecture, each communication is represented as a capability. A capability represents the purpose and use of communication in a way that users can easily understand. For a device to communicate with other devices, an appropriate capability must be issued to the device. This architecture has two authorization flows, *Automatic Authorization* and *Manual Authorization*. Under either authorization flow, capabilities are authorized and issued to devices.

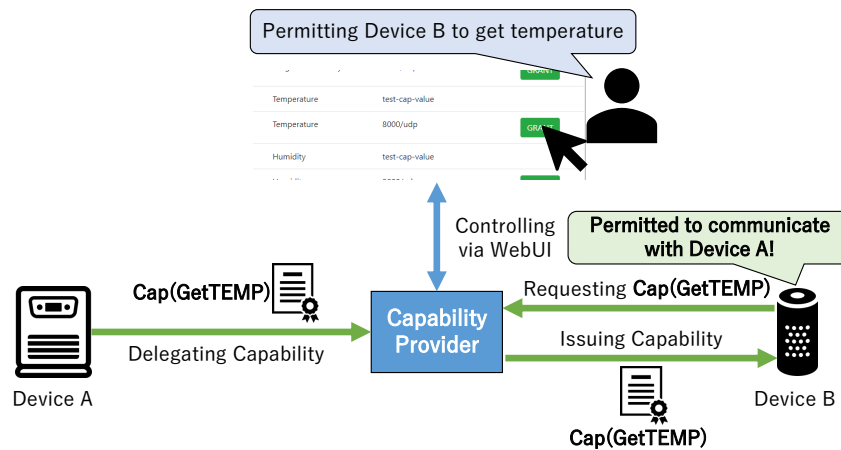


Fig. 3.2: Overview of the authorization architecture

```
{
  "capabilityID": "004bb012-3e39...",
  "assignerID": "ef4bfdac-5760...",
  "assigneeID": "23c186a6-8ec9...",
  "capabilityName": "GetTemperature",
  "capabilityValue": "80/tcp",
  "authorizeCapabilityID": "e1ee77ff...",
  // encoded authorization policy
  "capabilityGrantPolicy": "ehoIeLw...",
  "capabilitySignature": {
    "signerID": "ef4bfdac-5760...",
    "signeeID": "23c186a6-8ec9...",
    "signature": "gI+wYKnYxN/XI0wWh..."
  },
  ...
}
```

Fig. 3.3: Example capability

In the context of Zero Trust Architecture [84], this component functions as a policy decision point (PDP) that reflects the user's access-control intentions. More specifically, the authorization algorithm serves as the "Policy Engine": it decides whether to authorize a request under either Automatic Authorization or Manual Authorization. The capability issuance and delegation functions correspond to a "Policy Administrator". In this architecture, both the Policy Engine and the Policy Administrator are implemented as the Capability Provider.

Fig. 3.2 provides an overview of the authorization architecture. Each device has capabilities shown in Fig. 3.3. `capabilityName` represents the purpose and use of the communication. Users check this field and decide whether to authorize the capability or not. `capabilityGrantPolicy` is an encoded authorization policy used in Automatic Authorization. `capabilitySignature` is a signature created by a device that provides the capability when it delegates the capability. Capabilities are also signed

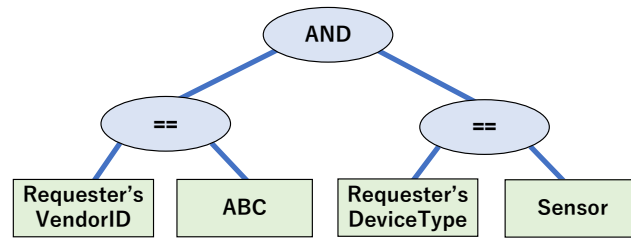


Fig. 3.4: Example binary expression tree

when the Capability Provider (CP) issues them to devices or delegates them to users. Devices and the CP check the signature when they apply the capability.

Capability Provider (CP) manages and authorizes device capabilities. All device capabilities are delegated to the CP. The CP issues capabilities to devices that request them when they are authorized with Automatic Authorization. When capabilities are authorized with Manual Authorization, the CP delegates them to users. Users can check the status of the authorizations and authorize capabilities manually using the WebUI implemented in CP.

Each device must have functionality to process capabilities. Most devices do not implement such functionality. Moreover, it is difficult to add this functionality to existing devices. To solve this problem, I use a *twin device*, which is described in Section 3.4.

3.3.3 Automatic Authorization

In the capability model, the number of requests to authorize a capability can increase because all communications must be authorized by users. To reduce the number of requests, the CP has a mechanism called *Automatic Authorization* using capability delegation. In Automatic Authorization, capabilities are authorized and issued automatically without the user's operations. Capabilities have a built-in authorization policy for Automatic Authorization. An authorization policy decides whether to authorize based on the requester's information. When a device requests a capability, the CP checks the authorization policy in a capability and authorizes it automatically or delegates the capability to a user for Manual Authorization.

An automatic authorization policy uses a Binary Expression Tree to represent a conditional expression. An example tree shown in Fig. 3.4 represents condition $(\text{Requester's VendorID} == \text{ABC}) \text{ AND } (\text{Requester's DeviceType} == \text{Sensor})$. The CP decodes the authorization policy and checks the requester's attributes to determine

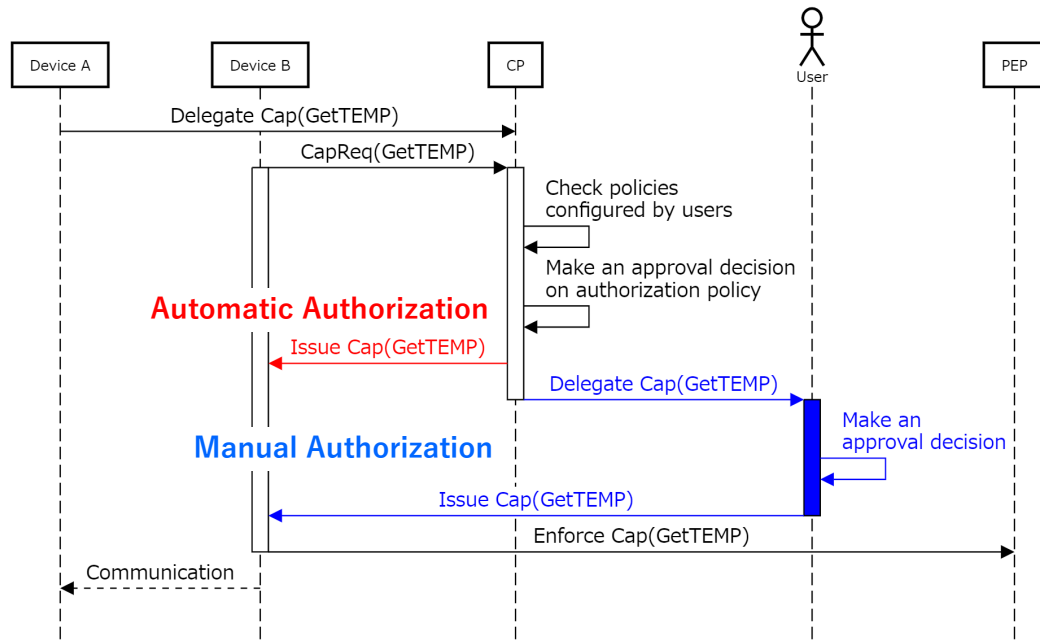


Fig. 3.5: Flows of Authorization

whether the requester meets the condition. If the requester meets the condition, the CP authorizes and issues the capability to the requester.

Some malicious devices may exploit Automatic Authorization, and capabilities can be issued unintentionally. To prevent exploitation, users can override Automatic Authorization by configuring policies not to authorize capabilities automatically. The policy can specify which capabilities and which requests from devices are not automatically authorized.

3.3.4 Flows of Authorization

Automatic Authorization authorizes and issues capabilities without the user's intervention. Manual Authorization treats capability requests that are not handled by Automatic Authorization. Users must authorize and issue capabilities themselves.

As an example, I define the capability required to obtain the temperature from the device as `Cap(GetTEMP)`. Furthermore, I define the capability request for `Cap(GetTEMP)` as `CapReq(GetTEMP)`.

A) Automatic Authorization

The flow of Automatic Authorization is as shown in Fig. 3.5. Device A delegates Cap(GetTEMP) to the CP, and Device B requests it with CapReq(GetTEMP). Cap(GetTEMP) has an authorization policy that allows it to be authorized when requested by Device B. The CP automatically issues Cap(GetTEMP) to Device B.

B) Manual Authorization

The flow of Manual Authorization is as shown in Fig. 3.5. Cap(GetTEMP)'s authorization policy does not permit it to be authorized automatically. Device B requests it with CapReq(GetTEMP), but Automatic Authorization fails. The CP delegates Cap(GetTEMP) to a user and asks whether the user authorizes it. If the user authorizes, the user issues Cap(GetTEMP) to Device B. In Manual Authorization, capabilities are first delegated to the CP and redelegated to users. Capabilities are issued by users, so it is easy to check who authorized the capabilities.

3.4 Network Access Control

3.4.1 Overview of Network Access Control

The objective of network access control is to control communications between devices according to authorized capabilities. Previous works [26, 80, 81, 70, 71] used flow-based control with OpenFlow. However, the granularity of flow-based control is limited to *flow*. This limitation allows malicious devices to bypass access control. For example, when a device is allowed to send messages via the same TCP stream, the device can send both authorized and malicious messages. This is because flow-based control relies on 5-tuples in a packet, so it cannot distinguish between authorized and malicious messages transferred via TCP streams. I propose three access control methods: flow-based, payload-based, and proxy-based access control. These methods enable more fine-grained and sufficient granularity in access control. They can even control messages transmitted through the same TCP stream.

I designed the Policy Enforcement Point (PEP) to control communications between devices with three access control methods. Fig. 3.6 provides an overview of the PEP. A Controller controls components in PEP and an external OpenFlow switch. OVS for access control is Open vSwitch (OVS) in the PEP. It forwards packets to an intended twin device or an OpenFlow switch. Twin device is software to process capabilities and provides proxy-based and payload-based control on behalf of a

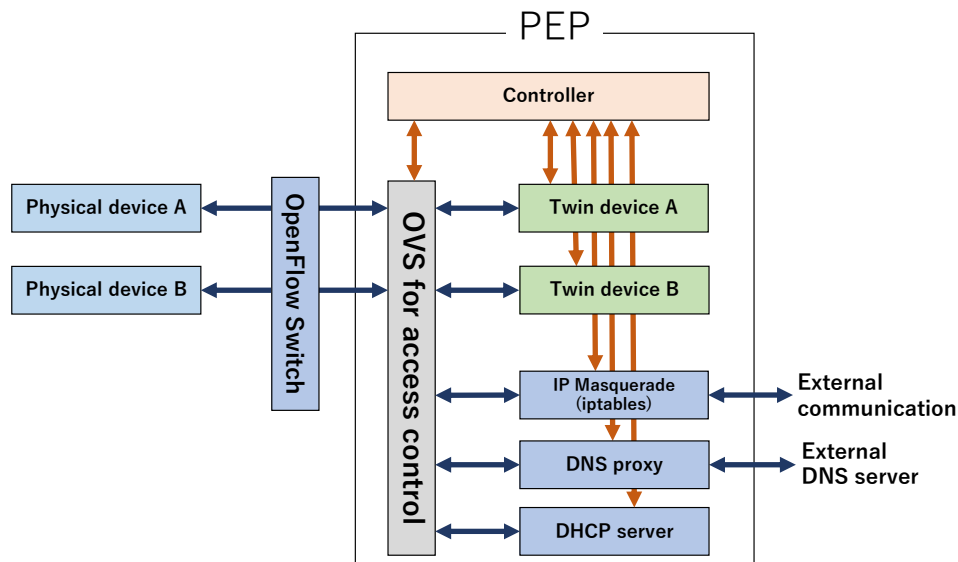


Fig. 3.6: Overview of the PEP

physical device. DHCP server is used to manage IP addresses for physical devices and configure twin devices. IP Masquerade and DNS proxy are used to control external communication. CREBAS provides domain-based access control for external communications. Capabilities used for control of external communications have a field like `*.aaa.example.com` to specify domains. If a capability is authorized, the controller configures IP Masquerade to allow communications to the domain's IP addresses. The endpoint IP addresses can vary, and the system needs to know the relationship between domains and IP addresses. DNS proxy handles DNS queries and reports this mapping to the controller.

Some malicious devices may attack these components to disable access control. To prevent such attacks, OVS for access control controls all flows of the components. The DHCP server, DNS proxy, and devices connected to the PEP host are strictly controlled, and only the minimum required communications among them are allowed.

3.4.2 A Twin Device

A twin device processes capabilities and provides access control on behalf of physical devices. Existing devices cannot handle capabilities and do not implement access control methods. A twin device fills the missing functions in these devices.

A twin device is implemented as a standalone process (binary). The controller sends authorized capabilities for a physical device to the corresponding twin device. A twin device recognizes received capabilities and checks whether UDP packets and TCP

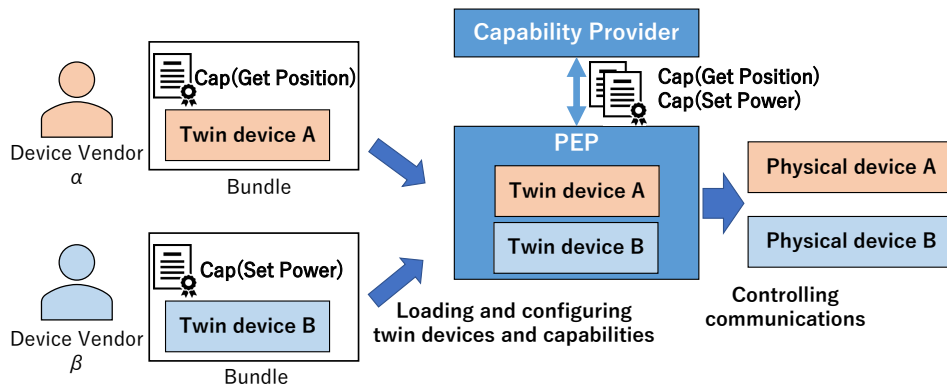


Fig. 3.7: Loading twin devices

messages from other devices are allowed or not. The code for a twin device is written in general-purpose programming languages, with no limitations on expressiveness. Therefore, twin device developers can define rules and filters as needed.

Each twin device is separated from the PEP host and other twin devices by Linux network namespaces (netns). A twin device has a veth pair connecting its netns and OVS for access control. This veth device is used to process packets controlled with payload-based or proxy-based methods. These access control methods are applied based on the capability. If a capability is defined for flow-based control, the controller configures OpenFlow switches to allow corresponding flows. If a capability is defined for payload-based control, the controller configures switches to forward the corresponding packets to the twin device. If a capability is defined for proxy-based control, the controller configures switches to forward and rewrite headers for the TCP transparent proxy in a twin device. Twin devices can receive and send only packets related to their devices.

Device vendors can control their own protocols by developing their own twin devices. As shown in Fig. 3.7, PEP can provide access control for vendors' original protocols by simply loading and configuring twin devices.

PEP needs to configure twin devices to control access. Using a dedicated DHCP server, PEP automatically configures a twin device. Users are not required to configure the twin devices themselves. A physical device sends a DHCP request. ClientIdentifier or hardware address in the DHCP request identifies the corresponding twin device. The DHCP server instructs PEP to configure a twin device. PEP identifies and configures a twin device based on device's hardware address or the ClientIdentifier.

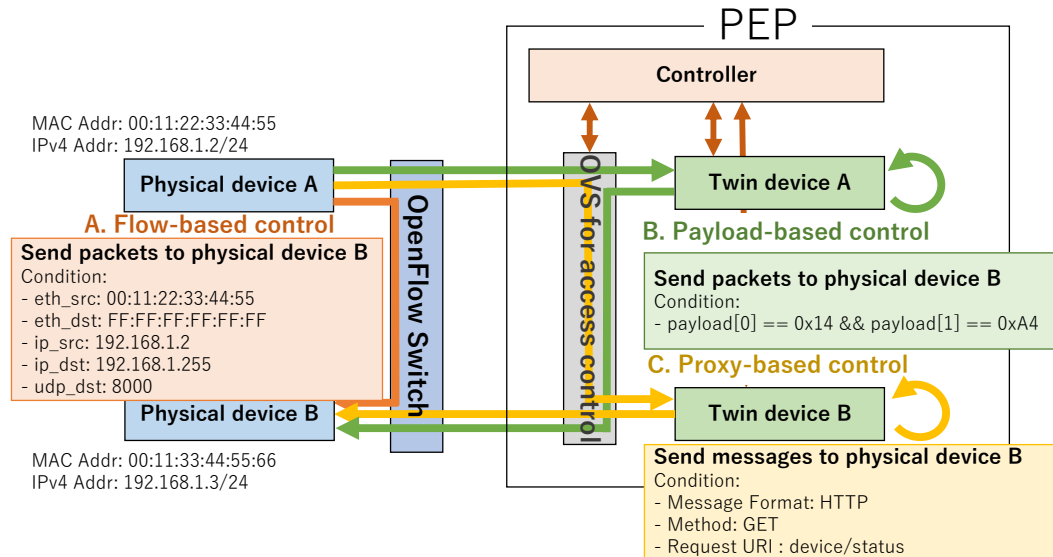


Fig. 3.8: Access control methods

3.4.3 Access Control Methods

Fig. 3.8 shows the proposed flow-based, payload-based, and proxy-based control methods.

A) Flow-based Control Method

Flow-based control uses OpenFlow to control flows. In this work, devices are connected to an OpenFlow switch or an OpenFlow wireless access point. If an appropriate capability is presented, the controller in the PEP installs flow rules in an OpenFlow switch to allow the corresponding communications. The devices can perform communications related to the capability, but other communications are denied.

However, as mentioned above, this method cannot protect the device if a malicious device performs both authorized and unauthorized communications to the same port on the target device. The following two methods solve this problem.

B) Payload-based Control Method

Payload-based control inspects packet payloads. Some protocols, such as UPnP [85] and SSDP, use a single UDP packet to discover devices in a network. For example, a situation can occur where some services are permitted to discover devices, but other services are not permitted. Access control is based on authorized capabilities, and a method is needed to control these discoveries. In payload-based control,

an OpenFlow switch directs packets to twin devices. The twin devices inspect the payload of the packet and pass or discard packets. In the implementation, a twin device performs `recv(2)` and `send(2)` with a raw socket. A twin device parses packets and passes or discards them according to the authorized capabilities. Some packets are broadcast, and it is difficult to control them in a network using an OpenFlow switch. Therefore, payload-based control inspects and controls the *response* packet. If the response packet is discarded, the device is not discovered.

C) Proxy-based Control Method

Proxy-based control inspects messages transferred by TCP. HTTP, which uses TCP, is often used for communication between devices. For example, some protocols [69, 86] are defined using the HTTP REST API. In the REST API, each operation is represented with an HTTP method and a URL, and capabilities are assigned to each operation. Messages, including HTTP requests and responses, need to be controlled according to authorized capabilities. Proxy-based control parses, checks, and passes or discards messages using a TCP proxy in twin devices.

An OpenFlow switch rewrites a packet's header to a temporary address to direct packets to the TCP proxy. The TCP proxy executes `recv(2)` to retrieve a message, checks it, and executes `send(2)` to pass it or discard it.

3.4.4 The Flow of Access Control

Fig. 3.9 shows the flow of access control. Physical devices and twin devices are configured with DHCP. Twin devices present a capability authorized by CP or a user to the controller in PEP. The controller adds flows generated from the capability to an OpenFlow switch. The controller then orders the twin devices to perform payload-based or proxy-based control according to the presented capability. Finally, authorized communications become available, and other communications are blocked.

3.4.5 A Trick for Forwarding Packets

In payload-based and proxy-based control, packets are forwarded to PEP. In Fig. 3.8, Physical device A (Device A) sends packets. These packets must be forwarded to a twin device in PEP. However, if an L2 switch exists between an OpenFlow switch and a PEP host, an L2 switch cannot forward packets to the PEP host correctly.

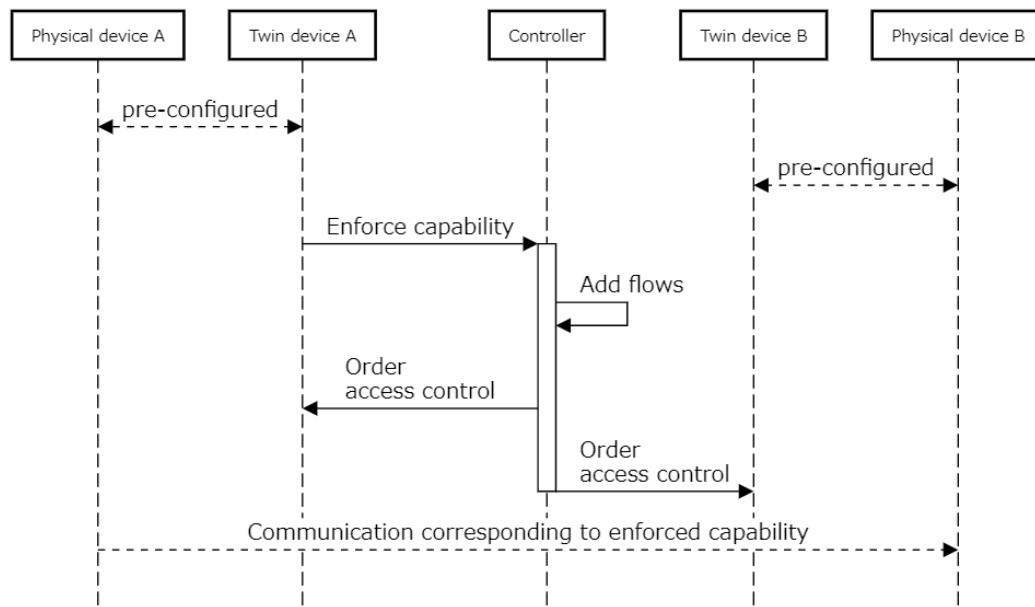


Fig. 3.9: Flow of network access control

Packets sent from Device A have Device A's Ethernet address in the source Ethernet address field. An L2 switch learns that Device A is connected to the port to which the OpenFlow switch is connected. Then, Physical device B (Device B) sends packets to Device A with Device A's Ethernet address in the destination Ethernet address field. The OpenFlow switch tries to forward packets to the PEP host. According to the learned Ethernet address table, an L2 switch forwards these packets to the port to which the OpenFlow switch is connected. As a result, packets from Device B to Device A are forwarded to Device A. Packets are not forwarded to the PEP host, and payload-based and proxy-based control do not work.

I solved this problem by rewriting a packet's Ethernet address. For packets from Device A to PEP, the OpenFlow switch rewrites the Ethernet Destination field from Device B's address to the PEP host's address. PEP knows the relationship between an IP address and a device's Ethernet address. OVS for access control rewrites the Ethernet Destination field from the PEP host's address to Device B's address based on the destination IP address. Then, packets are processed in PEP. For packets from PEP to Device B, the OVS for access control rewrites the Ethernet Source field from Device A's address to the PEP host's address. The OpenFlow switch rewrites the Ethernet Source field from the PEP host's address to Device A's address based on the source IP address. In this way, an L2 switch learns the ethernet addresses correctly and packets are forwarded to the PEP host.

Devices requested capability		Buttons to grant capability		
RequesterID	RequesteeID	RequestCapability	RequestCapabilityValue	Op
f7ad301d-b2ca-47b3-b934-a516e855c65f	fa1effd7-c56b-4a5e-850d-d86727ca82c4	NeighborDiscovery	test-cap-value	
15d06f04-2a8f-4c07-af46-9acde4e7e4ff	fa1effd7-c56b-4a5e-850d-d86727ca82c4	NeighborDiscovery	8000/udp	GRANT
f7ad301d-b2ca-47b3-b934-a516e855c65f	fa1effd7-c56b-4a5e-850d-d86727ca82c4	Temperature	test-cap-value	
15d06f04-2a8f-4c07-af46-9acde4e7e4ff	fa1effd7-c56b-4a5e-850d-d86727ca82c4	Temperature	8000/udp	GRANT
f7ad301d-b2ca-47b3-b934-a516e855c65f	fa1effd7-c56b-4a5e-850d-d86727ca82c4	Humidity	test-cap-value	
15d06f04-2a8f-4c07-af46-9acde4e7e4ff	fa1effd7-c56b-4a5e-850d-d86727ca82c4	Humidity	8000/udp	GRANT

Devices requesting capability		Requesting capabilityName					
AssignerID	AssigneeID	CapabilityName	CapabilityValue	AuthorizeCapabilityID	RequesterAttribute	GrantCondition	GrantValue
fa1effd7-c56b-4a5e-850d-d86727ca82c4	f7ad301d-b2ca-47b3-b934-a516e855c65f	NeighborDiscovery	8000/udp	830b5f67-cf07-45f8-8c40-d08f3986f8fd			
fa1effd7-c56b-4a5e-850d-d86727ca82c4	65c7c429-38aa-46a2-a667-237928cee256	Temperature	8000/udp	5b1e2b67-dbbb-4f7d-bc60-bd2a200cec3a			

Fig. 3.10: WebUI of the prototype implementation

3.5 Implementation and Evaluation

I implemented a prototype of CREBAS. Using the system, I measured performance in a real environment and confirmed that the methods incur limited performance overhead in a home network. Furthermore, I confirmed that CREBAS can provide more fine-grained access control than flow-based control methods in a real protocol.

3.5.1 Prototype Implementation

Fig. 3.10 shows the WebUI of the prototype implementation. All capability requests, pending authorization capabilities, and granted capabilities are listed. Users can authorize capabilities manually by simply checking CapabilityName and clicking the GRANT button. When the GRANT button is clicked, the CP processes the request and applies the capability to the PEP. Based on an applied capability, PEP configures an OpenFlow switch and twin devices. Access control is achieved with one click.

3.5.2 Performance Evaluation

In the performance evaluation, I used hostapd (a software access point), Allied Telesis AT-TQ4600 (an AP that supports OpenFlow 1.3), and Aruba AP11 (a low-cost AP for small businesses) with OpenWrt and Open vSwitch installed. For the PEP, a ThinkPad X260 (Core i5-6200U (2 cores/4 threads), 16 GB memory) was used. Two Raspberry Pi 4 devices were connected to the network, and each device had

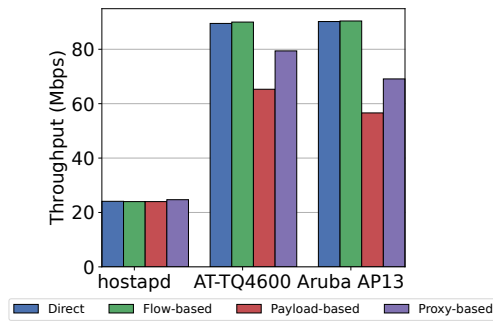


Fig. 3.11: Average throughput (iperf3)

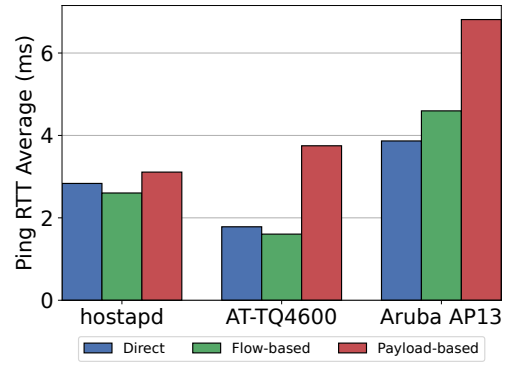


Fig. 3.12: Average RTT (ping)

one authorized capability. Bandwidth was measured using iperf3, and latency was measured using ping. Measured values are averages over 120 seconds.

The results of the measurements are shown in Fig. 3.11 and Fig. 3.12. Flow-based control achieved almost the same throughput as direct communication without control. Payload-based control achieved about 60 to 70% of direct communication. Proxy-based control achieved about 70 to 80% of direct communication.

The same tendency is observed for latency. The RTT of flow-based control is almost the same as direct communication. Payload-based control degraded RTT to 1.5 to 2 times that of direct communication. Proxy-based control is the same as payload-based control because ICMP packets in both methods go through the same path.

From these results, flow-based control is suitable for large data transfers such as video streams or latency-aware applications such as VoIP, and payload-based or proxy-based control is suitable for neighbor search and control messages like HTTP. Using the proper control method for each communication can minimize performance degradation. Additionally, the results show that the methods do not require expensive enterprise APs, and they also work well with low-cost APs. This lowers the barrier to introducing the methods in real home networks.

3.5.3 Resource Consumption in PEP

I also measured resource consumption in the PEP. CREBAS provides payload-based and proxy-based control via the PEP. Payload-based and proxy-based controls require more computational resources. I measured CPU usage in the PEP and confirmed that resource consumption is acceptable.

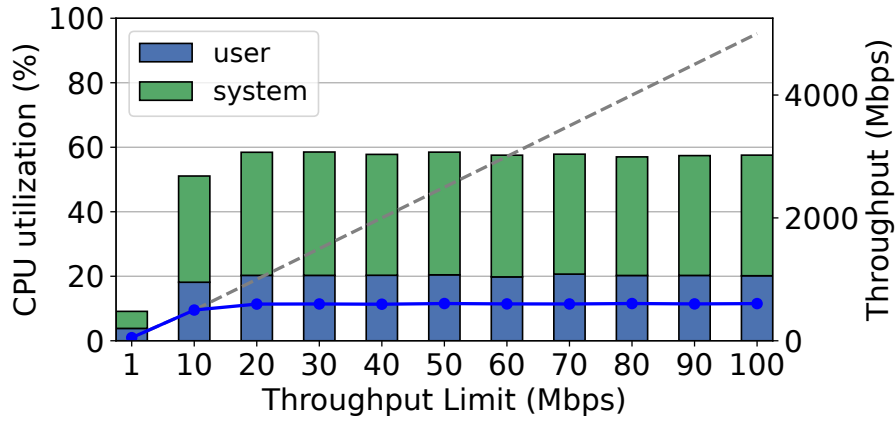


Fig. 3.13: CPU usage in payload-based control with iperf3
(A dotted line is ideal throughput)

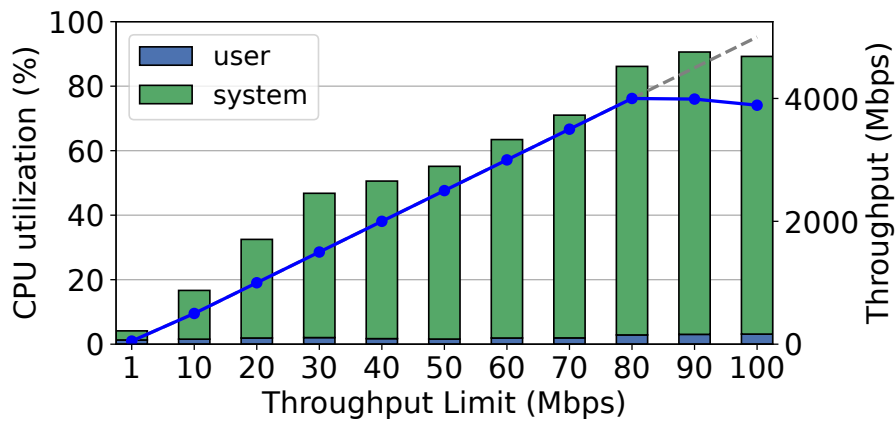


Fig. 3.14: CPU usage in proxy-based control with iperf3
(A dotted line is ideal throughput)

I conducted the measurements with the following scenario. PEP runs on a virtual machine (VM) with the same number of CPU cores and threads and the same memory capacity as in the performance experiments. Two VMs are connected to the PEP and run iperf3 and packet or message exchanges. Based on statistics [87, 88, 89, 90] about the number of devices in a home, in these measurements, I assume eight devices (four smart devices, two smartphones, one tablet, and one PC) as the load. From this assumption, if each device communicates in a full mesh, the number of simultaneous connections will be 28. Considering the increase in devices, I conducted the experiments with 50 parallel connections in iperf3.

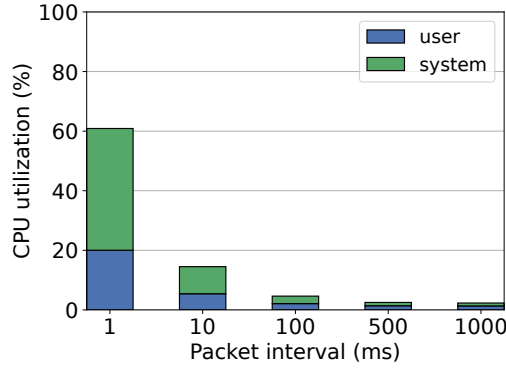


Fig. 3.15: CPU usage in payload-based control

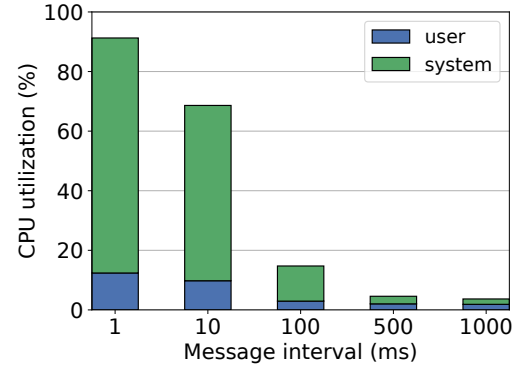


Fig. 3.16: CPU usage in proxy-based control

I conducted two experiments. One is iperf3 and the other is UDP packet or TCP message exchange. In the first experiment, I varied the rate limitation of each iperf3 connection and measured CPU usage. In the second experiment, I varied the interval of exchanging UDP packets or TCP messages. From the investigation of IoT device communications [91], I used 1 KiB UDP packets and 10 KiB TCP messages. This size setting covers most communications for IoT devices. The results are averaged over 60 seconds.

Fig. 3.13 and Fig. 3.14 show the sum of the throughput of iperf3 and the CPU usage in payload-based and proxy-based controls. In payload-based control (Fig. 3.13), payload-based control consumes about 50% to 60% CPU, and throughput does not scale. This happens because the payload-based control needs to handle, parse, and check all packets in a single thread. In proxy-based control (Fig. 3.14), CPU usage increases roughly linearly with throughput. This is because each TCP connection is handled in parallel in its own thread in proxy-based control. However, this control cannot adequately sustain connections above 80 Mbps.

Fig. 3.15 shows the CPU usage of the payload-based control with the exchange of UDP packets. CPU usage is less than 20% if the interval is longer than 10 ms. From the investigation [91], most devices send fewer than 100 UDP packets per second and less than 10 KiB/s. Therefore, payload-based control is expected to consume very few CPU resources.

Fig. 3.16 shows the CPU usage of the proxy-based control to exchange TCP messages. CPU usage is greater than 60% when the interval is shorter than 10 ms. This is because the TCP transparent proxy used in proxy-based control needs to establish connections for every message exchange. However, most devices rarely establish

TCP connections frequently [91]. Therefore, proxy-based control is also expected to consume very few CPU resources if used to control TCP message exchanges.

From these results, payload-based and proxy-based control will consume a lot of CPU resources when they are used in inappropriate ways, such as large data transfers. Such communications should be controlled with flow-based control. If payload-based and proxy-based controls are used to control communications that do not exchange large data, resource consumption is acceptable.

3.5.4 PoC with a Real Protocol

Attack Scenario

CREBAS aims to control communications among devices that use protocols lacking authentication and encryption. Because of the absence of authentication, attackers can easily perform malicious operations like stealing information and making devices malfunction. CREBAS assumes not only completely malicious attackers but also suspicious devices that work correctly but try to steal information covertly. Such suspicious devices cannot be purged because users want to use the devices, and users may not notice that devices are behaving incorrectly. CREBAS prevents attacks from such devices by controlling communications in units of operation.

As stated in Section 3.4, some protocols [69, 86] operate over the HTTP REST API. Previous works use flow-based control, so they can purge devices from the network, but they cannot block only malicious communications or operations. CREBAS prevents attacks from devices by controlling communications in units of operation using not only flow-based control but also payload-based and proxy-based control.

PoC with DIAL Protocol

I controlled the Discovery and Launch (DIAL) protocol [69] with the CREBAS prototype system. The DIAL protocol is based on the Simple Service Discovery Protocol (SSDP) and the DIAL REST Service. SSDP discovers devices in a network, and devices are controlled via HTTP REST API called the DIAL REST Service.

The DIAL protocol lacks access control mechanisms, allowing malicious devices to detect and control other devices in a home network. Flow-based access control can mitigate some of these problems, but it is insufficient. Flow-based control can only allow or deny device discovery as a whole. Therefore, it cannot selectively permit specific discovery packets. Moreover, because the REST Service uses HTTP, flow-based control cannot selectively allow or deny individual API operations.

Tab. 3.2: Example operations in DIAL

Operation	HTTP Method and URI
Netflix Start	POST /apps/Netflix?clientDialVer=2.1
Netflix Stop	DELETE /apps/Netflix?clientDialVer=2.1
Netflix Hide	POST /apps/Netflix//hide?clientDialVer=2.1
Netflix Status	GET /apps/Netflix?clientDialVer=2.1

For SSDP, the response to a multicast discovery packet contains a Search Target (ST) header that identifies the service. Payload-based control can inspect this header in the payload. If the service in the ST header is permitted to be discovered, the packet and its response packets are passed through. For the DIAL REST Service, all operations are defined as an HTTP REST API. Proxy-based control can inspect HTTP requests and enforce policies based on the method and URI. If the operation is permitted, the request is passed through; if the operation is not permitted, the request is discarded.

I confirmed that this control works with dial-reference [92], a simulator for the DIAL protocol. For SSDP, only responses that contain the ST header with urn:dial-multiscreen-org:service:dial:1 are permitted, and all others are dropped. For the DIAL REST service, capabilities are assigned to each of the Netflix-related operations listed in Table 3.2. I confirmed that only authorized devices could discover other devices, and only permitted DIAL operations could be performed.

3.6 Discussion

3.6.1 The Validity of the Authorization Architecture

The proposed authorization architecture uses a capability-based model rather than a policy-based model such as RBAC. The main benefits of the capability-based model are fine-grained access control and improved usability. A policy-based model requires users to define detailed policies for each device and each communication [93]. In particular, which device may communicate with which other device must be specified in advance. Users need to understand the details of communications among devices and define policies such as access control lists. This process can be complicated and may require testing before deployment in real environments. In addition, the

granularity of control depends on users' effort, which can lead to violations of the principle of least privilege.

In contrast, in the capability-based model, a device defines capabilities for each operation it provides. Authorizing a capability allows a device to perform the corresponding operation, including the associated communication. Users do not need to know the details of device interactions; instead, they authorize capabilities when devices request them. Therefore, the capability-based model can provide fine-grained access control with less user effort. However, the number of authorization requests may increase. To address this issue, I designed Automatic Authorization to reduce the number of authorization requests.

I also discuss the proposed authorization architecture and access-control methods from the perspective of the guidelines in Section 3.3.

(1) Allow fine-grained control

I propose access control methods based on flow-based, payload-based, and proxy-based control. As mentioned in Section 3.5, these methods cover not only flow-oriented bulk transfers but also messages carried over TCP. Therefore, the granularity of control depends on how capabilities are defined and how twin devices are implemented. If device vendors define capabilities appropriately, users only need to authorize a small number of capabilities to achieve fine-grained control.

(2) Plan for lending devices

A typical home has multiple users, and devices are loaned or shared among them [76, 77, 78]. In the proposed design, capabilities are issued to twin devices; therefore, user-specific access control can be achieved by assigning a per-user twin device to a shared physical device. In addition, the appropriate authorization scope can vary—for example, some capabilities should be authorized only by the device owner, while others may be authorized by family members or visitors. Therefore, the system should support per-user authorization restrictions.

In a multi-user system, the system needs to know who is using the device. Switching twin devices in the CREBAS WebUI or the device's UI is one option, but it can be burdensome. Mechanisms are required, such as encouraging a user to switch a twin device when a lent device is connected to a user's wireless AP. Developing a multi-user system that considers lending devices is left for future work.

(3) Include reactive policy creation

As shown in Fig. 3.9, users can manually authorize capability requests that were not automatically authorized with one click.

(4) Include logs

Since all capabilities are issued or delegated via CP, the capabilities used for access control are maintained on CP. Logs for authorization can be easily obtained from CP.

The logs of the actual network access control operations are based on a controller in the PEP and twin devices. In the implementation, a controller has logs for flow-based control. The logs contain what twin device was started, what capability was applied, and what flow was added to an OpenFlow switch. The twin devices have logs for payload-based and proxy-based control. The logs contain what packets or messages were received and how a twin device processed them. Since a controller and twin devices are just software, the controller and twin devices can be designed to collect more detailed logs and support analysis.

(5) Reduce or eliminate up-front complexity

By using twin devices and automatic configuration of twin devices, access control can be achieved by simply connecting physical devices to an OpenFlow switch connected to the PEP. Therefore, there is no need to configure settings or policies for each device. Users can control communications only by making some authorization decisions in Manual Authorization. Automatic Authorization is designed as a countermeasure to the increase in authorization requests to users.

A profile system [79] can extend Automatic Authorization. Capabilities that are often authorized and considered unlikely to cause security vulnerabilities can be authorized automatically. With profiles to authorize such capabilities automatically, Automatic Authorization will be more usable.

(6) Support iterative policy specification

Users may want to change authorized capabilities according to changes in their environment and devices. To handle changes, CREBAS needs to support revoking capabilities. This can be supported by simply adding “revoked capabilities” information to the PEP. Users can revoke or re-grant capabilities. Additionally, timeout mechanisms will help users to manage authorized capabilities by encouraging users to reflect their current environments.

As described above, the architecture and access-control methods largely meet the guidelines. However, some points need to be addressed in the future.

3.6.2 The Effectiveness of Automatic Authorization

Automatic Authorization is designed for cooperation among the same vendor's devices. In the authorization architecture, CREBAS needs to authorize and issue capabilities for all communications including device discovery. If a device wants to discover other devices, users are required to authorize each capability to the discovering device. As an example, suppose there are n devices, and each device wants to discover the others. Without Automatic Authorization, users need to authorize capabilities $n(n - 1)$ times. If there are 10 devices on a home network and they want to discover each other, users must authorize them 90 times. This is a weakness of the capability model. To reduce the number of authorizations in the capability model, I designed Automatic Authorization. Automatic Authorization reduces the number of authorizations from $n(n - 1)$ to *zero*. Users are not required to authorize many times, and devices can discover each other.

3.6.3 Comparison with Prior Work

CREBAS provides fine-grained network access control to control communications in units of operations. However, there is a dilemma that the more fine-grained the control provided, the more effort is required for users to configure the control. If the required operations are complex or burdensome, users will misconfigure the system or avoid using it. Usability is very important, and the access control system needs to account for that.

As discussed in Section 3.2, prior work [26, 80, 81, 70, 71] does not discuss how to configure such systems or only provides a profile-based system. Such approaches are unlikely to be adopted in home networks, and more work is needed to apply them in real deployments. Some appliances (e.g., FortiGate, YAMAHA, SquidGuard, etc.) provide rule-based or profile-based access control. These require users to configure rules described by regular expressions or use “black-box” profiles. Such setup processes are too burdensome for ordinary people.

CREBAS is designed to enable access control with one click based on knowledge for usable access control systems [74]. “One click” means that CREBAS does not require users to configure complex policies and rules. CREBAS generally requires

two steps to configure access control. Users just need to read the brief description of a communication and click the GRANT button on the WebUI to allow it, like permission management in smartphones.

This is achieved by an authorization architecture based on the capability model and a network access control method including auto-configuration mechanisms. The authorization architecture handles capabilities that have descriptions and specifications of communication and a policy for automatic authorization. Furthermore, as described in Section 3.4, components including capabilities and twin devices are automatically configured using DHCP. Users do not need to care about network status and protocols like “What IP address is assigned to the device?”. As demonstrated in Section 3.5.4, the one-click configuration with CREBAS works in a real environment, and users are not required to configure rules or policies.

However, users may make inappropriate authorizations by mistake. To reduce such mistakes, the user interface should be designed appropriately. The design of the user interface is beyond the scope of this chapter. However, the capability-based authorization architecture will make it easier because users do not need to define rules or policies. Additionally, the detection of mistakes by machine learning [94] will be easier. This is because all authorizations and controls are represented as capabilities like permissions in smartphones, not complex rules and policies. Detecting suspicious authorizations (capabilities) and recommending that users revoke them will help reduce misconfiguration.

3.6.4 The Limitations of CREBAS

CREBAS has two limitations arising from twin devices.

One limitation is that if a physical device does not have a twin device, CREBAS does not work. Users must develop a twin device themselves if a device does not have one. However, some devices may lack vendor support or public protocol specifications, and this prevents users from developing twin devices. This is also true for prior work discussed in Section 3.6.3. They require the protocol to be public to define their rules or profiles. Device vendors should develop twin devices or make their protocols public to enable CREBAS and prior work to protect other devices.

The other limitation is the granularity of access control. The granularity of access control relies on twin devices. Twin devices may provide insufficient granularity of control and capabilities. Such a twin device is undesirable, and device vendors should develop twin devices earnestly to protect their devices.

3.7 Conclusion

In this chapter, I proposed CREBAS, a system that enables access control in a home network with one click. To realize this system, I proposed a capability-based authorization architecture and network access control methods. I designed an authorization architecture according to the guidelines of access control in a home. Automatic Authorization uses capability delegation to make authorization decisions and reduce user burden. To provide sufficiently granular access control with authorized capabilities, I designed network access control with flow-based, payload-based, and proxy-based control. These methods enable more fine-grained access control than prior work. Twin devices process capabilities and control packets with minimal modification of existing devices. Finally, the evaluation with a prototype system has shown that the performance overhead is acceptable, and the methods work with a real protocol.

Acceleration of TCP/IP Communications in Rootless Containers

This chapter is based on a paper “bypass4netns: Accelerating TCP/IP Communications in Rootless Containers” published at AINTEC 2024 [95].

4.1 Introduction

Containers is an isolation technology provided by container runtimes that leverage Linux kernel features. The software stack that provides containers consists of low-level runtimes (e.g., runc [96]) and high-level runtimes (e.g., containerd [97]). Thanks to isolation, a process in a container cannot perform malicious operations on the host. However, vulnerabilities in low-level [54, 55] or high-level [56, 57] runtimes can break isolation [98]. Because these runtimes run with root privileges, such vulnerabilities may allow malicious code to run as root on the host.

Rootless containers [58] improves container security by running container runtimes without root privileges. Even if a runtime has vulnerabilities, malicious code is not executed with root privileges. Rootless containers uses mechanisms such as user namespaces to run without root privileges. However, it cannot use network components designed for rootful runtimes. Therefore, RootlessKit [99] and slirp4netns [100] provide communication with external endpoints for rootless containers. External endpoints are those outside the intermediate network namespace (NetNS) constituting the rootless container network. While these components enable a fully rootless network, they incur significant overhead, resulting in low communication performance. This performance degradation is one factor preventing wider adoption of rootless containers.

In this chapter, I propose **bypass4netns** (b4ns), a method to bypass the communication bottleneck in rootless containers. In principle, b4ns provides a rootless network with performance equal to or better than that of a rootful container. I introduce **socket switching** to bypass communication bottlenecks. It switches sockets created

in a container to sockets created on the host. Socket switching is also referred to as "*socket grafting*" [101] and "*socket replacement*" [102]. Existing approaches rely on LD_PRELOAD, which cannot handle statically linked applications, or on eBPF, which requires root privileges. b4ns uses Seccomp User-space Notification (Seccomp Notify) [103], supported since Linux kernel 5.0, and SECCOMP_IOCTL_NOTIF_ADDFD, supported since kernel 5.9, to switch sockets without modifications to applications or the kernel. This enables **fully rootless** socket switching, including for **statically linked applications**. However, switching sockets alone is insufficient for application compatibility. Misconfigured sockets can cause issues. For example, the apt-get command hangs when a socket is switched but configured improperly. I investigated socket behavior in applications and implemented b4ns accordingly.

Containers are often used in orchestration systems such as Kubernetes [60]. Usernetes [104] provides Kubernetes with rootless containers, but communication performance is very low. It uses VXLAN to communicate among multiple nodes. Rootless network components process the VXLAN packets, which causes significant performance degradation. To address this problem, I propose **rootless** multi-node communication with b4ns. It does not use any packet encapsulation approaches, including VXLAN.

I confirmed that b4ns can provide up to $30\times$ faster throughput than rootless containers without it. In addition, it showed significant performance improvements on several applications. b4ns removes barriers to using rootless containers. It provides more secure container technology without requiring application modifications. In summary, the main contributions are:

1. I propose bypass4netns to accelerate rootless container networking with socket switching without application modifications.
2. I propose a high-performance, **fully rootless** multi-node communication method.
3. I implemented b4ns and confirmed that the proposed methods can accelerate application communications without modifying them. b4ns provides more than $30\times$ faster communication.

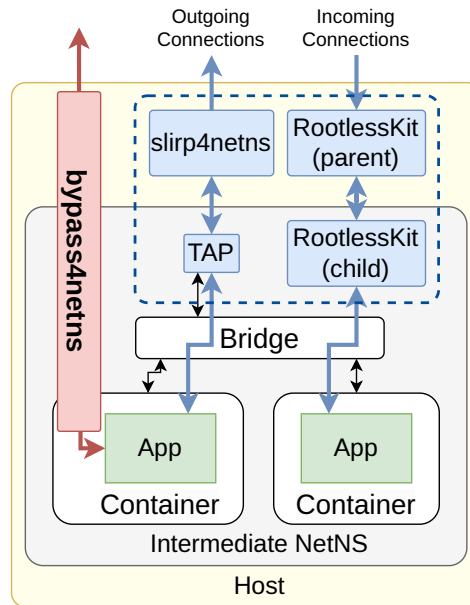


Fig. 4.1: Overview of the rootless container network
Components in the blue box provide a rootless network

4.2 Background

4.2.1 Rootless Containers

Rootless containers improves container security by running the container runtime without root privileges. If a rootless runtime has vulnerabilities that allow code execution outside containers, that code executes without root privileges. The impact is less than with privileged container runtimes.

Rootless containers uses user namespaces (UserNS) to create namespaces without root privileges. UserNS maps a non-root user such as `uid=1000` on the host to a fake root user (`uid=0`) in a UserNS. In the host environment, the user remains a non-root user. UserNS enables container runtimes to provide containers without root privileges. However, operations related to network interfaces cannot be achieved with UserNS alone. The container runtime must create a veth pair both in a container and on the host. This veth pair is the bridging point for container communication and requires root privileges to create on the host. To enable container networking without root privileges, rootless containers uses RootlessKit and slirp4netns.

Fig. 4.1 shows the current rootless container network. RootlessKit's port driver [99] handles connections from external endpoints. The port driver on the host accepts

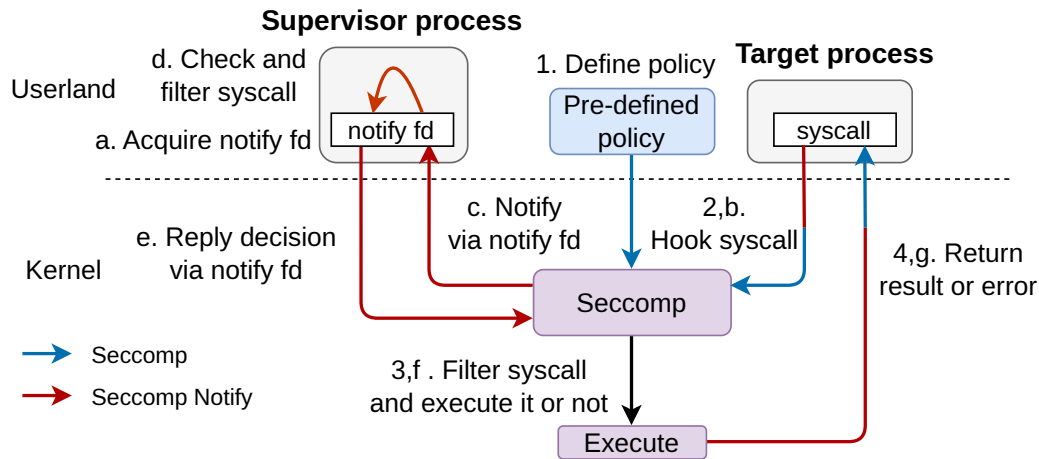


Fig. 4.2: The syscall filtering flow in Seccomp Notify

connections, and the port driver in the intermediate NetNS connects to containers. The port drivers relay connections by copying messages between two sockets. `slirp4netns` [100] handles connections from containers to external endpoints. It receives packets from containers via the TAP device and extracts payloads. Then, it creates a `SOCK_STREAM` socket and sends the payloads to a peer. When `slirp4netns` receives payloads from the peer, it constructs TCP packets and sends them to containers via the TAP device. These operations cause severe performance degradation.

4.2.2 Seccomp Notify

Seccomp (secure computing mode) [105] is a Linux kernel’s functionality for controlling syscall executions in a target process. Seccomp is combined with a container runtime. It hooks and filters syscalls when a target process in a container invokes them. Seccomp filters syscalls with a statically defined policy and cannot filter them dynamically. Seccomp Notify refers to “*Seccomp user-space notification*” or “*seccomp_unotify(2)*” [103]. It provides an interface to filter syscalls dynamically from a user-space process.

Fig. 4.2 shows the flow of syscall filtering in Seccomp and Seccomp Notify. Seccomp requires a predefined policy to filter syscalls. The Seccomp module in the kernel filters syscalls invoked by the target process based on the policy before execution. If the syscall is allowed to execute, the kernel executes it and the result is returned to the process. If not allowed, the syscall is not executed and Seccomp returns an error. Seccomp Notify enables filtering syscalls dynamically. A process that checks and filters syscalls is called the supervisor process. Seccomp Notify provides a file

Tab. 4.1: Comparison with socket switching approaches

	Fully Rootless	Performance	No kernel modification	No application modification	Publicly available
RootlessKit port driver [99] slirp4netns [100]	✓	Low	✓	✓	✓
lxc-user-nic [106]		Mid	✓	✓	✓
AF_GRAFT [101]		High		LD_PRELOAD	✓
Slim [107]		High		LD_PRELOAD	✓
O2H [102]		High	✓	✓	
bypass4netns	✓	High	✓	✓	✓

descriptor called notify fd to the supervisor process to interact with the Seccomp module. When the target process calls a syscall, the module notifies the call via the notify fd to the supervisor process. The notification contains various information including the type of syscall, its arguments, and PID. The supervisor process handles the notification and decides whether to execute the syscall with the information. The decision is returned via the notify fd, and the module filters the syscall based on the reply. Since Linux kernel 5.9, SECCOMP_IOCTL_NOTIF_ADDFD has been available. This allows the target process to use a file descriptor that references the supervisor's file description (for example, a socket).

4.2.3 Socket Switching

Table 4.1 compares socket switching methods. lxc-user-nic [106] is known as a network feature for unprivileged LXC [46]. It provides veth pair interfaces without root privileges. However, lxc-user-nic effectively runs as the root user. Therefore, if lxc-user-nic has a vulnerability such as [108, 109], malicious operations can be executed with root privileges. Some approaches bypass container networks to accelerate TCP/IP communications. AF_GRAFT [101] and Slim [107] replace socket operations in applications with their sockets with LD_PRELOAD. libc is a popular shared library that provides wrapper functions for system calls, including socket-related ones. LD_PRELOAD approaches hook these libc socket-related function calls and modify their behavior. However, these approaches cannot replace sockets in statically linked applications because those applications do not call libc functions. Furthermore, they require loading a kernel module. If such a module has vulnerabilities, it can cause serious damage to the entire host. Relying on a custom kernel module is inconsistent with the philosophy of rootless containers. O2H [102] switches sockets in containers to the host's sockets using eBPF. It does not require modifications to applications, including statically linked ones. However, attaching eBPF programs to the host

communication is not handled by b4ns because it uses veth pairs and bridges in the intermediate NetNS.

b4ns uses sockets created on the host, which is similar to using the host NetNS instead of the container NetNS. However, using the host NetNS directly can leak abstract UNIX domain sockets from the host into containers, resulting in vulnerabilities [111]. To ensure security, b4ns does not use the host NetNS directly; it switches only sockets connecting to or from external endpoints. This design preserves the container's NetNS. It also allows IP addresses to be assigned to containers for inter-container communication.

4.3.2 Analyzing Socket Behavior with eBPF

To choose sockets to switch, b4ns requires the connection endpoint or the port to bind. Additionally, some applications configure options on sockets, such as non-blocking mode. Incorrect configurations can lead to unexpected behavior. To ensure compatibility, b4ns must correctly capture such options and configure created sockets accordingly. This raises the following questions.

Q1: What kinds of syscalls are executed for sockets? Hooking syscalls with Secomp Notify introduces overhead. Hooking every syscall would degrade application performance. To minimize performance degradation, b4ns needs to handle only the syscalls related to switching sockets. Syscalls perform operations on sockets via file descriptors. Some syscalls (e.g., `read(2)`) are not socket-specific, and the set of socket-related syscalls is not explicit.

Q2: Which syscalls configure sockets? Applications sometimes configure the non-blocking option or buffer size on their sockets. To avoid affecting application behavior, b4ns records these options and configures created sockets accordingly.

I explored these points by collecting syscalls from several applications (`iperf3`, `wget`, `nginx`, `apache`). To collect the syscalls, I used `tracee` [112], an eBPF-based syscall tracer. I implemented a script to retrieve operations on each socket's file descriptor from the collected traces. The syscalls were investigated and classified into seven classes based on their behavior. Table 4.2 shows the classes. Previous works [101, 107] replace syscalls, including (6) Communication syscalls, with their own APIs. To avoid performance degradation, b4ns does not handle (6) Communication syscalls. b4ns uses Linux sockets as the switched sockets and therefore does not need to provide its own socket APIs or API replacements. b4ns switches the target of a file descriptor from a socket in a container to a socket on the host. It does not change

Tab. 4.2: Traced syscalls related to sockets

Bold syscalls are hooked by bypass4netns syscalls	
(1) Creation	socket
(2) Configuration	fcntl, setsockopt, ioctl
(3) Connection	connect, bind
(4) Status	getsockopt, getsockname, getpeername
(5) Derivation	accept, accept4, clone
(6) Communication	poll, recvfrom, sendfile write, select, read, listen lseek, readv, writev, epoll_ctl, epoll_wait
(7) Close	close, shutdown

the file descriptor or socket APIs. Processes execute syscalls like `write(2)` with the same file descriptor, and a socket on the host handles them instead of one in the container.

4.3.3 Socket Switching With Seccomp Notify

b4ns performs socket switching with `SECCOMP_IOCTL_NOTIF_ADDFD`. Fig. 4.3 shows the overview. The accelerated container is configured with a Seccomp Notify profile provided by b4ns. The profile is set to notify syscalls related to socket switching, as shown in Table 4.2. b4ns does not hook other syscalls; they execute normally. When the related syscalls are called, b4ns receives a notification message containing information about the syscall from the kernel via the notify fd. After validation, b4ns processes the syscall based on its class. When b4ns performs socket switching, it sets the `SECCOMP_IOCTL_NOTIF_ADDFD` flag in the reply message and includes the new socket's file descriptor to replace the original. With this flag, the Seccomp module changes the target of the file descriptor from a socket in the container to the newly created socket.

Fig. 4.4 shows the entire flow of socket switching. First, a process in a container acquires a socket via (1) Creation or (5) Derivation syscalls. b4ns ignores these syscalls. Then, a process configures the socket with (2) Configuration syscalls. b4ns needs to know what configurations are set on the socket, so it hooks (2) Configuration syscalls and records them as socket options with their arguments. When (3) Connection syscalls are called, b4ns reads their destination address or binding port. If the destination is an external endpoint or the port is a published one, b4ns performs socket switching. During socket switching, b4ns creates a new socket and configures

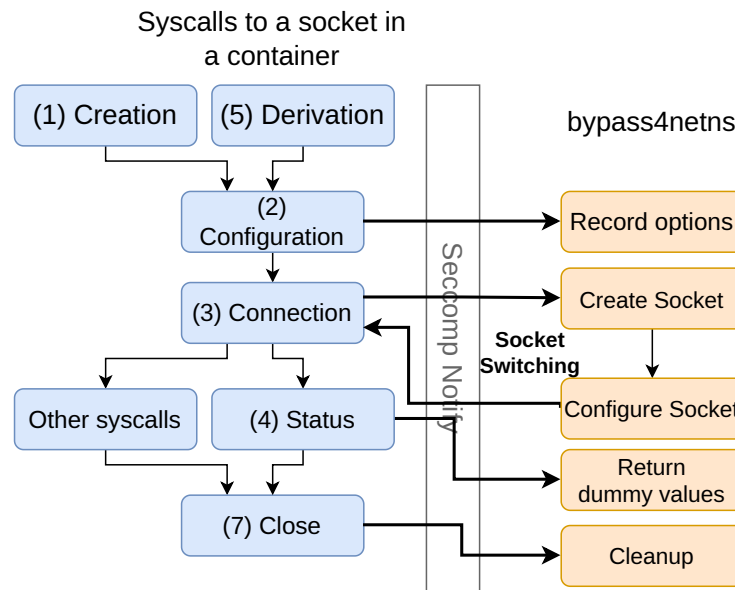


Fig. 4.4: Flow of socket switching

it with the recorded options to match the state of the container's socket. Then, using `SECCOMP_IOCTL_NOTIF_ADDFD`, b4ns switches the container's socket to the created socket. After socket switching, a process can call (4) Status syscalls. With b4ns, the peer's address is not the container's but the host's address when connecting to another container's published port. Connecting to another container but showing the host's address is confusing for some applications. To solve this issue, b4ns returns the container's address when the syscall is called. Finally, a process calls (7) Close syscalls, and b4ns cleans up the socket. For syscalls that are not related to socket switching, b4ns sends a response message to allow syscall execution.

4.3.4 Publishing Ports with Socket Switching

Users publish container ports to make them accessible to others. This operation is called *publishing ports*. For example, an application in a container listens on `80/tcp`. Other hosts cannot connect to the port because the application is in the container's NetNS and is isolated from the host's NetNS. If a user publishes the port as `8080/tcp`, others can access the application via the host's `8080/tcp` port.

b4ns accelerates published ports by binding the socket with the published port on the host, but this causes the issue shown in Fig. 4.5. In conventional rootless containers, ports are published with RootlessKit. Applications bind sockets in their container's NetNS, and RootlessKit relays connections from outside the intermediate NetNS.

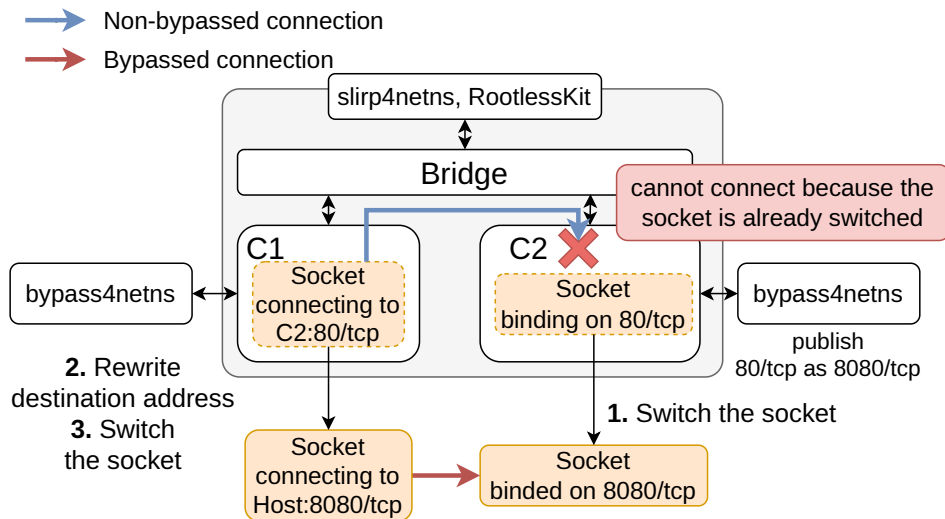


Fig. 4.5: Publishing ports with `bypass4netns`

When an application in a container listens on `80/tcp` and the port is published as `8080/tcp`, other containers in the same intermediate NetNS can connect to the application at `C2:80/tcp`. Others outside the intermediate NetNS connect to `Host:8080/tcp`, and the connection is relayed by `RootlessKit`. With `b4ns`, the socket is switched to a host socket, and other containers cannot directly connect to `C2:80/tcp`. To solve this issue, `b4ns` rewrites the destination address in `connect` (2). When a container connects to a published port (`C2:80/tcp`), `b4ns` rewrites the address to the host's published address (`Host:8080/tcp`). This enables containers to use `C2:80/tcp` transparently while external hosts use `Host:8080/tcp`.

4.3.5 Fast Multi-node Communication

Containers are often used in multi-node environments such as Kubernetes. In the Kubernetes network model [113], it requires four communications: Pod-to-Pod, External-to-Service, Pod-to-Service, and Pod-to-External. These communications require that every Pod and Service in a cluster can communicate with every other even when deployed to different nodes. To enable multi-node communication, users employ overlay networks with encapsulation approaches [114] or L3 routing-based approaches [115]. The L3 routing approach is not feasible because modifying the host's L3 routing table requires root privileges. Usernetes [104], which runs Kubernetes in rootless containers, provides multi-node communication with VXLAN [116]. In Usernetes, the VXLAN interface is created in rootless containers to avoid requiring root privileges. VXLAN packets are handled by `slirp4netns` and `RootlessKit`. Due

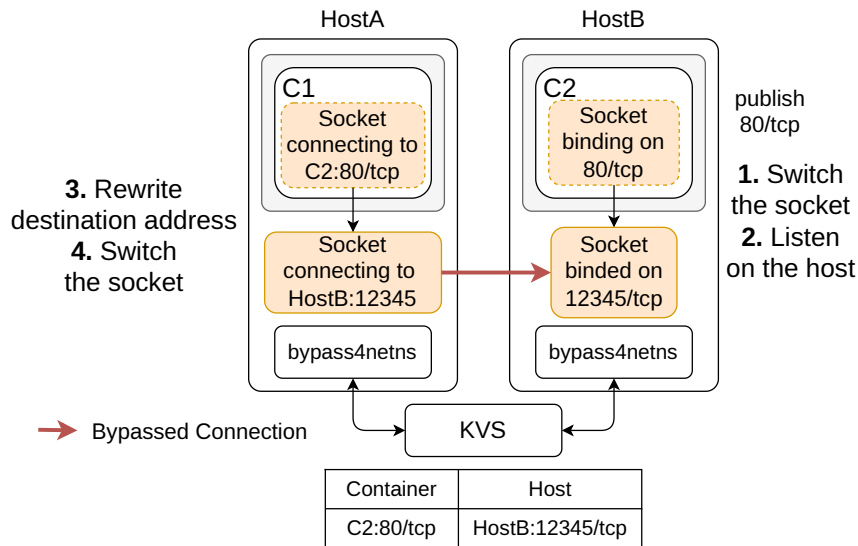


Fig. 4.6: Multi-node communication approach

to the overhead of these components, Pod-to-Pod communication performance becomes lower than port forwarding or rootful VXLAN communication. b4ns cannot accelerate VXLAN communication because it handles only user-process sockets, whereas VXLAN interfaces allocate sockets in the kernel. External-to-Service and Pod-to-Service are not affected by b4ns without multi-node support because they use NodePort [117], which does not rely on VXLAN [118]. Pod-to-External is accelerated with b4ns using the same approach as in normal rootless containers. Pod-to-Pod communications require another approach to accelerate.

To provide faster multi-node communication, including Pod-to-Pod, for rootless containers, I propose a simple multi-node communication method without VXLAN. Fig. 4.6 describes the approach. This approach is similar to Slim in that it does not use an encapsulation protocol. Each node runs rootless containers and publishes some ports. Each b4ns instance shares the mapping between a published container's address and the corresponding host address via a KVS. The implementation uses etcd as a KVS. When an application in container C2 calls `bind(2)` on `80/tcp`, b4ns assigns a port (e.g., `12345/tcp`) and binds the socket on the host. When C1 on HostA tries to connect to the published port (`C2:80/tcp`) of C2 on HostB, b4ns rewrites its destination address to the corresponding HostB address (`HostB:12345/tcp`).

4.4 Implementation and Insights

4.4.1 Implemented Components

I implemented components with the design in Section 4.3. The implementation is available at <https://github.com/naoki9911/bypass4netns>.

b4ns It receives notifications from containers and switches sockets as needed. One b4ns is assigned to one container. Each b4ns and container is isolated, except for multi-node communication.

b4nsd It manages bypass4netns lifecycle and configuration. When a container with b4ns starts, b4nsd receives container and network information from nerdctl, such as the ports to be published.

nerdctl [119] The implementation includes modifications to nerdctl to handle b4ns and b4nsd. nerdctl is a Docker-compatible CLI for containerd [97]. It cooperates with containerd and provides container functionality. When a container with b4ns starts or stops, nerdctl notifies b4nsd of the container's information to start or stop b4ns.

4.4.2 Insights in Implementation

Seccomp Notify handles only syscalls before execution

Seccomp filters syscalls **before their execution**; this is also true for Seccomp Notify. It checks syscalls and decides dynamically whether to execute them before execution. b4ns manages options and states based on file descriptors (FDs). However, b4ns cannot know which FD was assigned by socket(2) because b4ns cannot observe the return value due to the Seccomp limitation mentioned above. To overcome this limitation, b4ns checks whether an FD refers to a socket and registers the FD when relevant syscalls are hooked. Furthermore, FDs sometimes already refer to connected sockets. Such FDs can be seen after (5) Derivation syscalls in Fig. 4.4. b4ns checks whether the socket is connected with getpeername(2). If the socket is connected, it registers the corresponding FD as a non-switchable socket. Once the FD is registered, b4ns performs socket switching based on the registered state and recorded options.

Insufficient privileges

b4ns uses `/proc/<PID>/mem` to read and write destination addresses. Some container applications create users in a container and run as those users. b4ns runs without elevated privileges and does not have permission to read or write another user's process memory. To read and write another process's memory, b4ns launches an agent to open `/proc/<PID>/mem` as the process's user using `nsenter(1)`. The agent passes the opened FD to b4ns via a UNIX domain socket, and b4ns reads and writes the process's memory.

4.5 Evaluation

This section evaluates both network-level and application-level performance.

4.5.1 Environment

Evaluations were performed on QEMU/KVM virtual machines (VMs). The host has an AMD EPYC 7452 (32 cores, 2.35 GHz base clock) with Ubuntu 22.04. Each VM was assigned 8 vCPUs and 16 GB of memory with Ubuntu 22.04. Benchmarks were run between a benchmarking container in one VM and a server container in another VM on the same host.

Benchmarks covered six network configurations.

rootful-pfd Containers run with containerd as root. Ports are published with port forwarding (i.e., the `-p` option).

rootless-pfd Containers are rootless. Ports are published with port forwarding, and communications are processed by `slirp4netns` and `RootlessKit`.

b4ns-pfd Containers are rootless, and b4ns performs socket switching on published sockets.

rootful-vxlan Containers run with containerd as root. Hosts have a VXLAN interface, and containers on both hosts communicate via the VXLAN tunnel.

rootless-vxlan Containers are rootless. They have a VXLAN interface and `4789/udp` is published, which means `slirp4netns` and `RootlessKit` process all VXLAN packets.

b4ns-multinode Containers are rootless, and multi-node-enabled b4ns performs socket switching.

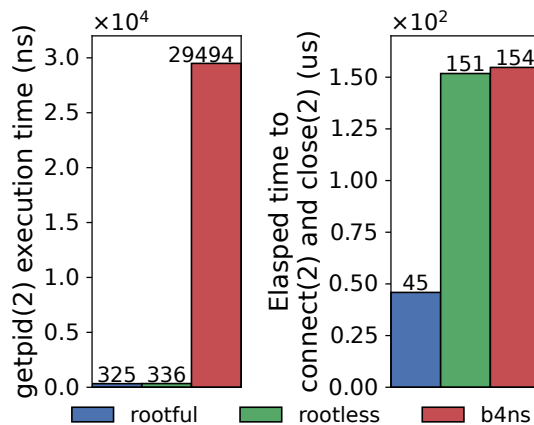


Fig. 4.7: Switch overhead

4.5.2 Switching Overhead

The evaluation measured the overhead caused by Seccomp Notify and by socket switching. b4ns was modified to hook `getpid(2)` and its execution time was measured to evaluate the overhead caused by Seccomp Notify. Socket switching does not require hooking `getpid(2)`, but this syscall is useful to measure Seccomp Notify overhead because it is simple and always succeeds. b4ns allows `getpid(2)` to execute and does nothing else. To measure the switching overhead, a program that calls `connect(2)` to a server and then `close(2)` was used. This is the smallest case that invokes switching. I measured the elapsed time from immediately before `connect(2)` to immediately after `close(2)`. Fig. 4.7 shows the results with the average time for 100,000 `getpid(2)` executions and 10,000 socket switches. With b4ns, `getpid(2)` took approximately 100 times longer than without it. b4ns was designed not to hook (6) Communication syscalls that directly affect communication performance. This design allows performance improvements despite Seccomp Notify's large overhead. Socket switching took about four times longer than in rootful containers and almost the same as in rootless containers. This may cause performance degradation when processing many short-lived connections. Some applications use a connection pool to reuse connections, which reduces socket switching and mitigates the degradation.

4.5.3 Application Compatibility

Compatibility was tested with the applications described in Table 4.3. To demonstrate that b4ns works without modifying applications, official container images were used

Tab. 4.3: Tested Applications

Application (version)	Language
apt (2.4.12) [120]	C++
iperf3 (3.16) [121]	C
nginx (1.18) [122]	C
redis (7.2.3) [123]	C
memcached (1.6.22) [124]	C
rabbitmq (3.12.10) [125]	Erlang
etcd (3.3.25) [126]	Go
postgresql (16.1) [127]	C
mysql (8.2.0) [128]	C++
HTTP Client (Go 1.21.3) [129]	Go (static linked)
redis-benchmark (7.2.3) [130]	C
memtier_benchmark (2.0.0) [131]	C++
rabbitmq-perf-tester (2.20.0) [132]	Java
etcd-benchmark (3.3.25) [133]	Go (static linked)
pgbench (16.1) [134]	C
sysbench (1.0.20) [135]	C

when available. If not available, images were built from packages or source code without modifications. b4ns was compatible with these applications, including statically linked applications, except for HTTP Client (Go 1.21.3) [129]. When running more than two HTTP clients concurrently using goroutines, Seccomp Notify silently dropped some syscalls. This behavior is unexpected and warrants further investigation before deploying b4ns in production environments. Running a single HTTP client did not cause the issue, and performance was evaluated with one HTTP client.

4.5.4 Application Performance

Application performance was measured with iperf3, nginx, Redis, and MySQL.

Throughput measurement with iperf3

Fig. 4.8 shows the throughput measured by running iperf3 [121] for 120 seconds; values are averages over 10 runs. The other VM ran the iperf3 server in a container. With one parallel stream, the results were rootful-pfd (17.9 Gbps), rootless-pfd (570 Mbps), and b4ns-pfd (19.7 Gbps). A rootless container setup with b4ns provided more than 30× higher throughput than one without it and was slightly faster than the rootful setup. b4ns bypasses iptables and other network modules, which appears to reduce overhead and provide better performance. For

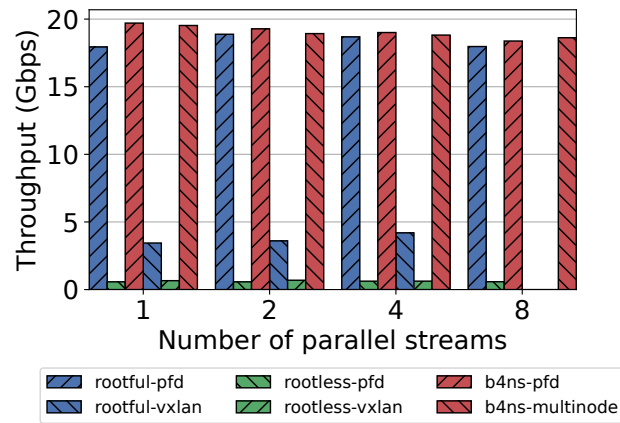


Fig. 4.8: Benchmark with iperf3

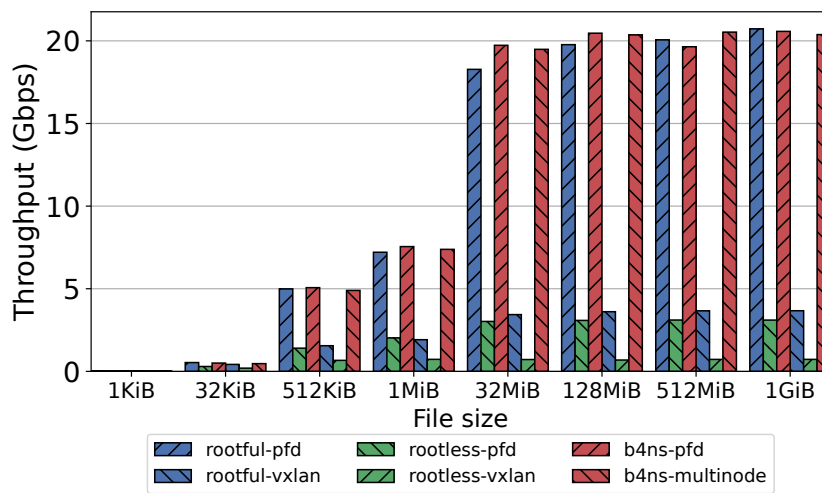


Fig. 4.9: Benchmark with nginx

multi-node communications, b4ns's multi-node mode achieved higher performance than VXLAN.

Static file distribution with nginx

Fig. 4.9 shows static file distribution performance with nginx [122]. The time to download files 1,000 times was measured and throughput was calculated. Files were generated by /dev/urandom. A benchmark client was implemented using Go's net/http as an HTTP client. The client was built as a statically linked binary. This benchmark demonstrated that b4ns can handle statically linked binaries that LD_PRELOAD cannot. Performance for rootful-pfd and b4ns-pfd was almost the same. rootless-pfd, rootless-vxlan, and rootful-vxlan showed lower performance, likely due to the overhead of rootless components and VXLAN.

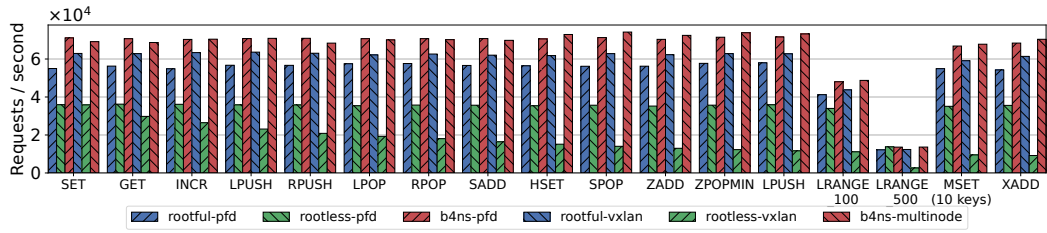


Fig. 4.10: Benchmark with Redis

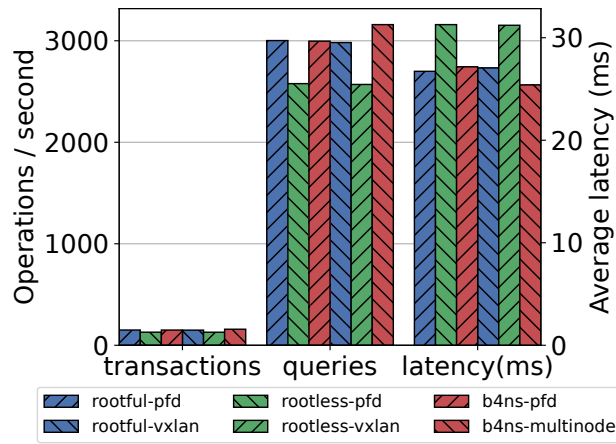


Fig. 4.11: Benchmark with MySQL

Redis benchmark with redis-benchmark

Redis [123] performance was measured with redis-benchmark [130]. Redis is a key-value store (KVS). redis-benchmark was run with default parameters (data size 3 bytes, 50 parallel connections, no multithreading). Fig. 4.10 shows that b4ns achieved almost twice the performance of rootless containers without b4ns. Rootful containers had lower performance than rootless containers with b4ns. In latency measurements, rootful containers had slightly higher latencies. In the iperf3 and nginx benchmarks, communications were bulk transfers and thus were not heavily affected by latency. Redis uses a request-response protocol, and latency appears to affect its performance. Network components in rootful containers, such as iptables and Linux bridge, increase latency. b4ns likely reduces latency by bypassing these components, resulting in the best performance.

MySQL benchmark with sysbench

Finally, MySQL [128] performance was measured with sysbench [135]. MySQL is a widely used relational database (RDB), and sysbench is a multithreaded benchmark tool for database systems. sysbench was run with the following parameters,

and values were averaged over 10 runs. The number of threads was 4 and the running time was 120 seconds. The `oltp_read_write` test, an online transaction processing (OLTP) test with read and write, was used. Fig. 4.11 shows the result. Rootless containers (`rootless-pfd`, `rootless-vxlan`) had relatively lower performance and higher latency than the others. However, the performance improvements were not as large as in `iperf3`, `nginx`, and `Redis`. Overall latency was higher compared to the `Redis` benchmark. This implies that `MySQL` requires more time to process transactions or queries on the server. For applications with longer round-trip times, the impact of network improvements is relatively small, and the benchmark reflects that.

4.6 Discussion

4.6.1 Security Considerations

Since `b4ns` switches sockets in containers to sockets created on the host, network isolation is partially weakened. One concern is that `iptables`-based access control is bypassed. Without `b4ns`, container communications are controlled by `iptables` in the intermediate `NetNS`, but `iptables` cannot control communications with `b4ns`. This is because communications with `b4ns` are treated the same as other communications on the host. Instead of relying on `iptables` for access control, `b4ns` checks reachability in the intermediate `NetNS` to reflect `iptables` policies in socket switching, using an approach similar to `XMasq` [136]. Thus, users can still control network access indirectly by configuring `iptables` rules. In addition, `b4ns` can control every `connect(2)` call, which will help provide `syscall`- or destination-based control.

`Seccomp Notify` poses a risk of allowing a container to connect to host services by exploiting a time-of-check to time-of-use (TOCTOU) attack [137]. TOCTOU is a race condition that modifies data after the check but before use. For `b4ns`, the address for `connect(2)` is validated to prevent connections to the host's loopback address. However, if the process rewrites the address after validation, a race could allow the connection. To mitigate this issue, `b4ns` should copy the address and execute `connect(2)` itself.

4.6.2 Kubernetes Support

b4ns includes a method to accelerate multi-node communication in Section 4.3.5. Multi-node communication in b4ns targets faster Pod-to-Pod communication. The evaluation shows that the method provides a faster multi-node network than current VXLAN-based approaches. However, with this approach, all Pods allocate sockets on the host. These sockets are exposed on the host, which may raise security concerns. The host may also experience port exhaustion due to many allocations. The Usernetes project plans to use a sidecar proxy to accelerate Pod-to-Pod communication without exposing container ports. Future work will explore improved approaches for faster rootless networking in Usernetes, potentially combining both strategies.

4.6.3 As a Versatile Socket Switching Method

Some fast TCP/IP communication approaches [138, 139] provide a BSD sockets API. To use them, developers need to rewrite socket-related function calls in applications. However, not all applications can be rewritten. LD_PRELOAD enables applications to use modules without rewriting, but statically linked applications cannot use it. b4ns provides functionality to switch application sockets, including those of statically linked applications. b4ns can also switch sockets to other custom kernel module sockets if the module provides file-descriptor-based sockets. In this chapter, its functionality was demonstrated with rootless containers and Linux's TCP/IP stack as the target sockets, but it can be used with other TCP/IP stacks. b4ns makes it easier to use newly developed socket modules with existing applications.

4.7 Conclusion

Rootless containers is a more secure container approach, but it suffers from communication performance issues that prevent full utilization. This chapter presented bypass4netns, a method to improve TCP/IP communication in rootless containers. bypass4netns provides rootless container networking without communication performance degradation. It switches sockets in a container to sockets created on the host and bypasses slow network components. It also provides fully rootless multi-node communication instead of VXLAN. I confirmed that bypass4netns provides up to $30\times$ faster communication than conventional rootless containers without application modifications. Future work includes applying b4ns to more applications and improving Pod-to-Pod communication in Kubernetes integration.

Efficient Container Image Updating with Delta Encodings

This chapter is based on a paper “D4C: Fast and Efficient Container Image Updating for Edge with Delta Encodings” published in IEICE TRANSACTIONS on Communications, Vol.E108-B, No.7, 2025, ©2025 IEICE [140]. **Copyright Notice:** The material in this chapter was presented in part at the IEICE TRANSACTIONS on Communications, and figures of this chapter are reused from [140] under the permission of the IEICE.

5.1 Introduction

Containers provides an execution environment isolated from the host. An independent root file system allows a container to run applications without modifying the host’s root file system. In addition, users can distribute an environment with the necessary libraries and applications preinstalled. This makes it easy to run applications on end hosts. The root file system, which contains applications and libraries, is called a container image. Images are distributed through registries that host them.

Pull refers to downloading the container image and extracting its root file system. If no image exists locally or the image has been updated, the host must pull the image. Container images often range from hundreds of megabytes to several gigabytes, even when optimized. Pulling new images repeatedly consumes substantial network bandwidth. Pull time increases with image size. These issues are challenges in container use. Serverless computing [141] requires containers to start quickly to avoid degrading service quality. In IoT environments or edge computing, where computing and network resources are severely constrained, pulling an image can take several minutes [142]. Distributing images to hundreds or thousands of devices increases network traffic costs on cloud-based distribution servers. Therefore, updating and distributing lighter container images is crucial. Reducing update time

and network resource consumption is important to provide better service quality in edge environments.

To improve pulling, lazy-pulling [143, 144, 145, 146, 147, 64] has been proposed. Lazy-pulling starts the container without pulling the entire container image. In these methods, only the required files to start the container are preferentially downloaded, and containers are launched. However, because these methods eventually pull the entire container image, the transferred data size is not reduced. Another line of work reduces the amount of data transferred during container image updates. Starlight [63] utilizes local images and sends only updated files. This method transfers whole updated files. Thus, even if only a tiny portion of the file is updated, the entire file must be transferred in this approach. In environments with limited network bandwidth, such as WANs [148] and cellular networks [149], it is necessary to minimize transferred data, and there remains room for improvement.

In this chapter, I propose **D4C** (Delta updating For Container images) to reduce the transferred data, the time to update images, and the storage usage required to store them by using delta encoding. Delta encoding is a technique for compressing data for updates by extracting a delta from files or contents. To apply delta encoding to container image updates, it is necessary to reduce the time to generate and apply deltas for container images. I propose a **merge-based delta generation strategy** and **Di3FS** to leverage delta encoding in container image updating. This strategy speeds up the generation of client-tailored deltas by merging already generated deltas. Di3FS applies deltas when reading updated files so that clients can use the updated container image without the time-consuming decompressing and application of all deltas.

The evaluation confirmed that D4C reduced the size of update data by up to about 95% and provided approximately $10\times$ faster updates compared to the file-by-file approach. D4C's merge-based delta generation approach produced required deltas over $60\times$ faster than generating deltas on demand. Even when the image size exceeded 1 GB, the strategy generated deltas in an acceptable processing time. Finally, performance benchmarks with PostgreSQL and PyTorch showed that D4C did not influence their performance. This means that D4C provides faster container image updates without significant application performance degradation.

5.2 Background

5.2.1 Leveraging Containers at the Edge

While containers are being explored for use in IoT and edge computing [142], these environments have constrained network resources compared to cloud computing. I consider mobile networks such as 5G and access networks with speeds below 1 Gbps [149] as edge networks. Container images can sometimes be several gigabytes, necessitating significant time for distribution and deployment. This situation elevates update costs and hinders developers from addressing bugs or adding new functionalities. Additionally, when images are distributed to hundreds or thousands of devices, the network traffic cost on the cloud-based distribution server will increase. Therefore, lighter-weight container image updates are a crucial issue in IoT and edge computing, where containers are deployed on numerous devices and computers.

5.2.2 Container Image Update Issues

The container image [150] features a layered structure designed to minimize the data size transferred and the disk space required to store the container image. A layer is a collection of files corresponding to each operation in a Dockerfile, the script for creating the container image. If a portion of the Dockerfile is modified, the layers preceding the modification are preserved, requiring the transfer of only the updated layers rather than the entire container image. This structure also facilitates the creation of other container images based on an existing image. In such instances, an image builder stacks layers containing modified or newly added files onto the existing container image. When multiple container images utilize the same base image, the layers of the base image are shared, thus reducing data transfer and disk space usage.

Although layer sharing is intended to decrease data transfer and disk space consumption, it is seldom practiced. According to a survey [151] on DockerHub, a container image registry, 90% of all layers are referenced by a single container image, and 99.4% of layers have file duplication with other layers. Moreover, if a file deletion operation is described in the Dockerfile, those files are not deleted but remain in a pseudo-deleted state. Consequently, unused files persist in the layer, and this leads to container image bloat. To mitigate such issues, a multi-stage build [152] isolates the necessary build files, and a technique [153] installs packages and removes

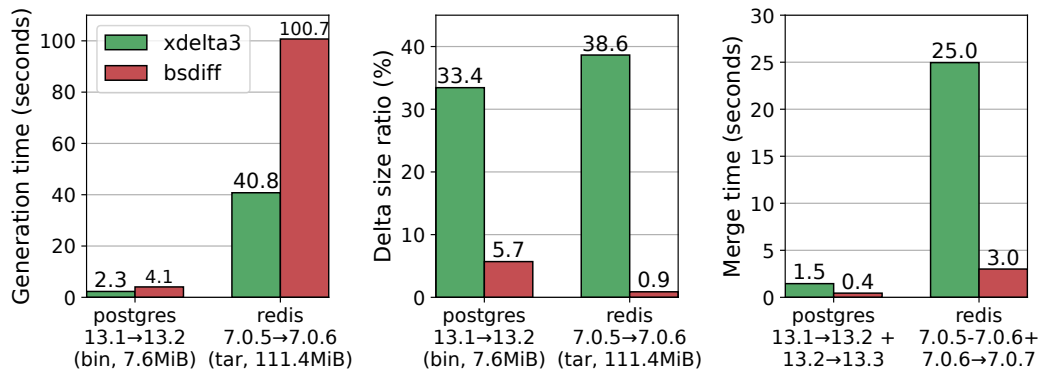


Fig. 5.1: Comparison of delta encodings

superfluous files in a single operation to minimize the number of files included in the layer. However, these efforts do not always result in efficient image updates.

Starlight [63] proposes an approach to accelerate container image pulling. It does not employ the layer-based image format. It transfers only the updated or added files by comparing files in entire container images. While layer sharing does not enhance efficiency, eliminating layers proves more efficient when distributing container images to numerous servers or devices. However, since Starlight transfers the entire content of updated files, there remains potential for reducing file size. D4C employs delta encodings to reduce the amount of data transferred and the disk space required for images.

5.2.3 Issues in Applying Delta Encoding to Image Updating

Delta encodings are algorithms that extract and apply changes between old and new data. Obtaining these changes involves finding a common subsequence (CS) with pattern matching, and there are numerous algorithms for this task. Two typical approaches are chunk-wise checksums to find a CS and indexing the entire file to obtain the longest CS. VCDIFF [154] is a well-known algorithm of the former. xdelta3 [155] is a notable implementation of VCDIFF. bsdiff [156] represents the latter, using a suffix array to obtain the LCS. These implementations are used to transfer or synchronize files with minimal data transfer.

Each delta encoding algorithm has unique characteristics as shown in Fig. 5.1. I generated deltas with xdelta3 and bsdiff for the postgres binary (bin) and a decompressed redis image (tar). The delta size ratio in Fig. 5.1 is defined as the generated delta size divided by the new binary or tar file size. xdelta3 generates

deltas quickly but has poor compression efficiency, whereas `bsdiff` offers better compression efficiency at the cost of increased time. For redis tarball files, `bsdiff` required significant time to generate deltas. The computational resources and time needed to generate deltas increase with file size. This makes delta encodings impractical for fast container image updates when applied directly. In turn, the right figure in Fig. 5.1 shows that merging deltas does not significantly increase time compared to generating deltas from scratch. This property is useful to accelerate delta generation.

Distributing layers in tarball format also poses challenges, especially in IoT environments. The tarball format combines files and directories into a single file and compresses it using compression formats like `gzip` or `xz` before distribution. However, IoT devices, such as the Raspberry Pi, require considerable time to decompress these formats due to limited storage and CPU performance [142]. Thus, a distribution format like `stargz` [64], which is used in lazy-pulling and allows access to included files without decompressing the entire image, is necessary.

In summary, the following issues exist in the distribution of container image updates with delta encodings:

- Fast and highly compressed delta generation considering client-owned images
- A distribution format that does not require decompression of the entire image

The approaches in Section 5.4.1 address these issues.

5.3 Related Work

5.3.1 Reduction in Data Size and Time Required for Pull

Several improvements have been proposed to reduce the size or time required to update container images. These reductions can be divided into two categories: reducing the container image size and improving the pull process itself.

One category focuses on reducing the size of the container image itself. Various approaches have been proposed [157, 158, 159] to eliminate files that are no longer needed in the layer or are duplicated between layers. Skourtis et al. [157] formulated the problem of image size reduction based on the trend of files included in a layer and proposed a method to merge similar layers between container images. Lu et al. [158] proposed a method to detect operations that cause image bloat in

the layer structure, such as deleting downloaded files in image builds. However, Starlight [63] pointed out that improving efficiency in a layer structure is difficult.

The other category involves improvements to the pull process itself, specifically the operation of downloading and decompressing the compressed container image. This approach is divided into three categories: not pulling container images, partially and lazily pulling container images, and using a dedicated system for updating container images.

Slacker [143], Cider [144], Wharf [145], and CernVM-FS [160] mount container images on shared storage or distributed filesystems. However, these methods require a stable network and shared storage, making them unsuitable for remote hosts in terms of availability. P2P-based methods [161, 162, 163, 164, 165] have also been proposed, taking advantage of each host having images. However, these are ineffective if network resources are limited.

An approach called lazy-pulling exploits a container-specific characteristic to reduce pulling time. When starting a container, downloading and decompressing the container image accounts for 76% of the startup time, while the files needed to start the container account for only 6.4% of the total image pulling time [143]. Lazy-pulling prioritizes downloading files necessary to start the container. `stargz-snapshotter` [64], `FogDocker` [146], and Gotanda et al.'s approach [147] employ lazy-pulling. These methods record essential files for container startup during execution and optimize downloading based on these records. `stargz-snapshotter` creates and distributes container images using `stargz` [166], a `gz`-compatible compression format that enables files to be read without full decompression. This allows lazily pulled images to be used like conventional ones. However, these approaches inherently suffer from the same problem as the first category. If the network becomes unavailable, required files cannot be read, and processes in a container may hang or abort.

Starlight [63] aims to achieve fast updates and startups, even in environments with limited network resources, by transferring only the updated files. This method compares container images in a local and a remote registry, and transfers only updated files. However, when a large file is updated, the entire file must be downloaded even if the change is small. This inflates the update data size. Such an approach is undesirable in environments with limited network bandwidth, such as WANs [148] or mobile networks [149]. To provide more efficient and fast updating, it is necessary to reduce the size of deltas. Additionally, Starlight generates deltas on demand based on the container images in the client. It increases the server's load and updating time. This is also true for delta encodings. While smaller deltas reduce data transfer,

Tab. 5.1: Comparison between applications with delta encoding

	Content type	Granularity of update	Protocol	When applying deltas
Web [171, 172]	a few files	a file	batch	immediately
Firmware [173] Application [174] Updating	a bundle with a few files	a bundle	batch	immediately
File Sync [175, 176] Remote File System [177]	many files	a file	sequential	immediately
Container Image Updating	a bundle with many files	a bundle	batch	lazily allowed

they can increase the time to generate deltas. This increases overall update time. It is necessary to consider how to generate the required deltas in a short time.

DADI [167], Nydus [168], and `zstd:chunked` [169] employ chunk-based deduplication to reduce the size of container images. In contrast, D4C employs various delta encoding algorithms, including non-chunked methods such as `bsdiff`, to achieve higher compression ratios. Although the techniques proposed in these works could be beneficial for improving D4C, they are outside the scope of this chapter.

I previously proposed a delta encoding-based updating approach [170]. However, it cannot provide faster updating in some cases and cannot handle large images. In D4C, I improved the method with approaches including an improved merge strategy and a multithreaded process considering large images. Additionally, more evaluations were performed for delta generation and Di3FS. The results show that the approach provides faster updating than the previous approach.

5.3.2 Content Distribution Using Delta Encoding

Table 5.1 compares applications with delta encoding and container image updating. WebExpress [171] and Banga et al. [172] proposed extracting deltas for each URL-tagged content. Delta encoding is also used in file synchronization, such as `rsync` [175] and `NetSync` [176]. A server calculates hash values for each chunk and sends them to a client. The client sends the unmatched chunks to the server, allowing files to be synchronized with minimal data transfer. LBFS [177] is a file system enabling fast file operations even in low-bandwidth networks with delta encoding and client caches.

These approaches in Web and firmware or application updating handle a few files or a single bundle containing a few files, so the time to generate deltas is short

or not considered. Unlike them, container images contain various files, such as executable binaries and compressed files. In container image updating, servers must generate deltas for many files quickly to reduce updating time. An approach is needed to generate deltas tailored to each client rapidly. Additionally, container image updating has a unique characteristic: there is no need to apply all deltas when the updated images are used, and applying deltas lazily is acceptable like lazy-pulling. This characteristic is beneficial for faster updating.

In terms of a protocol, file synchronization and remote file system methods use a sequential transfer protocol that employs a hash to detect different chunks between the client and server and transfers the deltas sequentially. This protocol is designed to work for files that are being modified. However, Chen et al. [63] pointed out that such a sequential protocol can cause performance degradation when provisioning container images. In container image updating, each image is bundled as a static file. It is possible to generate deltas in advance and bundle them into a single file. Therefore, the protocol for container image updating should use a batch protocol that transfers all deltas as a single bundle.

5.4 Design of D4C

5.4.1 Approaches to Apply Delta Encodings

This section summarizes the issues of applying delta encodings to container image updates in Section 5.2.3. D4C addresses these issues with the following approaches.

For fast and highly compressed delta generation for container images, I propose two methods: **file-oriented delta generation** and a **merge-based delta generation strategy**.

In file-oriented delta generation, deltas are generated for each file within the container image instead of generating a delta for the entire container image. As shown in Fig. 5.1, larger files require more time for delta generation. By targeting individual files and handling them in parallel, D4C reduces the time to generate deltas.

While file-oriented delta generation accelerates delta generation, it is still necessary to avoid generating deltas individually for each client. As noted in Section 5.2.3, merging deltas does not take significant time compared to generating deltas from scratch. Utilizing this characteristic, D4C employs a merge-based delta generation strategy. Deltas are generated in advance and then merged based on the client's

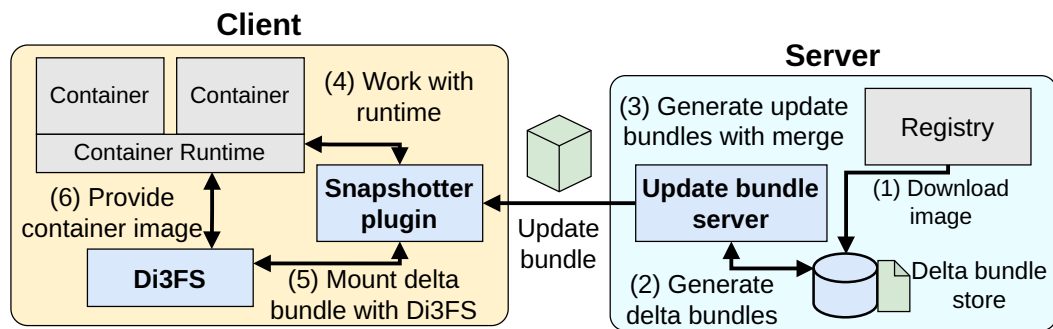


Fig. 5.2: Overview of D4C

images. Despite the computational intensity of delta encoding, merging deltas bypasses the need for finding common subsequences, thus enabling high-speed delta generation. Although `xdelta3` supports merging deltas, `bsdiff` does not. Therefore, I propose an algorithm called **DeltaMerging** to merge `bsdiff`'s deltas.

For an image format that does not require decompression of the entire image, I propose a dedicated format and **Di3FS** utilizing the `estargz` approach. The `estargz` format compresses each file independently, allowing file operations without decompressing the entire image. D4C uses this format as a bundle of deltas and accesses each file via a dedicated file system called **Di3FS**. **Di3FS** provides updated files by applying deltas contained in a bundle.

5.4.2 Overview of D4C

Fig. 5.2 shows the D4C system. Deltas between two container images are bundled into a single file called a `delta bundle`. A `delta bundle` consists of deltas and metadata, as detailed in Section 5.4.3. An `update bundle`, comprising a `delta bundle` and metadata necessary for launching containers, is generated based on `delta bundles`, and the client uses the `update bundle` to perform update operations.

D4C deploys components on both the client and server sides for transferring `update bundles`.

The server includes an `update bundle server` and a `delta bundle store`. According to the `delta bundle` generation strategy outlined in Section 5.4.4, `delta bundles` are pre-generated and stored in the `delta bundle store`. The D4C strategy requires three `delta` encoding functions. *Generation of a delta* produces a `delta` from two data sets. *Application of a delta* creates new data from old data and a `delta`. *Merging of deltas* creates a `delta` from two existing `deltas`. D4C is compatible with

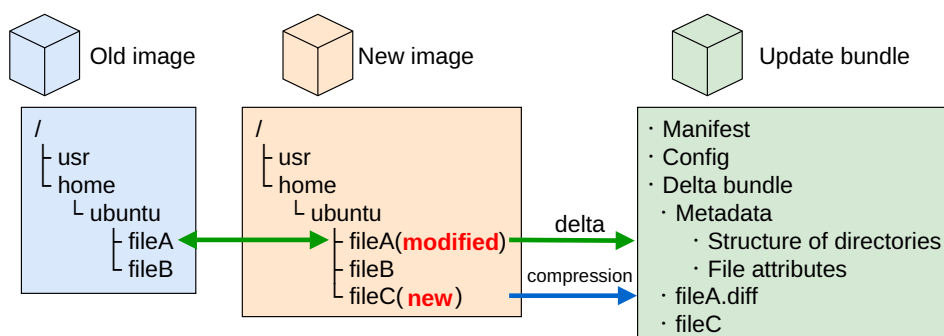


Fig. 5.3: Overview of delta bundle generation

any delta encoding that supports these functions. I used `xdelta3` and `bsdiff`. The update bundle server generates an update bundle based on the client's existing images and transfers it to the client. As discussed in Section 5.3, the sequential transfer protocol used in file synchronization and remote file systems is unsuitable for container image updates. Instead, D4C employs a batch transfer protocol. It sends the entire update bundle at once.

The client comprises a `snapshotter` plugin, which works with the container runtime, and `Di3FS`, a file system providing files for containers with update bundles. The `snapshotter` plugin presents updated container images as a snapshot, allowing the container runtime to use the updated images via `Di3FS`. `Di3FS` provides new images based on the update bundle without decompressing it entirely. The new image is mounted using `Di3FS` with the provided update bundle, and a container is launched via `Di3FS`.

5.4.3 Generating Delta Bundles

Fig. 5.3 shows the process of generating delta bundles from container images. D4C bypasses the traditional layer structure by combining all layers into a single layer.

D4C manages container images and delta bundles based on their contents, using hash values (e.g., `sha256:a4f...`) as image IDs instead of tags (e.g., `nginx:1.2`). This ensures proper handling of container images even if the tag changes. Each delta bundle contains the IDs of the new and old images: the old image's ID indicates the source image for applying deltas, while the new image's ID denotes the updated image.

D4C generates deltas by comparing files between the old and new images. Deltas created through delta encoding are termed *delta files*. If a file is newly created, the

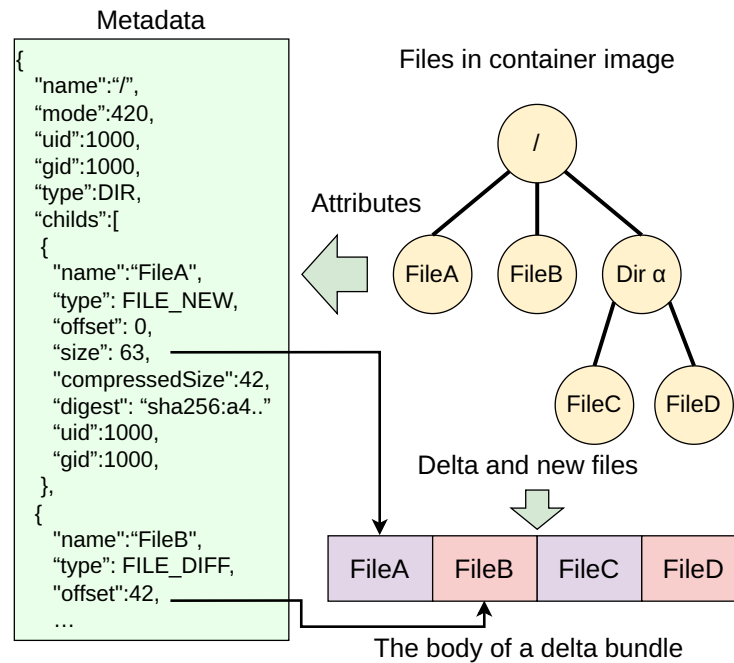


Fig. 5.4: Structure of a delta bundle

entire file is compressed and copied, referred to as a *new file*. Delta bundles include only files present in the new image. During delta bundle generation, the directory structure and file attributes (name, size, permissions, owner) of the new image are preserved as metadata. To accelerate the delta generation, D4C utilizes multiple threads, independently generating deltas for each file in each thread. Writing results to a delta bundle, which requires specific offsets, is performed by a single thread to maintain consistency.

As discussed in Section 5.2.3, delta bundles must be accessible without decompression. The delta bundle format utilizes a key structure used in existing methods [64, 63]. Delta bundle metadata includes file attributes and offsets. It provides the list of files and their attributes without reading file content. Metadata is placed at the head of the delta bundle, with delta and new files concatenated at the tail, forming a single file that can be accessed without decompression.

To safeguard against corruption or tampering, delta bundle metadata includes hash values for each file and directory. For files, D4C calculates a SHA256 hash of both file attributes and the updated file body, storing it in the metadata. For directories, D4C computes a hash value from the metadata, incorporating the hash values of the directory's children. When opening an updated directory or file, D4C verifies the integrity of the directory structures, each file's attributes, and the content by validating the hash values.

Tab. 5.2: Types of entry in metadata

Type	Meaning	Content
FILE_NEW	Newly created file	Attributes and new file
FILE_SAME	Nothing modified file	Attributes
FILE_DIFF	Something modified file	Attributes and delta file
DIR	Directory	Attributes with its child
SYMLINK	Symlink	Attributes with real path
HARDLINK	Hardlink	Attributes with resolved target path

The entry types included in a delta bundle are listed in Table 5.2. Tarballs can include hardlinks, which must be managed to prevent unnecessary increases in delta size. D4C generates delta bundles directly from a tarball image. When encountering a hardlink entry during delta bundle generation, D4C resolves the target path and records it in the file attribute. Upon mounting the delta bundle, Di3FS links the HARDLINK entry's inode in the same manner as traditional file systems.

5.4.4 Merge-based Delta Bundle Generation Strategy

D4C generates the required delta bundles by merging multiple delta bundles, taking into account the client's old images. One characteristic of updating using delta encodings is that delta files require old files, which poses a challenge for methods with delta encodings. Let V_i denote image tags and $\Delta_{(V_i, V_j)}$ represent the delta bundle from $Image_{V_i}$ to $Image_{V_j}$. When a client has $Image_{V_1}$ locally and needs to update to $Image_{V_3}$, skipping $Image_{V_2}$, there are three approaches to provide the delta bundle. The first is that a server transfers both $\Delta_{(V_1, V_2)}$ and $\Delta_{(V_2, V_3)}$. However, the total size of bundles can exceed the size of the new container image [178]. The second is to generate a new delta bundle $\Delta_{(V_1, V_3)}$ from $Image_{V_1}$ to $Image_{V_3}$. Generating delta bundles upon request or in advance becomes impractical as the number of candidate delta bundles increases. The third method is to generate delta bundles $\Delta_{(V_i, V_{i+1})}$ in advance and generate the required delta bundle by merging them. This method leverages the fact that merging deltas is fast as shown in Fig. 5.1. It enables the quick generation of the required delta bundle. D4C uses the third approach.

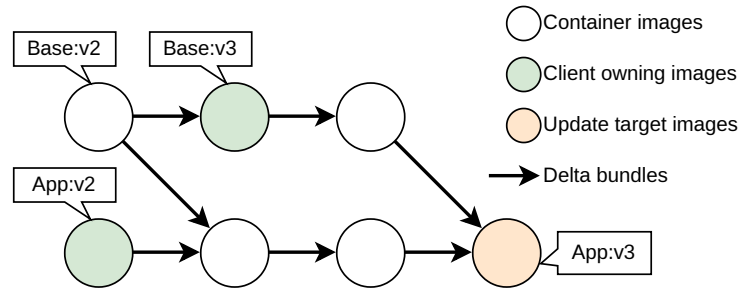


Fig. 5.5: Example DAG-based merge graph

To merge delta bundles, D4C needs to determine which delta bundles to merge. Each delta bundle contains the new and old image IDs, which form a Directed Acyclic Graph (DAG) where nodes represent container images and edges represent delta bundles. Fig. 5.5 shows the example. Green nodes represent the images retained by the client. Red nodes represent the images being requested. Edges are delta bundles and indicate the direction from the old image to the new image. When requesting a new image, the client provides information about all container images it holds, including both complete images and those provided via delta bundles. The server uses a breadth-first search (BFS) algorithm to calculate the shortest path from the requested image to one of the client's images. The BFS algorithm treats edges in reverse, starting the search from the updated target image node and continuing until it reaches a client image node or visits all image nodes. This approach identifies the smallest combination of delta bundles needed to update from the image held by the client to the requested image. In the example, the algorithm finds the path from the `App:v3` node to the `Base:v3` node. If two or more delta bundles are identified, they are merged and provided to the client.

Merging delta bundles in D4C is optimized through a two-step parallelization to increase speed. The first step is file-level parallelization. Each file is merged in parallel, similar to the approach used in delta bundle generation. The second step is bundle-level parallelization. This employs a divide-and-conquer strategy to merge delta bundles. This method further enhances efficiency by breaking down the problem into smaller, more manageable sub-problems that can be solved concurrently. By utilizing these two parallelization strategies, D4C ensures that the merging process is both efficient and scalable, significantly reducing the time required to generate delta bundles tailored to the client's images. The merge of delta bundles is formulated as follows. Let Δ be the merged result and D_i ($i \in \mathbf{N}$) be the delta bundle in the path from the client's image to the requesting image.

Tab. 5.3: Operations for MergeDeltaBundle

Lower entry	Upper entry	Operation
Not existing FILE_SAME SYMLINK HARDLINK	Any file types	Copy the upper file
Any file types	FILE_NEW SYMLINK HARDLINK	Copy the upper file
Any file types	FILE_SAME	Copy the lower file
DIR	DIR	Recursively merge
Not DIR	DIR	Copy the upper file
FILE_NEW	FILE_DIFF	Apply upper diff to lower file and copy as FILE_NEW
FILE_DIFF	FILE_DIFF	Merge delta files (DeltaMerging)

$$\Delta = \text{Merge}(i, j) \ (i \leq j) = \begin{cases} D_i & (j = i) \\ \text{MergeDeltaBundle}(D_i, D_j) & (j = i + 1) \\ \text{Merge}(\text{Merge}(D_i, D_k), \text{Merge}(D_{k+1}, D_j)) & (k = \lfloor \frac{i+j}{2} \rfloor, k \in \mathbf{N}) \end{cases}$$

In this algorithm, $\text{Merge}(D_i, D_k)$ and $\text{Merge}(D_{k+1}, D_j)$ are merged in parallel because they do not depend on each other. If the number of merged bundles is n , it takes $O(n)$ time when done in series, but with this method, it can be done in $O(\log n)$ with unlimited computing resources.

Table 5.3 outlines the operations for the MergeDeltaBundle process. In this process, entries in the older delta bundle are referred to as *lower entries*, and entries in the newer delta bundle are referred to as *upper entries*. Each upper entry is processed according to the rules specified in the table, and the results are stored as the merged delta bundle. Two FILE_DIFF entries are merged using the delta encoding's merge function. xdelta3 provides it, but bsdiff does not. To merge delta files generated by bsdiff, I propose DeltaMerging.

5.4.5 Merging bsdiff's Delta Files

DeltaMerging is an algorithm to merge delta files generated by bsdiff into a single delta file, similar to the algorithm [179] for VCDIFF. This algorithm enables high-speed delta file generation without regenerating delta files from scratch. Let $\delta_{(V_i, V_{i+1})}^{FileA}$

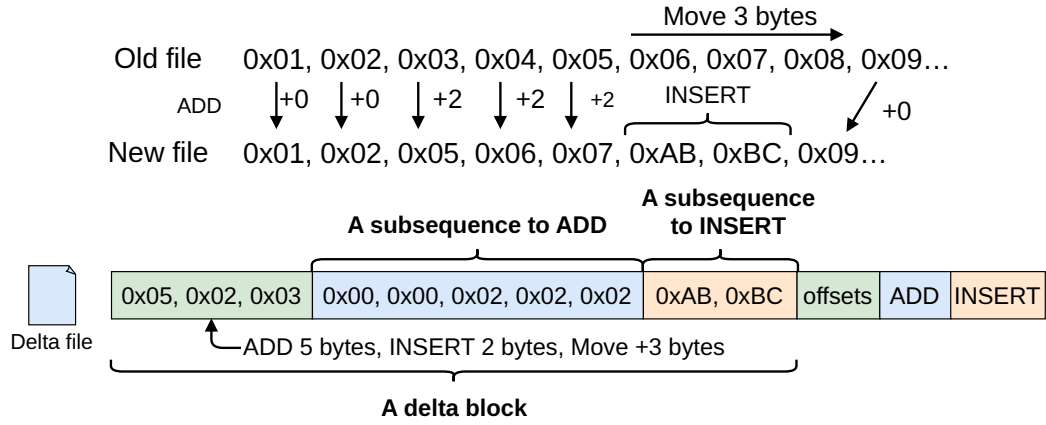


Fig. 5.6: Format of a delta file in bsdiff

be the delta file for file *FileA* contained in the delta bundle $\Delta_{(V_i, V_{i+1})}$. DeltaMerging is formulated as follows.

$$\delta_{(V_i, V_{i+2})}^{FileA} = \text{DeltaMerging}(\delta_{(V_i, V_{i+1})}^{FileA}, \delta_{(V_{i+1}, V_{i+2})}^{FileA})$$

Fig. 5.6 illustrates the structure of the delta file produced by bsdiff. In bsdiff, the delta is managed through three operations: adding (ADD), inserting (INSERT) subsequences into the old file, and adjusting the delta's offset within the file. This set of operations is termed a DeltaBlock. During merging, the delta blocks are transformed into the structures depicted below, and the entire delta file is treated as an array of DeltaBlocks.

```

type DeltaBlock = struct {
    oldPos      int64
    newPos      int64
    addBytes     []byte
    insertBytes []byte
}

```

oldPos is the absolute offset in the old file where the subsequence is added, and newPos is the absolute offset where the result of ADD is written to the new file. addBytes and insertBytes are the subsequences for ADD and INSERT.

DeltaMerging is performed after converting an older delta file (lowerFile) and a newer delta file (upperFile) into arrays of DeltaBlocks, namely lowerBlocks and upperBlocks. It integrates all delta blocks in upperBlocks with those in lowerBlocks. Algorithm 1 describes a procedure to merge an upper block with lowerBlocks, and Fig. 5.7 shows the example. It takes a delta block in upperBlocks as upper. cur is the current offset in upper for merge starting from 0. FIND_BLOCK

Algorithm 1 Merge a delta block in DeltaMerging

Require: upper: a delta block to be merged
Require: lowerBlocks: an array of delta blocks for lowerFile
Ensure: merged: an array of merged delta block

```
1 function MERGEBLOCK(upper, lowerBlocks)
2   merged ← NewArrayOfDeltaBlock() // Array for merged DeltaBlocks
3   cur ← 0 // Current offset for merge
4   mergeBlock ← NewDeltaBlock()
5   // mergeBlock has state NotAdded, Added, and Inserted.
6   // NotAdded: mergeBlock is initialized and nothing is merged into it.
7   // Added: Something merged as ADD into mergeBlock.
8   // Inserted: Something merged as INSERT into mergeBlock.
9   // Merged subsequence as ADD cannot be inserted into mergeBlock.
10  // mergeBlock is complete DeltaBlock and appended to merged.
11  state ← NotAdded
12  while cur is in the upper do
13    lower ← FIND_BLOCK(lowerBlocks, upper.oldPos+cur)
14    if state is Added then
15      merged.append(mergeBlock)
16      state ← NotAdded
17    end if
18    while cur is in the upper and the lower do
19      if state is NotAdded then
20        mergeBlock ← NewDeltaBlock()
21      end if
22      if lower is ADD at cur then
23        if upper is ADD at cur then
24          if state is Inserted then
25            merged.append(mergeBlock)
26            mergeBlock ← NewDeltaBlock()
27          end if
28          mergeBlock ← Merge ADD regions as ADD
29          state ← Added
30        else
31          mergeBlock ← Merge ADD and INSERT regions as INSERT
32          state ← Inserted
33        end if
34      else
35        if upper is ADD at cur then
36          mergeBlock ← Merge INSERT and ADD regions as INSERT
37        else
38          mergeBlock ← Merge INSERT regions as INSERT
39        end if
40        state ← Inserted
41      end if
42      cur ← cur + merged length
43    end while
44  end while
45  if state is not NotAdded then
46    merged.append(mergeBlock)
47  end if
48  return merged
49 end function
```

gets the delta block lower that satisfies $\text{lower.newPos} < \text{upper.oldPos} + \text{cur} < \text{lower.insertEnd}$ from lowerBlocks. Then, lower and upper are merged according to the rules described in Table 5.4. The merged delta block (mergeBlock) is stored in an array (merged). After merging all delta blocks in upperBlocks, delta blocks in merged are converted into a merged delta file.

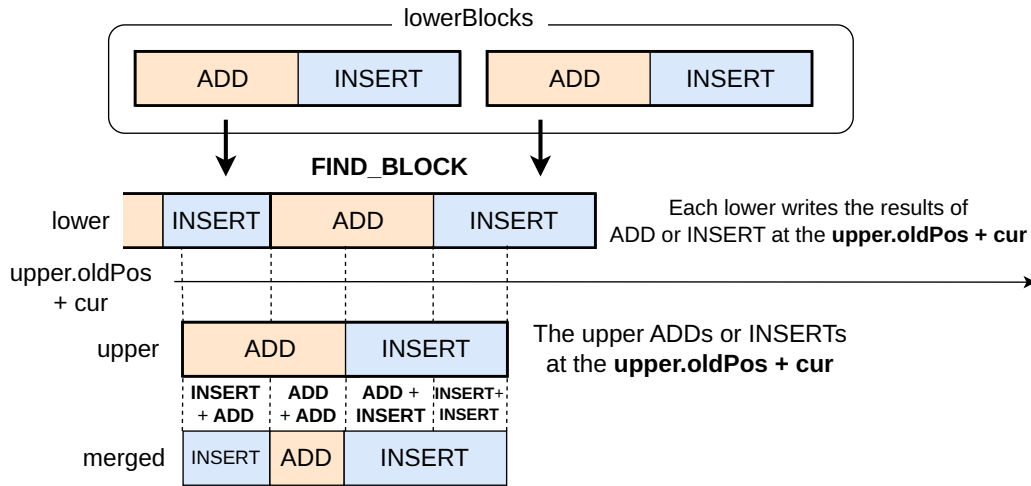


Fig. 5.7: Example of DeltaMerging

Tab. 5.4: Operations to merge delta blocks

lower	upper	Operation
ADD	ADD	Adding each subsequence and storing it as ADD
ADD	INSERT	Storing INSERT of upper
INSERT	ADD	Adding each subsequence and storing it as INSERT
INSERT	INSERT	Storing INSERT of upper

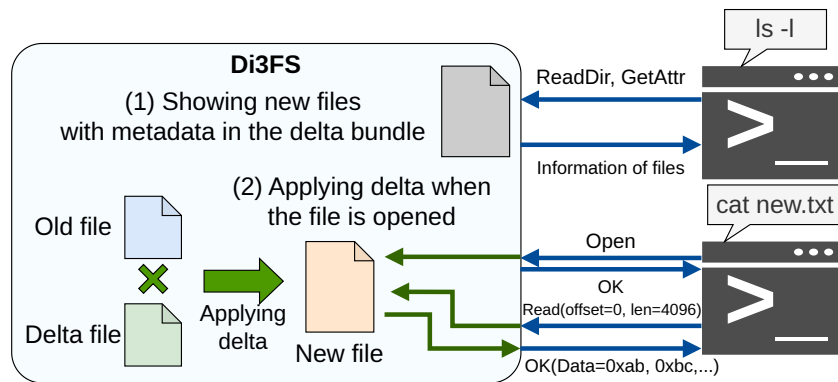


Fig. 5.8: Reading files with Di3FS

5.4.6 Di3FS

Di3FS (DiFF FileSystem) is a file system designed to make a new image available through delta bundles by applying deltas lazily. In delta encodings, an operation to apply the deltas is necessary. The time required to apply all deltas increases with the number or size of target files. This trend is especially evident in environments with low storage performance. Di3FS mitigates this issue by providing the new image

through lazy application of the deltas, thereby reducing the overall time needed to apply them.

Di3FS constructs a tree of files and directories using the delta bundle's metadata and presents them as a file system. During mounting, Di3FS only requires the attribute information of files and directories, eliminating the need to apply the deltas. Fig. 5.8 describes the process from mounting the file system to reading files with Di3FS. When a file is opened, the handler for applying deltas is invoked. The relevant delta file is retrieved from the delta bundle and applied. Since the result can be large (over 1 GB), retaining it in memory is impractical due to high memory consumption. To prevent out-of-memory issues, Di3FS stores the results in local storage. When the handler to read is called, Di3FS reads content from the file with the given offset and length and returns the result.

Di3FS does not handle writing data or modifying file attributes. Instead, a writable layer allocated locally is stacked over Di3FS to store modified files. This functionality is provided by a snapshotter plugin.

5.5 Implementation

I implemented D4C for containerd [97], a widely-used container runtime. The implementation is available at <https://github.com/naoki9911/d4c>.

5.5.1 Di3FS

I implemented Di3FS with FUSE [180]. The FUSE (File system in Userspace) kernel module relays filesystem operations between the kernel and a userland program. File systems are often implemented using libraries. Di3FS was implemented with `go-fuse` [181], which provides abstracted interfaces to provide directory trees and file operation handlers. The procedures to process file open and read operations are described in Section 5.4.6. Di3FS requires a complete image as a base and delta bundles to provide an updated image. The snapshotter plugin manages images and delta bundles and launches Di3FS when the image is requested.

Tab. 5.5: Container images used for evaluations

Image	Version (Size)
postgres [184]	13.1(109.44 MB), 13.2(109.51 MB), 13.3(109.62 MB)
redis [185]	7.0.5(40.43 MB), 7.0.6(40.44 MB), 7.0.7(40.44 MB)
nginx [186]	1.23.1(54.14 MB), 1.23.2(54.19 MB), 1.23.3(54.25 MB)
pytorch [187]	2.2.0(3.41 GB), 2.2.1(3.41 GB), 2.2.2(3.73 GB) with cuda12.1-cudnn8-runtime

5.5.2 Dedicated Snapshotter Plugin

containerd provides a snapshotter plugin [182, 183] for more flexible container image management. containerd manages OCI container images' layers as *snapshots*. Each snapshot corresponds to each image's layer. A snapshotter plugin manages snapshots. It mounts snapshots and registers snapshots' information. When a container is launched, containerd looks for a snapshot of the image and notifies the corresponding plugin to prepare it. The plugin prepares a writable filesystem for the snapshot. Then, containerd continues to launch with the prepared filesystem.

D4C provides a dedicated snapshotter plugin with Di3FS, and it allows users to use D4C without modifying container runtimes. It handles all images as snapshots. When the update is pulled, the plugin mounts it with Di3FS and notifies containerd that the image is available. After that, containerd launches a container with the updated image. containerd requests D4C's plugin to prepare a writable filesystem. The plugin creates a temporary writable directory and stacks it over Di3FS's mounted image with `overlayfs`. This makes the updated image provided by Di3FS writable and the container continues to launch.

5.6 Evaluation

5.6.1 Evaluation Environment

Table 5.5 lists the container images used in the evaluation. These images are available on DockerHub¹. The size refers to the image size noted in each DockerHub image repository. Tags assigned to images are considered as versions.

I prepared the evaluation environment using two virtual machines. Table 5.6 details the specifications. An update bundle server and a client run on the same physical host

¹<https://hub.docker.com/>

Tab. 5.6: Evaluation Environment (Virtual Machine)

CPU	AMD EPYC 7452(8 core)
Memory	32 GB
Storage	SSD 200 GB
	Sequential read 2 GB/s
	Sequential write 700 MB/s
OS	Ubuntu 22.04

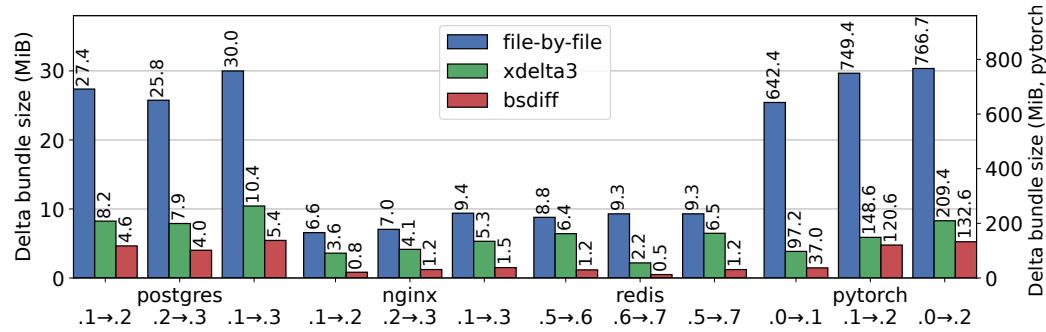


Fig. 5.9: Comparison of delta bundle size

as VMs. Each VM is connected to a virtual network provided by Linux. Throughput and latency are managed using Linux's traffic control (tc).

5.6.2 Generating Delta Bundles

I evaluated the time to generate the delta bundles and their sizes. Starlight [63] compares updated files on a file-by-file basis and transfers the entire updated file. This delta generation method is referred to as *file-by-file* delta. In this approach, FILE_NEW was always generated as deltas instead of FILE_DIFF.

Fig. 5.9 compares the size of delta bundles between file-by-file delta and D4C. For pytorch, the value scale corresponds to the right axis. D4C reduces the size by up to about 95% compared to file-by-file delta. Notably, bsdiff achieved the most efficient compression across all updates. Although xdelta3 generated slightly larger deltas than bsdiff, it still achieved significantly better reductions than file-by-file delta. This efficiency difference stems from the encoding algorithms used in each delta encoding method.

Fig. 5.10 shows the time to generate delta bundles. The delta generation was run with 8 threads, and values represent the average time of 10 measurements. In most cases, file-by-file delta generation was completed in less than 2 seconds. Conversely, D4C with xdelta3 and bsdiff took more than twice as long as file-by-file. Notably,

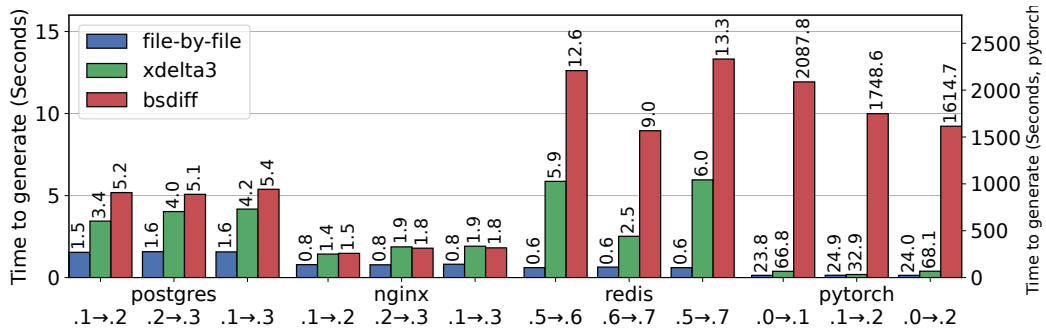


Fig. 5.10: Comparison of time required to generate delta bundles

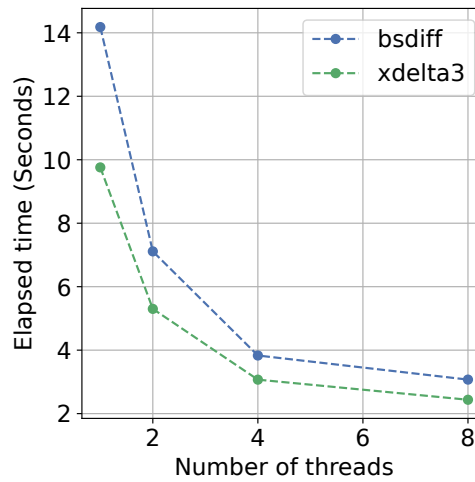


Fig. 5.11: Effect of multithreaded delta bundle generation (postgres with bsdiff)

bsdiff required over 2,000 seconds to generate delta bundles for pytorch. The pytorch images contained files exceeding 1 GB. The bsdiff algorithm operates with a complexity of $O((n + m) \log n)$ [156], where n is the size of the old file and m is the size of the new file. The complexity increases as the file becomes large. Indeed, Fig. 5.1 demonstrated that bsdiff takes longer with larger files. This led to long generation times.

D4C supports multi-threaded delta generation. Fig. 5.11 shows the performance of multithreaded delta generation for postgres 13.1→13.2. Each time was the average of 10 measurements. This approach reduced the time needed to generate delta bundles. However, parallelization is performed on a per-file basis, with each delta file being generated by a single thread. Consequently, if a file requires significant time for delta generation, it limits the overall delta generation time. Achieving faster multi-threaded delta file generation requires enhancements in delta encoding

Tab. 5.7: Files with the greatest size reduction in postgres 13.1→13.2 with bsdiff

path	reduced size (bytes)	ratio
/usr/lib/postgresql/13/bin/postgres	3305635	0.12
/usr/lib/x86_64-linux-gnu/libcrypto.so.1.1	1367715	0.03
/usr/lib/x86_64-linux-gnu/libapt-pkg.so.5.0.2	812333	0.02
/lib/x86_64-linux-gnu/libsystemd.so.0.25.0	307566	0.00
/usr/lib/postgresql/13/bin/psql	233662	0.10

Tab. 5.8: Files with the least size reduction in postgres 13.1→13.2 with bsdiff

path	reduced size (bytes)	ratio
/usr/share/postgresql/13/man/ man7/ALTER_POLICY.7.gz	-1373	2.11
/usr/share/postgresql/13/man/ man7/ALTER_OPERATOR.7.gz	-1362	2.02
/usr/share/postgresql/13/man/ man7/CREATE_OPERATOR_FAMILY.7.gz	-1331	1.96
/usr/share/postgresql/13/man/ man7/DROP_DATABASE.7.gz	-1312	2.01
/usr/share/postgresql/13/man/ man7/DROP_INDEX.7.gz	-1309	1.91

algorithms and their implementations. This is future work. *xdelta3* did not take as much time as *bsdiff*. *xdelta3*'s delta encoding algorithm employs checksum-based string matching [188], which enables rapid delta generation. However, checksum-based string matching does not always yield the longest common subsequence, resulting in slightly larger delta bundle sizes. The findings reveal a trade-off between compression ratio and delta generation time.

Table 5.7 and Table 5.8 show the reduction efficiency between postgres 13.1 and 13.2 using *bsdiff*. These figures compare the sizes of delta files (*FILE_DIFF*) and new files (*FILE_NEW*). The *ratio* is the proportion of a delta file's size to a new file's size. If the ratio exceeds 1, the file-by-file delta is more efficient; if it is less than 1, D4C is more efficient. Table 5.7 shows that executable binaries and shared libraries, such as *postgres* and *libcrypto.so.1.1*, achieved significant size reduction. These files are the primary targets for *bsdiff*. Conversely, Table 5.8 shows that *.gz* (gzip [189]) compressed files exhibit the worst reduction efficiency. For gzip files, the delta file size became larger than the new file size. Compressed files like gzip are encoded using their dictionary, and their content can change drastically. Additionally, gzip employs the Lempel–Ziv algorithm [190], which encodes file contents in a bit-oriented format. *bsdiff* targets binary files encoded in a byte-oriented format and cannot efficiently extract deltas from bit-encoded files. These

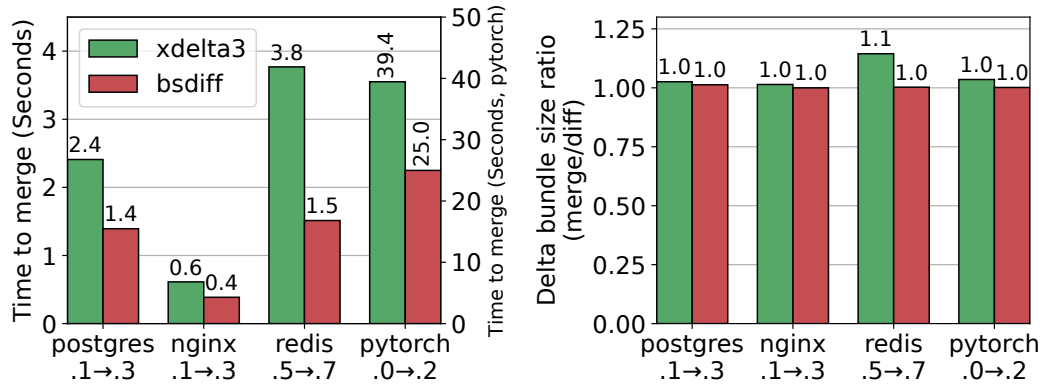


Fig. 5.12: Comparison of time to merge delta bundles and the ratio

factors explain why bsdiff could not extract deltas efficiently from gzip files. To achieve better compression, one approach is to uncompress a gzip file, extract deltas, and then recompress it as an updated gzip file. However, the updated gzip file may change, and it prevents the use of hash-based content validation. For this reason, this approach was not employed.

5.6.3 Merging Delta Bundles

D4C employs a merge-based delta generation strategy. To demonstrate the effectiveness of merging delta bundles rather than generating them from scratch, I measured the time required to merge delta bundles. The merging process was conducted using 8 threads, with each value representing the average time from 10 measurements. The left figure in Fig. 5.12 shows the time taken to merge delta bundles. Each image denoted as $.X \rightarrow .X+2$ indicates that delta bundles for $.X \rightarrow .X+1$ and $.X+1 \rightarrow .X+2$ were merged. Both xdelta3 and bsdiff achieved faster delta generation compared to generating deltas from scratch, as shown in Fig. 5.10. The results were almost as fast as or faster than file-by-file deltas. Specifically, pytorch with bsdiff achieved a $60\times$ acceleration. This demonstrates that the merge-based strategy enables the rapid generation of delta bundles, even for large images.

Results with bsdiff were faster than those with xdelta3. This is likely because D4C executes xdelta3 as a process for each merge. While D4C uses xdelta3 as a library for generating and applying deltas, it does not use it as a library for merging due to xdelta3's merge functions being available only via the command line. This implementation issue could be resolved by implementing xdelta3's merge function as a library, which would eliminate the overhead caused by process spawning and

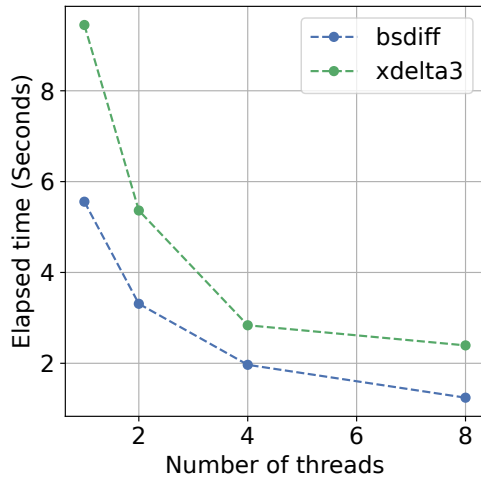


Fig. 5.13: Effect of multithreaded merge (postgres with bsdiff)

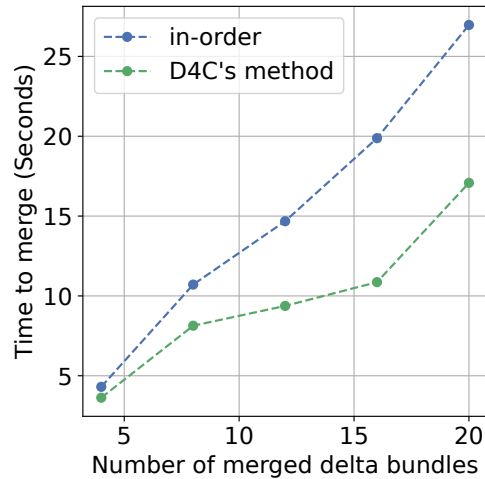


Fig. 5.14: Merge performance with bsdiff

result in faster delta bundle merging. For bsdiff, DeltaMerging was implemented as a library, thus the overhead seen with xdelta3 was avoided.

The right figure in Fig. 5.12 compares the delta bundle size between a merged one and a generated one. If the ratio is greater than 1, generating deltas results in better compression than merging deltas, and vice versa. Except for redis .5→.7 (xdelta3), the ratios were nearly 1.0. This indicates that merging delta bundles provides faster delta bundle generation without increasing the delta size.

D4C provides multithreaded, divide-and-conquer-based merging to efficiently merge delta bundles. As described in Section 5.4.4, this merging process includes two-step parallelization. The first step involves parallel merging delta files. I evaluated the performance of multithreaded merging with delta bundles for postgres 13.1→13.2 and 13.2→13.3. Fig. 5.13 presents the results, showing the average of 10 measurements. As the number of threads increased, the merging time decreased. Unlike delta bundle generation, where performance is limited by the time for large files, merging delta files is relatively fast even for large files, thus reducing the overall merging time.

The second parallelization step involves running multiple merge processes in parallel. Fig. 5.14 compares the time taken to merge delta bundles using a naive in-order approach and D4C's method. The in-order approach merges delta bundles linearly like $\text{Merge}(\text{Merge}(D_i, D_{i+1}), D_{i+2}, \dots)$. 20 nginx images from version 1.20.1 to 1.25.4 were used for this performance measurement. D4C was configured to run each merge process with 8 threads and to run up to 4 merging processes in parallel.

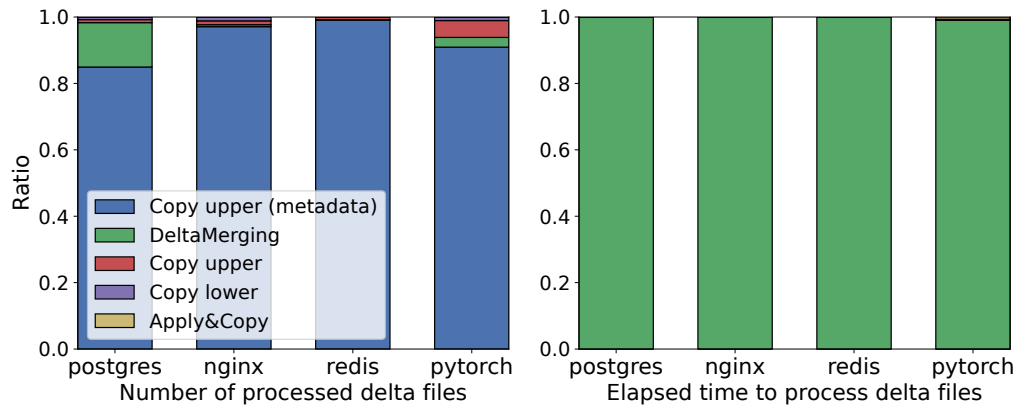


Fig. 5.15: Ratio of merging delta file operations in number and time

It did not achieve $O(\log n)$ scaling because the environment had only 8 cores, but it indicates that D4C's approach can halve the time required to merge delta bundles.

The merge-based strategy enables D4C to quickly provide the necessary delta bundles, reducing the overall pull time even when multiple delta bundles need to be merged.

5.6.4 Breakdown of Merge and DeltaMerging

I performed a benchmark to break down merge and DeltaMerging. Fig. 5.15 shows the ratio of merging delta file operations to merge delta bundles. Most of the operations were not-updated files or links. These files copy only metadata and do not take time. Although DeltaMerging had a smaller ratio than other operations, it accounted for most of the overall processing time. This shows that the performance of merging delta bundles depends on DeltaMerging's performance.

Fig. 5.16 shows each DeltaMerging running time and the relations between the entire processing time and the number of processed delta blocks. It shows that the target file size and processing time were proportional. This result is roughly equivalent to the result in Fig. 5.1. The number of delta blocks did not affect the processing time. These results show that DeltaMerging's processing time depends on the size of merged delta files and the composition of delta blocks in them does not affect it.

Fig. 5.17 shows the breakdown of the operation to merge delta blocks. The merge of ADD+ADD took most of the operation. Other operations were processed fewer times and bytes. This implies bsdiff generates ADD blocks whenever possible. To generate

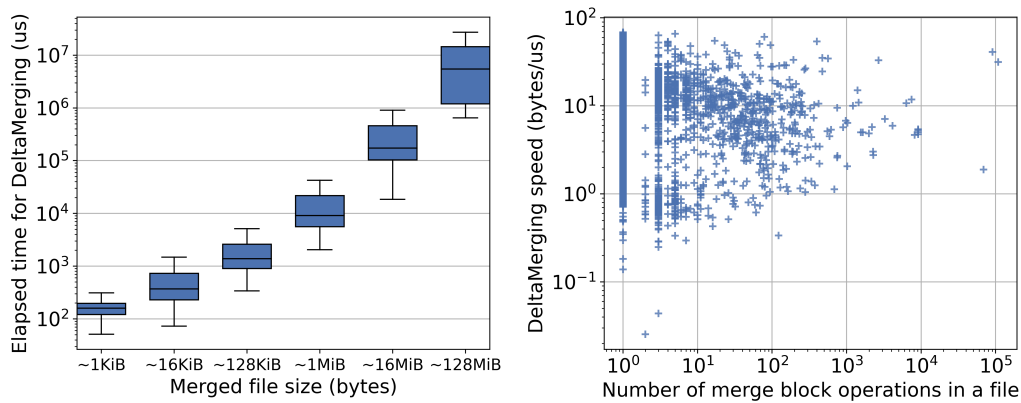


Fig. 5.16: DeltaMerging performance with file size and number of blocks

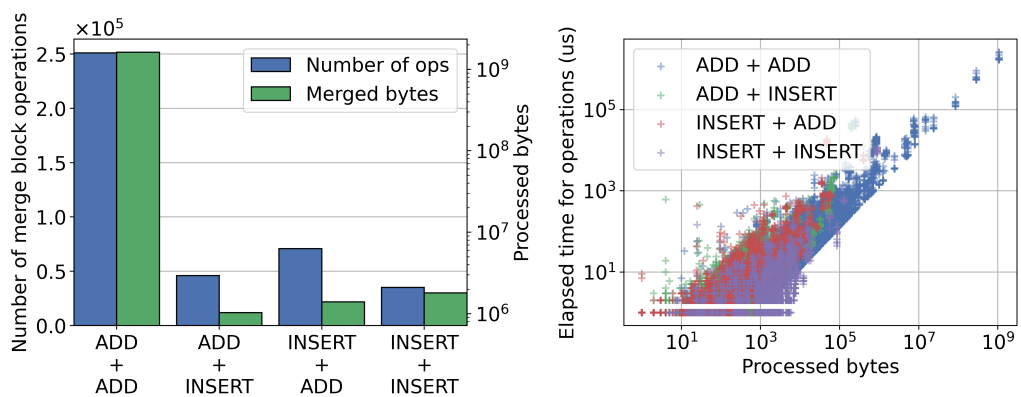


Fig. 5.17: Breakdown of merge delta block operations

these blocks from two raw files, bsdiff must find similar sequences, which is time-consuming. DeltaMerging does not require sequence searching; it simply merges two ADD blocks by concatenation. This is why DeltaMerging provides faster delta generation. Regarding performance, ADD+ADD handled more bytes per operation, reducing per-byte buffer allocation and improving throughput. Because ADD+ADD dominated overall processing time, other operations had little impact on end-to-end performance.

5.6.5 Applying Delta Bundle

I compared the time required to fully apply delta bundles to the old container image against mounting delta bundles with Di3FS. Fig. 5.18 shows the results, averaged over 10 measurements. Both xdelta3 and bsdiff demonstrated that patching all deltas to old images takes over 10 times longer than mounting delta bundles. Specifically,

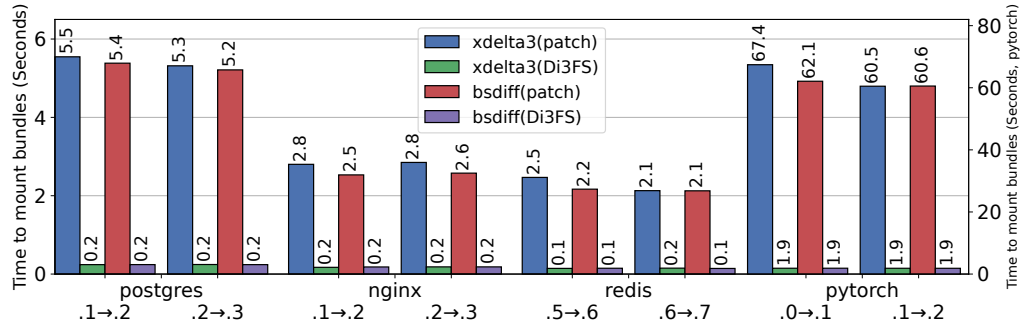


Fig. 5.18: Time to mount delta bundles

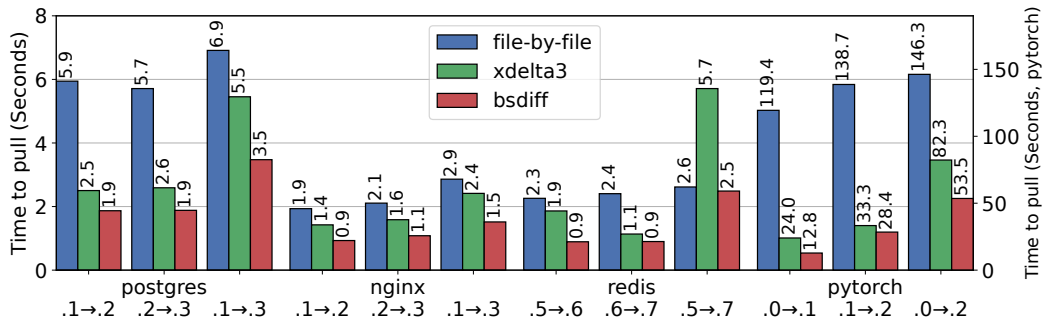


Fig. 5.19: Time to pull update bundles

applying all deltas to pytorch took over 60 seconds, indicating that after downloading the delta bundle, it would take 60 seconds before the updated image is available. D4C enables rapid container image updates by minimizing transferred data and utilizing a decompression-less image format. These results highlight the importance of the decompression-less image format and Di3FS in significantly reducing update times.

5.6.6 Image Pulling with D4C

I measured the time required to update a container image, which is defined as the duration from the start of the container image update to the point when the downloaded update bundle is mounted. The results are averages of 10 measurements. The assumed network between the server and client is typical of mobile networks, such as 4G and 5G, with a bandwidth of 50 Mbps [149] and a 40 ms delay (RTT) [191].

Fig. 5.19 shows the comparison of the time to pull updated images. *Pull* includes both downloading and mounting an update bundle with Di3FS. For updates $.X \rightarrow .X+1$,

the delta bundles are generated in advance. When the update bundle $.X \rightarrow .X+2$ is requested, delta bundles $.X \rightarrow .X+1$ and $.X+1 \rightarrow .X+2$ are merged on the update bundle server.

D4C achieved more than twice the speed of file-by-file delta when pulling pre-generated delta bundles. Especially for `pytorch .0→.1`, it showed almost 10 times faster pull. Additionally, D4C with `bsdiff` demonstrated the same improvement even when delta bundles needed to be merged. However, updating `redis .5→.7` with `xdelta3` took longer than with file-by-file delta and `bsdiff`. This is because the merging cost in this update outweighed the size reduction provided by `xdelta3`. As shown in Fig. 5.9 and Fig. 5.12, the `redis .5→.7` update with `xdelta3` provided only slight compression, and the merge process increased the size. Furthermore, it took over 3 seconds to merge, accounting for about half of the pull time. This case illustrates that using more efficient delta encoding and accelerating the merge process significantly influences the overall pull time.

5.6.7 Di3FS Performance

I measured the performance of file operations in Di3FS. Fig. 5.20 and Fig. 5.21 show the comparison of open and read performance. The target image was `postgres 13.2`, with Di3FS mounting a delta bundle for `postgres 13.1→13.2`. The `postgres 13.2` image was decompressed into an `ext4` file system, referred to as Native. The performance evaluation assessed the latency from opening to reading the entire file. Di3FS applies deltas when a file is opened, thus both open and read latencies were measured to include any overhead. File systems typically have caching mechanisms, and Di3FS stores delta-applied files on a native file system. To evaluate the impact of caching, the latency of the first and subsequent operations was measured, clearing the file system's cache between each measurement.

Fig. 5.20 shows the open latency. Except for the first open with Di3FS, the latencies were stable regardless of file size. For the first open with Di3FS, the open time increased with file size. The second open with Di3FS showed steady latency, as Di3FS applies deltas when the file is first opened and stores the result in the local file system for reuse. The second open with Di3FS had slightly higher latency than Native due to Di3FS metadata management and FUSE's user-kernel boundary overhead.

Fig. 5.21 shows the elapsed time to read the entire file. The second read with Native achieved the best performance. Other reads were 10 times slower for smaller files.

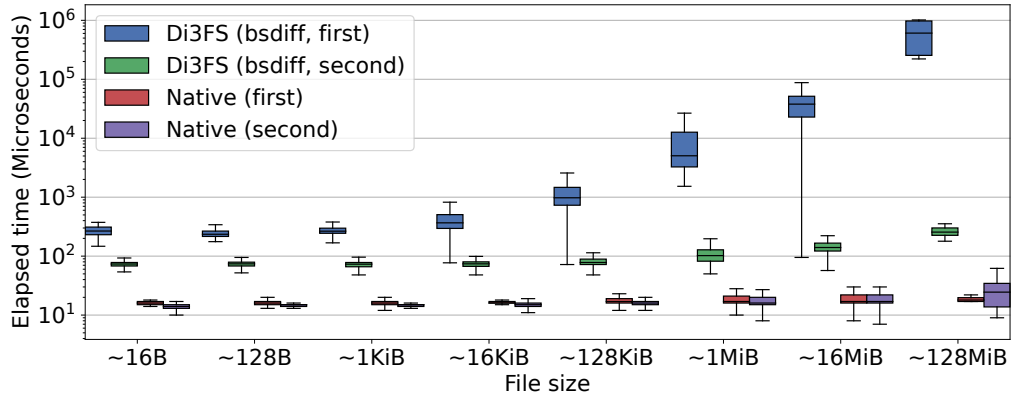


Fig. 5.20: Open latency comparison between Di3FS and Native

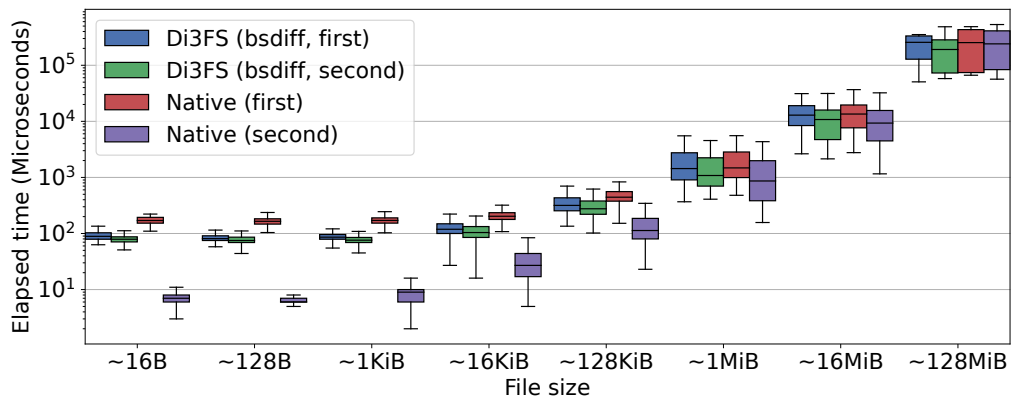


Fig. 5.21: Read performance comparison between Di3FS and Native

As file size increased, performance differences diminished. Di3FS's overhead from FUSE was large for smaller files, as FUSE can handle reads up to 128 KiB in an operation [180]. With larger read sizes, FUSE overhead became less dominant. As a result, Di3FS provided performance similar to Native for larger files.

Fig. 5.20 and Fig. 5.21 indicated some performance degradation with Di3FS. However, Di3FS primarily provides application binaries and libraries, not user data. Typically, once an application binary or shared library is loaded, its content is cached in memory. While file system performance may affect process start times, it does not degrade application performance itself. To confirm this, actual application performance was measured in Section 5.6.8.

Tab. 5.9: TPC-B benchmark comparison in postgres 13.2

Methods	Latency Average(ms)	Transactions Per Second
Di3FS(bsdif)	11.549 $\pm$ 0.722	869.453 $\pm$ 57.484
Native	11.540 $\pm$ 0.843	871.386 $\pm$ 65.541

5.6.8 Impact on Application Performance

I evaluated the impact on application performance caused by D4C. PostgreSQL [192] and PyTorch [193] were chosen as target applications.

Benchmark with PostgreSQL

I measured the impact of D4C on PostgreSQL performance using pgbench [194]. The performance of PostgreSQL launched using Di3FS was compared against that from the native file system using the official postgres 13.2 container image. For Di3FS, a delta bundle postgres 13.1→13.2 was used as an update over the base image postgres 13.1. pgbench is a benchmarking tool for PostgreSQL, capable of simulating real loads. The parameters used were: 10 concurrent connections, 120 seconds of measurement time, database scaling factor of 100, with others at their default values. The benchmark was conducted 10 times with these parameters, and Table 5.9 shows the average values and standard deviation.

The results in Table 5.9 show no significant performance degradation. This indicates that Di3FS does not impact PostgreSQL performance. pgbench issues a series of queries specified by TPC-B [195], characterized by significant disk I/O, moderate system and application execution time, and transaction integrity. In other words, TPC-B involves substantial disk I/O and CPU resource usage for processing queries and transactions. Based on the results and TPC-B characteristics, applications with significant disk I/O and CPU resource consumption will not be affected by Di3FS.

Benchmark with PyTorch

I evaluated the performance of PyTorch. To evaluate its efficiency, pytorch_benchmark [196] was utilized. This tool benchmarks inference using PyTorch models. Specifically, EfficientNet [197], a lightweight model designed for image classification that operates exclusively on CPU, was employed. The pre-trained model efficientnet_b0 provided by PyTorch was used. The target image referenced was pytorch:2.2.1 in Table 5.5. Native containers utilized the official

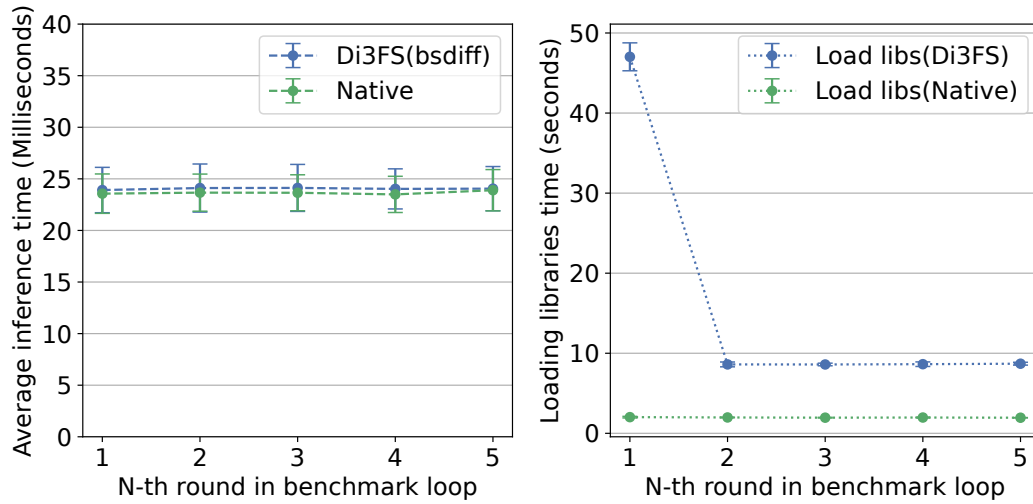


Fig. 5.22: Performance benchmark with PyTorch

image, while Di3FS employed a delta bundle `pytorch 2.2.0→2.2.1` as an update over `pytorch:2.2.0`. The delta encoding method used to create the delta bundle was `bsdifff`. The following Python code was used as the benchmark.

```
import time
start = time.time()

import torch
from torchvision.models import efficientnet_b0
from pytorch_benchmark import benchmark
load_finish = time.time()

model = efficientnet_b0().to("cpu")
sample = torch.randn(1, 3, 224, 224)
benchmark(model, sample, num_runs=1000)
```

This code evaluates inference performance. The inputs consisted of randomly generated images sized 224×224 pixels with three color channels, and the batch size was set to 1. The benchmark ran 10,000 inferences. Additionally, to evaluate the overhead introduced by the lazy delta application, the time required to load PyTorch libraries was measured. This measurement was conducted by executing the same Python code in the identical container 5 times. Upon completion, Di3FS's delta-applied cache was cleared and the measurement process was repeated 10 times.

Fig. 5.22 presents the results. Each value corresponding to the Nth round in the benchmark loop represents an average based on 10 measurements. The left plot indicates that inference performance with Di3FS was nearly identical to that with Native

and was unaffected by Di3FS. However, the right plot reveals that the initial library loading time took approximately 50 seconds, calculated as `load_finish - start` in the benchmark code. Di3FS applies deltas when the corresponding file is accessed. The PyTorch image includes large shared libraries (over 100 MB), which are loaded during Python library imports, causing significant overhead during the first launch of the benchmark code. Subsequent rounds benefited from Di3FS's cached delta-applied files, resulting in shorter loading times than the first round, though still longer than Native. Fig. 5.20 and Fig. 5.21 demonstrate degraded performance even with cached data, contributing to Di3FS's slower loading times compared to Native. In summary, inference performance remained unaffected by Di3FS, but library loading times were longer compared to containers with Native file systems. Considering both pull time in Fig. 5.19 and library loading, Di3FS with bsdiff remained the fastest option. Moreover, the result shows the delta-applied cache mechanism accelerates file operations in practical applications such as library loading.

5.7 Discussion

5.7.1 Delta Bundle Generation with Merge Strategy

To provide fast generation for requested delta bundles, I proposed the merge-based delta bundle generation strategy including DeltaMerging. The evaluation confirmed that it was faster than generating deltas independently, as shown in Fig. 5.10 and Fig. 5.12. Additionally, the size of the merged delta bundle was nearly identical to that of the independently generated delta bundles. This demonstrates that generating client-specific delta bundles at high speed is feasible without increasing their size. The proposed merge-based delta bundle generation strategy enables rapid delta generation even if the delta encoding's delta generation is slow.

In all cases requiring merging, as depicted in Fig. 5.19, D4C achieved faster pull times compared to file-by-file delta generation. With a sufficiently fast network, the time to transfer an update decreases, and merging time becomes dominant. This implies that D4C may not provide faster pulls in high-speed networks such as data center networks. In the evaluation of pull, the worst-case scenario was measured, always requiring merges, and merged delta bundles were not reused. However, in practical use, merges are expected to occur less frequently due to delta bundle reuse. This can significantly reduce pull time. Once a required delta bundle is generated, it is cached, allowing the server to serve the same delta bundle without merging.

All merging is performed based on dependencies in the DAG. New delta bundles generated by merging are added as edges on the DAG and can be reused for future requests. The merged result serves as a "shortcut" on the DAG, applicable not only for the same request but also as a delta bundle for other image updates. Thus, although merging may be necessary for the initial update request, the merged result can be reused for subsequent combinations, greatly reducing pull times and the server's computational merging process. Once the required delta bundles are generated, an update bundle server simply sends the requested update. Downloading the update bundle becomes the dominant factor in the pull process, allowing most devices to benefit from fast pulls.

I proposed the algorithm to minimize the number of merged delta bundles, and it may not produce the best path to minimize delta bundle size. The merge algorithm leaves room to improve, but the impact is considered negligible. Merging delta bundles is time-dominant in pull. Actually, for the update from pytorch 2.2.0 to 2.2.2, Fig. 5.19 shows the entire pull took 53.5 seconds and Fig. 5.12 shows the merge took 25.0 seconds; it was almost half the time of the pull. Furthermore, as shown in Fig. 5.14, merge time increased as the number of merged delta bundles. This is why the algorithm minimizes the number of merged delta bundles. D4C aims to reduce pull time, and this is achieved by reducing the delta bundle size itself and adopting a fast generation approach with merge. Reducing the size of merged delta bundles is important, but other factors, including delta bundle generation, also affect pull time. A merge graph can have multiple paths from the client's image to the updated target image. To choose the best path, it is necessary to predict the size of the merged delta bundle from multiple candidates. This would require additional time, and the size may not shrink much. The algorithm is very simple, but it actually reduces pull time, and Fig. 5.12 shows that the merged delta bundles were not bloated compared to ones without merge. More sophisticated algorithms may produce both size efficiency and fast merges, but that is not in the scope of this work.

To provide more efficient and faster container updates, future work will focus on more sophisticated delta bundle management as discussed.

5.7.2 Overhead and Application Performance with Di3FS

Di3FS mitigates the overhead of applying deltas, and the evaluation confirmed that container images can be used via lazy delta application. Specifically, in environments with low storage performance, while it took about 60 seconds to apply all deltas,

Di3FS was able to mount the updated image in about 2 seconds. Di3FS plays an essential role in accelerating container image updates. However, it caused some performance degradation in file operations, particularly during the initial file open, which becomes significantly slower than with a native file system. This slowdown is due to Di3FS applying deltas upon file opening, a trade-off between fast mounting and lazy delta application. Subsequent file opens and reads also experienced some performance degradation. In these instances, D4C merely opens and reads cached delta-applied files. The primary cause of the overhead is considered to be the handling of FUSE operations, including reading cached files and replying with the read result. Implementing Di3FS as a kernel module could eliminate FUSE's overhead, but due to the numerous features Di3FS includes, such an approach could introduce vulnerabilities. Since the current Di3FS does not impact application performance significantly, this performance degradation is considered negligible.

Although some performance degradation was observed in file operations, benchmarks targeting PostgreSQL and PyTorch demonstrated that Di3FS did not affect their performance. As detailed in Section 5.6.8, the PostgreSQL benchmark involved substantial disk I/O and CPU resources to process queries and transactions. This indicates that applications that do not frequently read files from the container image are unaffected by Di3FS, even if they have significant disk I/O and CPU resource consumption. In D4C, files in delta bundles are provided by Di3FS as a read-only layer. Applications in containers create files, and these files are stored in a temporary read-write layer or host volumes provided by a native file system. Di3FS does not interact with these operations, which is why performance degradation was not observed.

In the PyTorch benchmark, the first library load took significantly longer compared to subsequent loads and the native file system. This was caused by Di3FS's lazy delta application, potentially hindering fast image updating in terms of service availability. To address this issue, several approaches can be considered. Applying deltas speculatively in the background and applying only requested regions can reduce open latency. These improvements are left for future work.

5.7.3 Lower CPU or Storage Environment

D4C is designed for edge servers and devices. Such devices may have lower CPU or storage than those used in Section 5.6.

With lower CPU performance, the file open latency will be longer, but D4C and Di3FS will not degrade actual application performance. The main CPU-intensive

process in a client is to apply deltas to an old image. Di3FS employs an approach to apply deltas lazily to remove time and CPU resource-consuming entire image decompression [142] and all deltas patching. In return, it brings longer open latency as shown in Fig. 5.20. With lower CPU performance, this latency will be longer. It may cause the first-time process launching and library loading to be longer. After the application, the result is retained on local storage and Di3FS does not require much CPU resources to process file operations. Results in Section 5.6.8 show that Di3FS almost does not affect the actual application performance. Aside from startup time, performance is not expected to degrade even in lower-CPU environments.

As for storage, devices can use slower storage interfaces like SATA3 [198], but the slower interfaces will not be a bottleneck in Di3FS. In the update from PostgreSQL 13.1 to 13.2, Di3FS requires at most 105 MiB/s to open a file. SATA3 provides a 6 Gbps interface, effectively about 4.8 Gbps (572 MiB/s). This will not be a bottleneck in Di3FS's open process. Additionally, the patched result is written to local storage, and the kernel may cache the content in memory. Table 5.9 and Fig. 5.22 show that Di3FS does not affect performance thanks to them.

5.7.4 Practicality in Daily Operations

To provide better efficiency and faster image updating in daily operations, there are two operational considerations.

One is that merge-based delta bundle generation requires pre-generated deltas. D4C assumes that users generate adjacent deltas between $.X \rightarrow .X+1$. It generates deltas when users upload a new image to an image registry. However, as shown in Fig. 5.10, generating deltas takes time. If the deltas are not used, this wastes time and CPU resources. This is a kind of speculative approach to accelerate the generation of required delta bundles. D4C does not provide mechanisms to reduce such waste, but users can reduce it through operational choices. If the image $.X+1$ is not used, users may skip generation of a delta for $.X \rightarrow .X+1$ and generate the next delta for $.X \rightarrow .X+2$. This reduces wasted time and CPU resources. As a trade-off, D4C cannot provide updates when a delta for $.X \rightarrow .X+1$ is requested. Whether this trade-off is acceptable depends on user requirements. D4C provides operational flexibility to reduce wasted resources.

The other is that Di3FS applies deltas lazily. The evaluations showed that it does not affect application performance but raises concerns about additional CPU consumption to apply deltas. If it needs to apply deltas after reboots, it results in higher

CPU usage or longer startup on reboots. Di3FS retains delta-applied results on local storage and uses them as long as they are available. Their permanency depends on the storage. They can be made persistent and reused after reboots, which will reduce CPU usage or startup time on reboots. After the delta is applied, Di3FS relays read requests from processes to the file on local storage. The read performance without the kernel's cache mainly depends on the local storage performance. This is the same as a traditional container image, and the impact on application performance is not unique to Di3FS.

5.8 Conclusion

In this chapter, I proposed D4C, a method to reduce the amount of data transferred and the time for container image updating using delta encodings. The evaluation demonstrated that D4C reduced the size of the delta bundle, which contains deltas for a container image, by up to about 95% and provided approximately 10× faster updates compared to the file-by-file approach. When updating container images using binary delta encoding, it is essential to quickly generate the update bundle for each client. Clients need to make the updated image available as soon as possible. To address these issues, I proposed a merge-based delta bundle generation strategy and Di3FS. The strategy accelerates the generation of the required delta bundle by merging multiple delta bundles rather than generating it from scratch. Di3FS provides updated images by applying deltas lazily, eliminating the time-consuming operations of decompressing the delta bundle and applying deltas. The evaluation confirmed that the performance degradation of applications with D4C is limited and acceptable.

Data Analysis Platform with Data Provider's Control

This chapter is based on a paper “yak: WebAssembly-based Data Analysis Platform with Data Providers Driven Analysis Control” published at COMPSAC 2025, ©2025 IEEE. Reprinted with permission from [199].

Copyright Notice: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Kyoto University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

6.1 Introduction

As services that utilize statistical processing on datasets advance, the distribution and flexible use of data are becoming increasingly important. Datasets sometimes contain sensitive information about individuals. General Data Protection Regulation (GDPR) [200] limits the use of such sensitive data. However, companies sometimes violate the regulation [4, 5]. Data providers cannot directly control how their data is used and must trust counterparties to comply with contractual obligations. For this reason, anonymization is widely used to reduce re-identification risks. Anonymization masks some parts of the data or adds random noise. However, it can make advanced uses of data, such as linking across datasets, difficult. While techniques like anonymization are needed for privacy, they inevitably create barriers to data utilization. An approach is needed to enable advanced analysis of unanonymized datasets under the control of data providers without exposing sensitive information.

To process sensitive data, methods utilizing Trusted Execution Environments (TEEs) have been proposed [201, 202, 203]. With TEEs, processing can be performed

Tab. 6.1: Comparison of different privacy-preserving data analysis approaches

Feature	yak	PDMS [206, 207]	TEE-based systems [201, 202, 203]	DCR [204, 205]
Environment with attestation	✓	✓ (with TEE)	✓	✗
Privacy protection	✓	✗ (only simple access control)	✓/✗ (depends on system)	✓/✗ (can be insufficient)
Data provider's authority in analysis	✓	✓	✗	✓/✗ (can be lost)
Extensibility	✓	✓/✗ (dedicated task system)	✗	✗

without exposing sensitive data to the processor or server owner. Processing is isolated from the normal processor and memory areas, and the data in memory is always encrypted. However, while these applications ensure the confidentiality of computation, they often lack safeguards for inputs and outputs. Malicious analysis queries can leak sensitive data as a result. “Data Clean Rooms” (DCRs) [204, 205] provide similar solutions. They claim that their DCRs preserve privacy through pseudonymization or differential privacy. However, DCR vendor employees could exfiltrate sensitive data, and controls, including anonymization, may not be fully enforced. Data providers lack a means to verify or strengthen these protections. As shown in Table 6.1, TEE-based systems and DCRs face challenges in providing robust privacy protection. To overcome these limitations, it is necessary to reduce reliance on vendor trust and allow data providers to control how their data is used in analysis.

The Personal Data Management System (PDMS) [206] is a system to handle individual data while preserving privacy. Its design allows data providers to manage their data collection and retrieval. An extension of PDMS [207] provides a task system with access control on data. A task processes data on the data provider side and suppresses data leakage. However, it provides only simple access control and lacks a thorough discussion of the task system. To enable more functional and secure data analysis, it is necessary to design mechanisms for controlling data in complex analyses and a module system to support these mechanisms.

In this chapter, I propose **analysis control** and a platform called **yak**. Analysis control enables data providers to control data provisioning and analysis based on defined policies. It not only prohibits specific analysis functions but also enforces the execution of specific functions. Data providers can limit analysis queries and have

results modified or anonymized as needed while checking the input and output of functions.

yak is an analysis platform that enables more flexible data usage with analysis control. yak prohibits or allows the execution of analysis functions according to the policy provided by a data provider. It is designed to be easily extensible to implement various analysis functions and validators, which verify whether the input and output of functions meet policy requirements. For its module system, yak utilizes WebAssembly [35]. It provides isolation among modules, and platform users, including data providers, can develop them with general-purpose programming languages like Rust. To protect sensitive data from platform operators, yak runs inside an AMD SEV-SNP confidential virtual machine (CVM), a form of TEE. Data providers can verify the platform with an attestation protocol and provide datasets via a dedicated communication channel with authentication.

With a prototype implementation of yak, I evaluated performance using synthetic datasets. The evaluation showed an overhead that was negligible in practice. I also confirmed that the authorization mechanism provided by analysis control can restrict or enforce specific analytical processes.

6.2 Related Work

6.2.1 TEE-based Data Management and Analysis System

EnclaveDB [201] is a well-known database application that utilizes Intel SGX, a type of TEE. It ensures the integrity and confidentiality of data and database queries while minimizing overhead compared to existing database systems. A similar technique [202] has been proposed for Key-Value Stores. In addition to specific data processing such as MapReduce in Intel SGX environments [203], various approaches [208, 209, 210] have been proposed for running general-purpose applications within TEEs. Methods for running WebAssembly modules in TEEs include Twine [211] for SGX.

While these technologies ensure confidentiality by executing applications within TEEs, they do not sufficiently discuss data control. Therefore, although they can serve as fundamental technologies for this infrastructure, they cannot fulfill the requirements for authorizing and controlling sensitive data analysis. yak addresses these authorization and control requirements for analysis. These works could help integrate database systems into yak.

6.2.2 Personal Data Management System

PDMS [206] is a data management system designed to utilize individual data while safeguarding privacy. Extensive and Secure PDMS [207] is a secure PDMS, featuring trusted core modules for encrypted communication, policy enforcement, auditing for data tasks, and data storage.

These architectures offer access control and auditing for data processing. However, it remains unclear how the analysis process is authorized between these parties and what interfaces are used to configure modules when data providers and analysts are distinct. A distributed PDMS [212] has been proposed in which data owners maintain PDMS nodes and process queries from analysts. It consists of separate data providers and analysts, and it provides an authentication mechanism. However, it lacks fine-grained control over analysts' analytical processes. To realize more practical and flexible data provision and analysis processing, yak employs a modular system that is easily extensible and enables policy enforcement. This modular system is designed to leverage existing programming languages and general-purpose libraries. Furthermore, validator modules can provide various anonymization techniques based on data provider policies.

6.3 Overview of Data Analysis Platform

To enable control in analysis, I propose **analysis control**. In the context of data sovereignty, data providers have the right to control how their data is used. To provide stronger control over analysis, analysis control regulates the execution of analysis functions based on policies defined by the data provider. It not only restricts execution but also enforces function execution, giving data providers greater authority over data analysis. Further details are provided in Section 6.4.

Analysis control introduces a gradation in analysis. Currently, raw data must be anonymized before being shared with analysts. TEE-based analysis platforms [201, 202, 203] ensure data confidentiality during processing but remain vulnerable to data leakage through malicious queries. Analysis control directly governs the execution of analysis functions, allowing data providers to block harmful or undesirable queries. Without modifying the raw data, they can mitigate data leakage.

Fig. 6.1 shows an overview of the platform. Data providers supply their data and define policies for analysis control. The analysis platform, called **yak**, provides a controlled environment for data processing based on these policies. An analyst

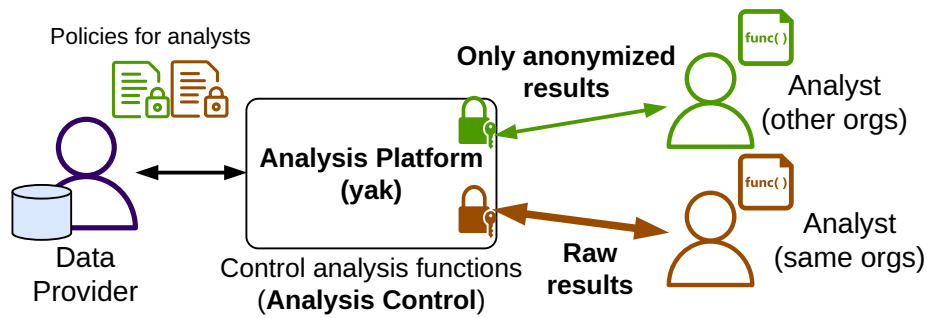


Fig. 6.1: Analysis platform with data provider’s control

submits their queries to yak, which processes the data according to the given query and policy. The data provider transmits the requested data to yak via a dedicated channel. If a query violates the policy, the analysis is aborted. yak returns the output if the analysis and results comply with the policy.

To ensure data confidentiality with respect to platform operators, yak runs on a TEE. Running yak on a TEE ensures that data providers’ datasets remain concealed from platform operators. Furthermore, all analysis functions are isolated within yak to enforce policies strictly. Section 6.5 provides further details. With these features, yak offers a secure environment for analyzing confidential data under analysis control.

6.4 Analysis Control with Policy

6.4.1 Requirements and Approaches

Methods for controlling analysis are divided into passive and active approaches: “reject” and “require”.

“Reject” refers to banning specific analysis functions. Some functions, such as retrieving the head values of a column, can expose raw data. Such functions must be prohibited when analyzing sensitive data. The “reject” approach aims to prevent data disclosure or leakage through policy in a passive manner. However, banning functions alone is not sufficient. For example, the function `mean` poses a different type of data leakage risk. The output reveals the exact value if the input dataset contains only a single item. Conversely, if the dataset contains thousands of items, it becomes difficult to infer individual values. The risk depends on the function’s input. Both the platform and policy must consider the function and its input.

“Require” enforces the execution of specific analysis functions. This approach actively verifies data and functions to prevent information disclosure. In the context of Statistical Disclosure Control (SDC) [213], data providers manage information leakage by modifying data. Specifically, rule-based SDC [214] anonymizes the output according to the defined rules. “Require” enables this in analysis control. The policy and platform must check the input and output of functions and modify the output as needed.

Finally, while data is processed across multiple functions, the policy for each data must be enforced just before the result is returned. If a function combines data governed by different policies, the output must inherit and comply with both policies.

6.4.2 Fundamental Design

The policy and analysis control system were designed based on two principles to meet the requirements.

The first principle is to provide control at the level of individual analysis functions. To achieve precise control, it is essential to intercept each analysis function and regulate its execution. The policy must handle both “reject” and “require” for each function, including its parameters. The analysis control system employs validators to support various functions, each corresponding to a specific function. Details of the validator are described in Section 6.5. The validator processes a function’s input and output, determining whether they satisfy the policy based on the given parameters.

The second principle is to define policies for each column and ensure their propagation throughout the analysis. In the analysis control framework, input data is assumed to be in columnar format. Each column consists of an array of values and a name. The columnar data is considered the smallest unit of control. The policy defines “rules” that enforce the rejection or requirement of functions on a column basis.

During analysis, columns may be transformed or processed into different data formats. The analysis control system must track the analysis process to ensure that rules defined at the column level are properly enforced. The framework propagates rules associated with a column throughout the analysis. For example, when column data is processed into a histogram, the policy is propagated to the result. It ensures strict control across various functions and reduces the risk of leakage.

6.4.3 Formulation for Policy Format and Decision Making

A policy consists of three sets: **rules**, **function policies**, and **table validators**. A rule specifies functions that are either rejected or required on a column basis. A function policy defines validator parameters for a function. A rule uses function policies to specify the controlled functions and validators. A table validator defines validators for table-formatted data.

The policy is formulated as follows: Let R , FP , and TV represent sets of rules, function policies, and table validators, respectively, while $rule$, fp , and tv denote individual elements. Each $rule$ defines the column name to which the policy is applied. A $rule$ includes multiple function policies, specifying rejected functions as fp_{reject} and required functions as $fp_{require}$. All fp_{reject} policies must not be violated, and at least one $fp_{require}$ policy must be satisfied when returning the result. Each rule is formulated as follows:

$$rule := (column_name, \{fp_{reject} \in FP\}, \{fp_{require} \in FP\}, tv \in TV \text{ (} tv \text{ can be empty)})$$

Each function policy fp defines the function name to control and the arguments used to determine whether the function should be rejected (arg_{reject}) or satisfies the require policy ($arg_{require}$). The platform validates the input and output of a function using a corresponding validator and these arguments.

$$fp := (function_name, arg_{reject}, arg_{require})$$

A table validator is used when the result to be returned is in table format. Function policies are intended to be used for specific functions. Table validators aim to enforce more generic anonymization for table data.

$$tv := (validator_name, args)$$

The analysis control system must enforce policies throughout the entire analysis process. However, data may be transformed into different formats. To track such changes and ensure policies are enforced just before the final result is returned, this analysis control implements **policy propagation**.

The policy propagation is formulated as follows. First, the corresponding rules are embedded into single-column data $Data$. This embedding occurs at the time of data provisioning. The embedded rules in $Data$ are denoted as ER :

$$ER(Data) = \{rule \in R \mid column_name(Data) = column_name(rule)\}$$

A function $funcA$ takes $Data_{in}$ as input and produces $Data_{out}$ as output. If the input data originates from multiple sources, they are concatenated into a table. The embedded rules in $Data_{in}$ are then propagated to $Data_{out}$:

$$ER(Data_{in}) = \bigcup_{k \in I} ER(Data_{in}^k)$$

$$ER(Data_{out}) = ER(funcA(Data_{in}))$$

Algorithm 2, Algorithm 3, Algorithm 4, and Algorithm 5 form the analysis control. These verify whether the policy requirements are met.

Algorithm 2 Procedure to check “reject” function policies

Require: $Data_{in}$, $Data_{out}$, $funcA$

```

1 for all  $rule \in ER(Data_{in})$  do
2   for all  $fp \in Rejects(rule)$  do
3     if  $function\_name(fp) = funcA$  then
4       if  $Validate(fp, Data_{in}, Data_{out})$  returns error then return Abort due to re-
        jected function execution.
5     end if
6   end if
7 end for
8 end for return Continue analysis

```

Algorithm 3 Procedure to check “require” function policies

Require: $Data_{in}$, $Data_{out}$, $funcA$

```

1 for all  $rule \in ER(Data_{in})$  do
2   for all  $fp \in Requires(rule)$  do
3     if  $function\_name(fp) = funcA$  then
4        $validated = Validate(fp, Data_{in}, Data_{out})$ 
5       if  $validated$  returns error then
6         Ignore
7       else
8          $mark\_require\_satisfied(Data_{out}, rule)$ 
9         if  $validated$  contains data then
10           $Data_{out} = validated$ 
11        end if
12      end if
13    end if
14  end for
15 end for return Continue analysis

```

For “reject” policies, Algorithm 2 presents the algorithm used to verify policy compliance when a function $funcA$ is executed. It ensures that the pair $(Data_{in}, Data_{out})$ does not violate any of the rules embedded in $Data_{in}$. If at least one rule is violated, the procedure aborts the analysis.

Algorithm 4 Procedure to check table validators

Require: $Data_{out}$ (table-formatted)

```
1 for all  $rule \in ER(Data_{out})$  do
2   if  $rule$  has  $tv$  then
3      $validated = \text{ValidateTV}(tv, Data_{out})$ 
4     if  $validated$  returns error then return Abort analysis
5   else
6      $\text{mark\_require\_satisfied}(Data_{out}, rule)$ 
7     if  $validated$  contains data then
8        $Data_{out} = validated$ 
9     end if
10  end if
11 end if
12 end for return Continue analysis
```

Algorithm 3 presents the algorithm for verifying “require” rules. All rules embedded in $Data_{in}$ must be satisfied before the result is returned to the analyst. Furthermore, validators for “require” can modify $Data_{out}$. If modified, $Data_{out}$ is overwritten with a validator’s output.

Algorithm 4 shows the procedure to check table results with table validators. If the result has rules that contain table validators, check them and return a modified result as needed.

Algorithm 5 Procedure to check all rules are satisfied

Require: $Data_{out}$

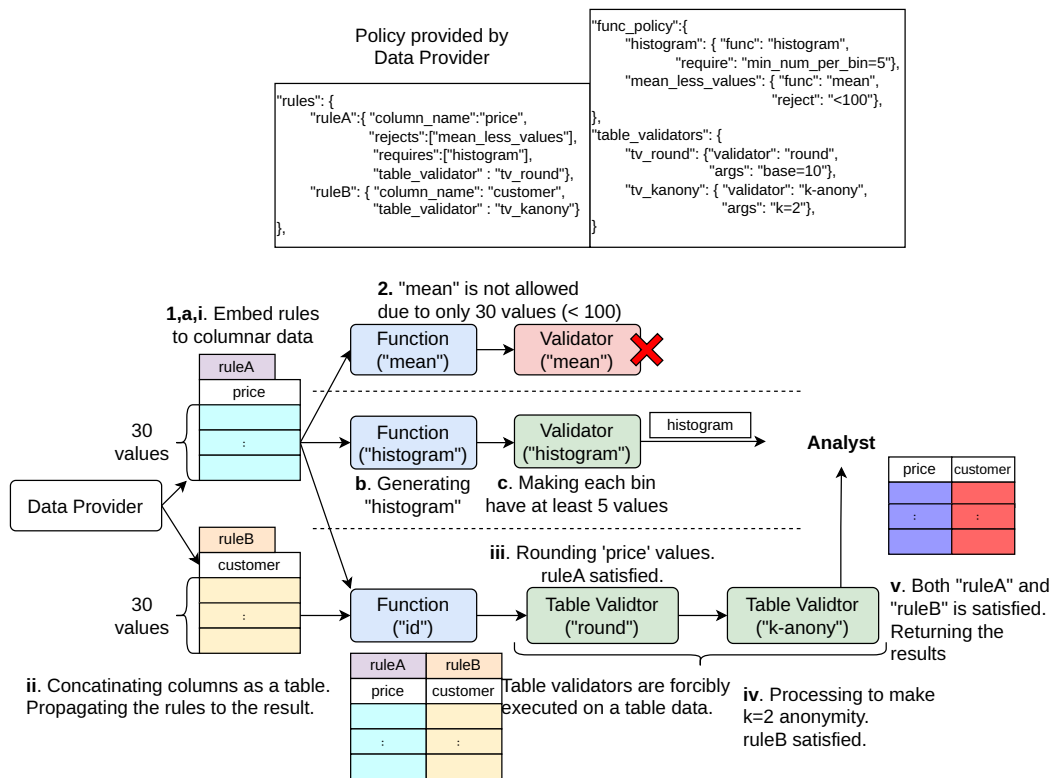
```
1 for all  $rule \in ER(Data_{out})$  do
2   if  $rule$  has  $fp_{require}$  or  $tv$  and not marked satisfied then return Abort due to required functions not being executed.
3   end if
4 end for return  $Data_{out}$  is returned to the analyst.
```

After the validators’ checks, Algorithm 5 is executed. It checks that all rules are satisfied. If satisfied, the system returns the result to the analyst. Otherwise, the analysis is aborted.

6.4.4 Flow of Analysis Control

Fig. 6.2 shows an example of analysis control based on a data provider’s policy. The columns price and customer are retrieved from a data provider and have embedded rules.

When mean is executed, ruleA defines the reject function policy mean_less_values. It prohibits the execution of mean when the input contains fewer than 100 values.



This violates the policy since price has only 30 values. The validator for mean returns an error, and the analysis is aborted.

When `histogram` is executed on `price`, the function policy `histogram` is enforced by `ruleA`. The policy includes a parameter `min_num_per_bin=5` specifying a requirement. The validator for `histogram` checks the output of `histogram`. If the minimum number of values per bin is below 5, the validator merges bins to satisfy the condition defined by the parameter. This modification depends on the implementation of the validator. This analysis control system enables more flexible anonymization by incorporating passive and active control mechanisms.

Table validators are strictly enforced for table data. `id` concatenates `price` and `customer` into a table. The table inherits rules `ruleA` and `ruleB`. `ruleA` specifies `tv_round` as a table validator. `tv_round` invokes the table validator `round`, which rounds the values in `price`. Executing the table validator `round` satisfies the requirements of `ruleA`. Following this, the table validator `tv_kanony` specified by `ruleB` is executed. It ensures k -anonymity with $k = 2$ within the table. These modifications satisfy both `ruleA` and `ruleB`. Finally, the result is provided to the analyst.

6.5 yak: WebAssembly-based Analysis Platform

6.5.1 Towards Implementing Analysis Control

The analysis platform must possess the following functionalities to implement analysis control, as outlined in Section 6.4.

First, each analysis function must be distinctly isolated, allowing the platform to control function invocation. Breaking isolation leads to incomplete control and potential data leakage. To ensure effective policy enforcement, functions must not share data except through a dedicated API designed to enforce these policies. This API must also handle diverse data formats, ranging from tabular data to statistical metrics.

Second, the platform must execute validators according to the defined policies. Validators are required to process both the input and output of modules.

Third, data providers furnish data and policies tailored to each analyst. Each data provider maintains its remote data storage. The data provider module establishes connections to the storage and supplies data through the platform's interface. Furthermore, the relevant policy must be provided when the analyst initiates an analysis session, as policies vary across analysts. Therefore, the module must dynamically provide the appropriate policy at runtime.

Fourth, the platform should give analysts an easy way to define analysis procedures. This implies that analysts should be able to conduct analyses in the same way in conventional environments, such as Python.

Finally, the platform must ensure the confidentiality of data. Unauthorized individuals, like operators, could steal the data. This would make the platform ineffective. Therefore, the platform must operate in a confidential environment such as a TEE. Additionally, the platform must provide verifiable proof that the environment is secure and operates as intended.

6.5.2 Design of yak

Fig. 6.3 shows the overview of yak. yak is a data analysis platform that satisfies the requirements outlined in Section 6.5.1. Analysts define their analysis procedures as a Directed Acyclic Graph (DAG). The runner controls the analysis based on the DAG and enforces the policy provided by the data provider. All functionalities, including

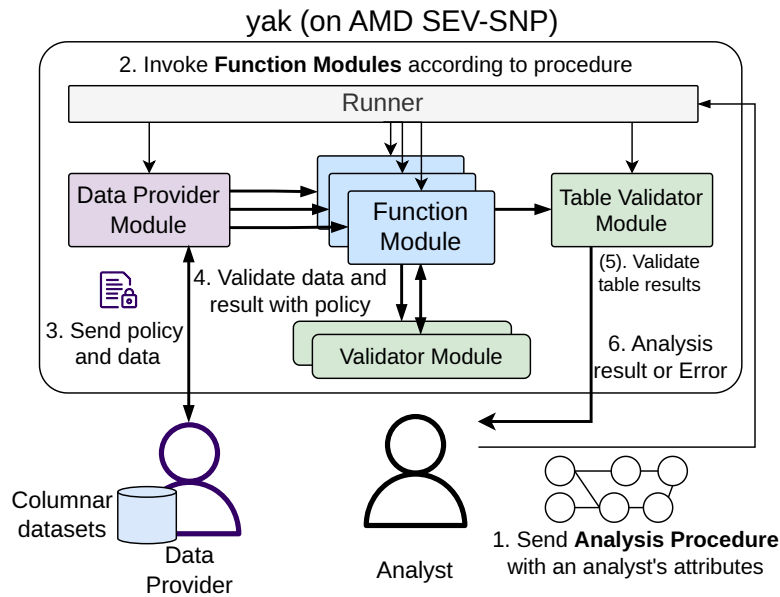


Fig. 6.3: Overview of yak

data provision and analysis functions, are implemented as separate modules. Each module is isolated using WebAssembly [35] and provides dedicated APIs for data processing. The runner chains data from a function module to a validation module and subsequent function modules. To ensure data confidentiality, yak operates inside a Confidential Virtual Machine (CVM), a form of TEE. A CVM provides data confidentiality and integrity for the guest virtual machine. However, simply running yak inside a CVM is insufficient. Data providers also need a way to verify that the running yak instance and the underlying CVM are genuine and that data confidentiality is assured. This verification method is known as “attestation”. yak provides an attestation interface for data provider modules to verify the platform’s authenticity and confidentiality before data provision. Data providers verify yak through a data provider module and then supply policies and requested data.

6.5.3 WebAssembly-based Module System

yak’s module system is designed to be both extensible and secure. This design allows developers to implement a wide range of analysis functions, from simple to complex. To support extensibility while enforcing analysis control strictly, each function must be isolated. To meet these requirements, yak uses WebAssembly to provide isolation among modules. WebAssembly offers lightweight isolation among modules and operates across various platforms. Linux processes provide similar isolation. However, binaries for these processes have dependencies on an

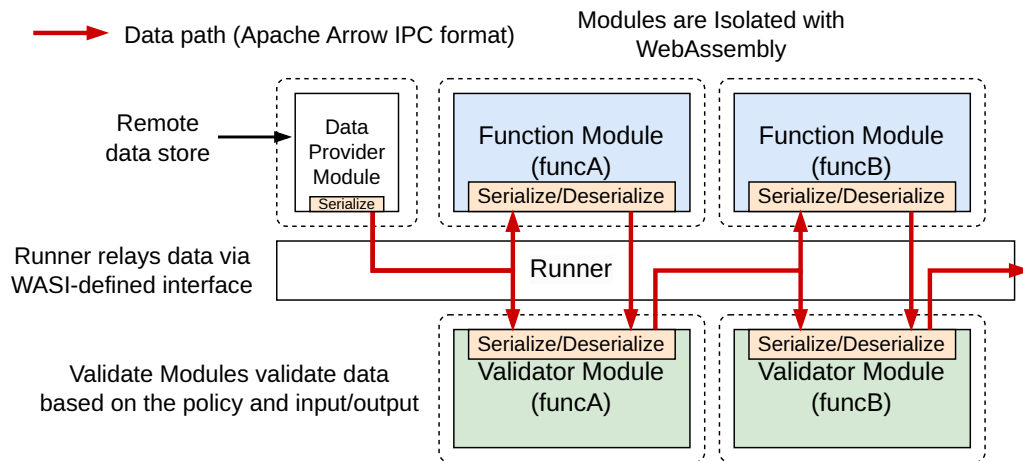


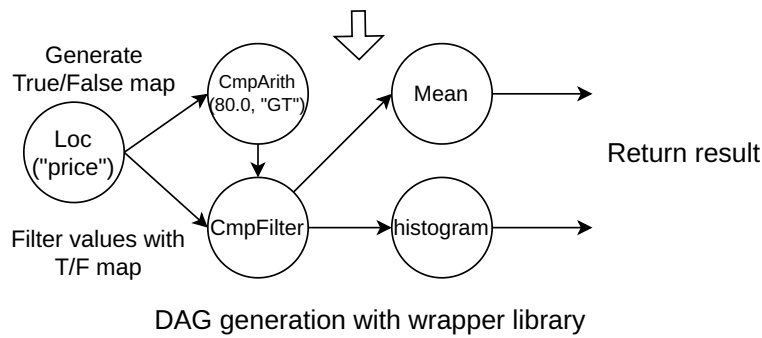
Fig. 6.4: Data chaining among modules

environment, such as architecture and libraries. Container technology addresses this issue, but its footprint tends to be large, which becomes problematic when running tens or hundreds of modules. WebAssembly modules require only a runtime. Modules do not depend on external libraries, so there is no need to consider the underlying architecture. This module system is extensible: developers can implement new analysis functions and validators as WebAssembly modules without modifying the platform.

To support various data formats, including columns, tables, and statistics such as histograms, yak uses the Apache Arrow IPC format [215]. It is a binary data format that efficiently serializes columnar data. The format supports columnar data, matrices, and other statistical data through user-defined extensions. To provide extensibility with modules, the runner does not interpret the data format but chains binary data among modules. As shown in Fig. 6.4, each module deserializes binary data and processes it. The processed result is serialized into IPC format, and then the runner relays it to the next validation and function modules. Module inputs can originate from multiple sources. To handle this, modules process multiple binary data inputs. The process depends on the implementation. For example, a function `id`, which passes input data through unmodified as output, concatenates input columns into a table. A function `mean` calculates each column's mean and returns it as a table of mean values. This design allows each module to flexibly handle different data representations.

The runner enforces policies during analysis. Policies and analysis control are implemented as described in Section 6.4. After executing a function module, the runner checks data according to the algorithms. Validate in them corresponds to

Query: SELECT mean(price), histogram(price) WHERE price > 80.0



DAG generation with wrapper library

```
price = dag.Loc("price")
ps = dag.CmpFilter([dag.CmpArith(price, 80.0, "GT"), price])
(d, _) = dag.DAGBuilder().add(dag.Mean(ps)).add(dag.Histogram(ps)) \
    .build()
```

Fig. 6.5: Analysis procedure is defined with DAG

calling a validator module. The runner calls the validator module and decides whether to continue or abort the analysis.

6.5.4 Defining Analysis Procedure with DAG

Analysts define their procedures in a Directed Acyclic Graph (DAG) format in yak. A clear representation of analysis procedures is crucial for analysts. Some analysts use programming languages like Python. They can describe their procedures by writing code. However, directly using procedural programming languages can lead to information leakage and reduced flexibility. A typical example is intermediate results. Analysts store intermediate results as local variables in code. While other code might reuse these results, it necessitates yak returning them to the analyst. This implies that intermediate results must comply with policies, which can be challenging regarding policy complexity. Analyzing code and variable dependencies might eliminate the need to return intermediate values. DAGs provide a dependency representation by explicitly defining function and variable dependencies. This eliminates the cost of analyzing dependencies, and the runner can know explicitly which results should be returned.

In a DAG representing an analysis procedure, nodes and edges represent analysis functions and data dependencies. Fig. 6.5 shows the example. A query filters price values greater than 80.0 and calculates the mean and histogram. Manually writing a DAG definition can be difficult. yak provides a lightweight wrapper library for Python. Analysts can define analysis procedures procedurally. Additionally, yak

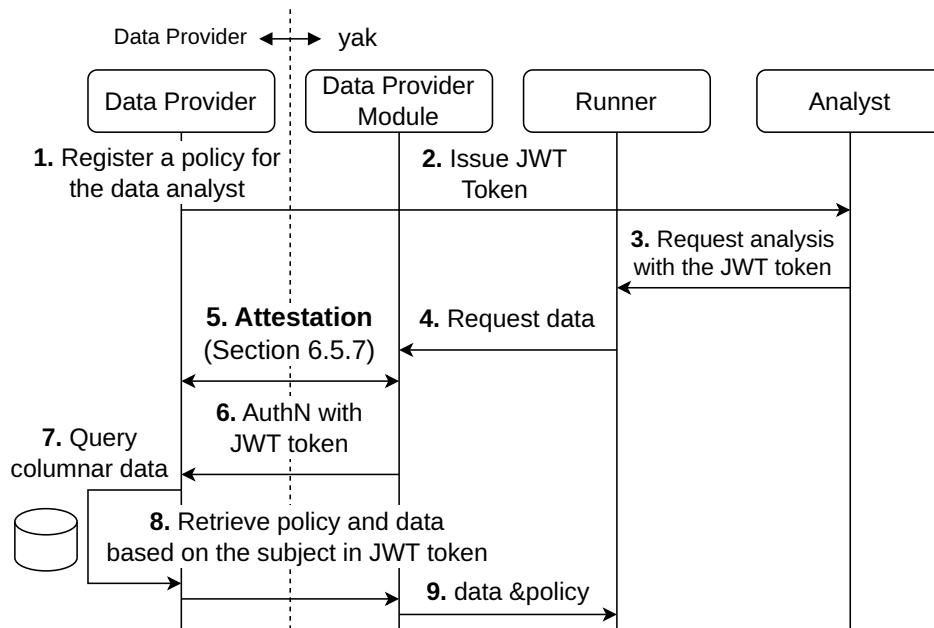


Fig. 6.6: Handshake and authentication between an analyst and data provider

supports pre-defined DAGs as **SubDAGs**. Analysts can obtain results of complex analyses like Ordinary Least Squares with this. They can invoke SubDAGs in the same way as function modules without a procedure definition.

6.5.5 Providing Policies and Data From Remote with AuthN

yak is designed to handle datasets provided by remote data providers. Data providers can provide columnar datasets stored in their data store, like RDBMS servers, via a data provider module. This module fulfills two roles: providing policies based on analysts and delivering data with authentication.

Firstly, data providers define multiple policies based on analyst attributes, such as affiliation. A policy allowing most analysis functions is provided for analysts from the same company or organization. Data providers impose policies that permit only restricted functions for analysts from other organizations. yak facilitates this gradation of analysis control through the proper application of multiple policies.

To enable authentication and policy provision, yak employs JWT [216] token-based authentication for data provider modules. Fig. 6.6 shows the handshake. Data providers issue JWT tokens to analysts. When an analyst initiates an analysis, the runner initializes the data provider module. The module then establishes a handshake connection with the data provider via a WASI socket. The connection

between the data provider module and the data provider is encrypted using TLS. During the handshake, the token is transmitted to the provider, and the provider verifies it. If the verification is successful, the provider supplies a policy based on the subject information embedded in the token. Simultaneously, the requested column data is also provided. Policy management is provider-specific, and embedding analyst attributes in the token enables flexible policy selection. The runner enforces the selected policy throughout the analysis.

6.5.6 Volatile Platform without Persistent Storage

yak is designed to operate without persistent storage. Persistent storage enables data to be retained after analysis, creating vulnerabilities for data leakage. A volatile platform is essential for controlling the data lifecycle.

yak operates within AMD SEV-SNP [217]. AMD SEV-SNP is a kind of CVM that provides confidentiality and integrity for a virtual machine's memory and CPU register state. It supports the execution of the Linux kernel, and yak runs on Linux within it. In yak, all software and libraries are bundled as a RAM disk. The CVM has no persistent storage, and all data retained in memory is flushed upon CVM termination. It ensures that data is reliably removed after analysis and cannot be reused for subsequent analyses.

6.5.7 Remote Attestation Protocol for Data Providers

Data providers care about the confidentiality of their data. yak runs on an AMD SEV-SNP's CVM, which ensures data confidentiality and integrity. However, yak must prove that the running yak and CVM are genuine and that analysis control is correctly enforced. To verify it, yak provides a remote attestation protocol for data providers.

Fig. 6.7 shows the attestation protocol for data providers. AMD SEV-SNP can generate an attestation report that includes a 64-byte report-data field [218]. yak's attestation protocol utilizes this field and implements challenge-response-based attestation [219]. First, a data provider module calls a WASI function for handshake, and a runner and a data provider's remote server establish a secure end-to-end connection. The runner requests a nonce n from the data provider's server, and the server returns n . The runner generates an attestation report with the nonce and sends the report to the provider's server. This protocol relies on a secure

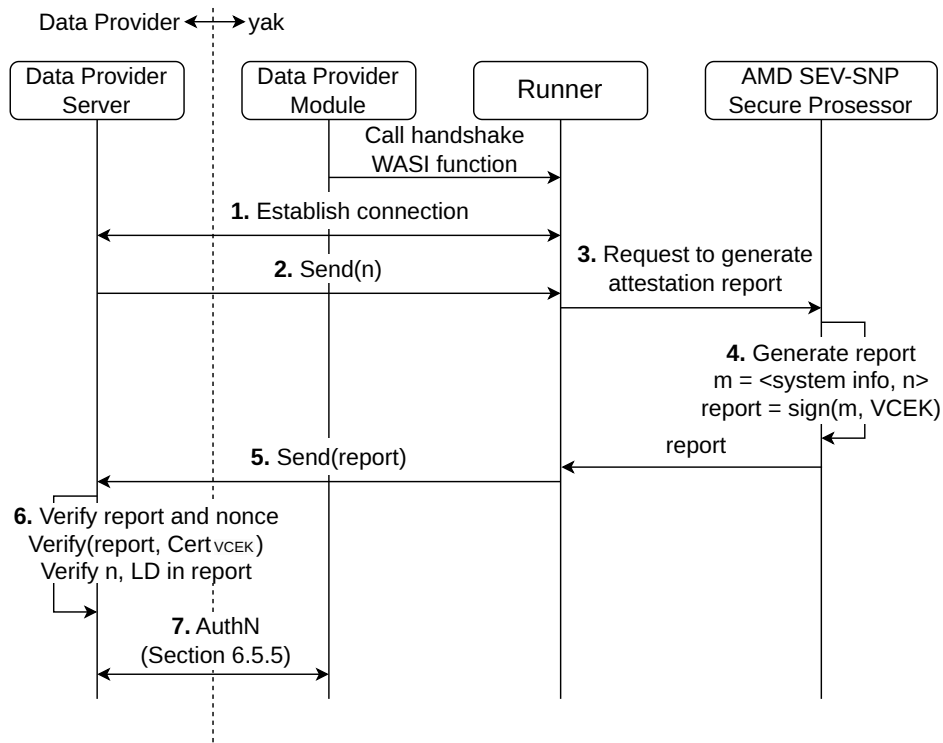


Fig. 6.7: Remote attestation protocol for data providers

communication channel between the runner and the data provider's server. When employing TLS to secure this channel, the server must prevent potential Man-in-the-Middle (MITM) attacks. Channel binding with exported TLS keying material [220] can prevent such attacks. Specifically, $Hash(n \parallel Key_{exported})$ is used as the report-data instead of the nonce, where $Key_{exported}$ denotes the exported keying material. The report is signed with the SEV-SNP Secure Processor's Versioned Chip Endorsement Key (VCEK) [221]. The secure processor ensures that the currently running processor is genuine. Then, the server verifies the report. AMD provides certificates to verify the VCEK-signed attestation report. A data provider verifies the certificates and confirms that the report is signed within an SEV-SNP environment. If the report verification is successful, the server proceeds with the authentication described in Section 6.5.5. If the verification fails, the server aborts the connection.

SEV-SNP provides a Launch Digest (LD) in an attestation report [218]. LD is a digest of the memory content at CVM startup. Verifiers can check whether the launched kernel and the RAM disk are as expected. The LD for yak is calculated in advance, enabling users to verify whether the running yak is genuine by comparing the LD.

6.6 Threat model

yak aims to prevent the unexpected theft of data from data providers. If an analyst has malicious intent, analysis control, as described in Section 6.4, prevents unauthorized parties from accessing and analyzing the data. If the processing infrastructure or computing environment has been maliciously modified, the attestation protocol detailed in Section 6.5.7 ensures the integrity of the computing environment and underlying software. The remaining threats include the security of analysis functions, the yak implementation, and the potential leakage of the analysis procedure to the data provider. Regarding the security of analysis functions, the provided functions are assumed to be correctly implemented. For the security of the underlying software, it is assumed that no vulnerabilities exist and policies are correctly enforced. As for the leakage of the analysis procedure, it is protected within the CVM, but the types of retrieved data may be leaked to the data provider.

6.7 Evaluation

I evaluated the performance of the analysis using yak and native Python execution. The evaluation environment was a CVM with 4 vCPUs (AMD EPYC 7313P) and 8 GiB of memory.

6.7.1 Performance Evaluation with Synthetic Data

First, I evaluated the basic performance using synthetic data. This evaluation measured performance with varying data volumes and function counts. In this evaluation, the data provider did not use remote storage, providing data from module-embedded storage to align conditions with Python. Each evaluation represents the average of 100 measurements. For functions, `var` and `LinearRegression` (LR) were used. To evaluate the overhead caused by the module system, `id` was also measured. `id` concatenates inputs and outputs without modifying the values.

Fig. 6.8, Fig. 6.9, and Fig. 6.10 illustrate the performance of basic analysis functions with varying data volumes. Up to 10,000 values, the overhead in yak and Python remained comparable. This suggests that function call costs were dominant. As the number of values increased, execution time increased approximately linearly in both yak and Python. This is attributed to the cost of processing values becoming dominant, closely aligning with the number of values. yak exhibited up to seven times

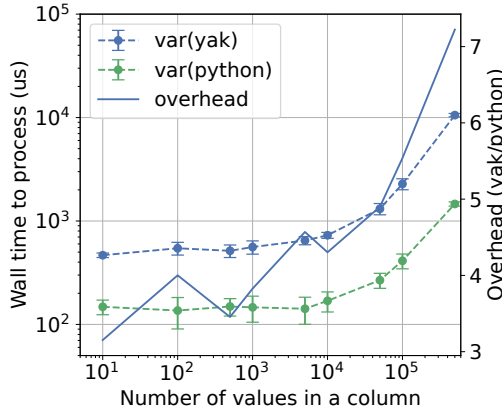


Fig. 6.8: Function var performance

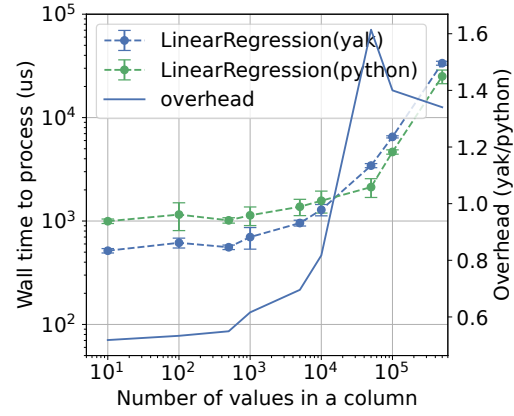


Fig. 6.9: Function LinearRegression performance

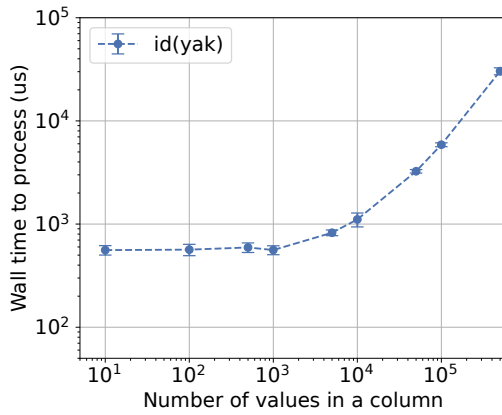


Fig. 6.10: Function id performance

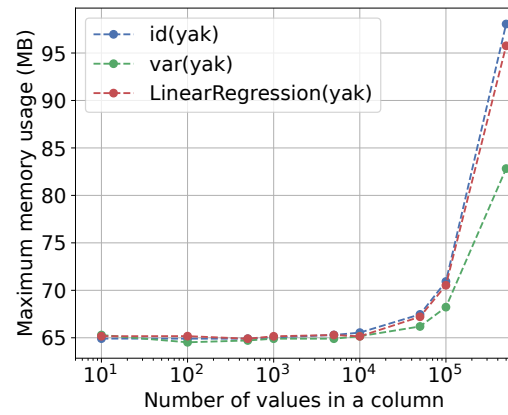


Fig. 6.11: Memory usage on the number of values

the overhead of Python in the worst case. Notably, var had substantial overhead, which worsened with increasing data volumes. Python's var execution time did not increase much. Considering the increased wall time in id, deserialization caused overhead in yak that grew with dataset size. LR exhibited different behavior. With smaller data volumes, yak's LR was faster than Python's. yak's LR was implemented using `linfa_linear`, which provides efficient linear algebraic operations, resulting in faster linear regression calculations. However, as data volumes increased, yak became slower than Python. This was likely due to the same data deserialization overhead. Fig. 6.11 shows the memory usage for the number of values. id with 1 value and 500,000 values consumed about 65 MB and 98 MB in the entire system, respectively. Values were represented as 32-bit floating-point numbers; thus, 500,000 values occupy about 2 MB. The system consumed roughly 33 MB to retain these values—about 15 times inflation.

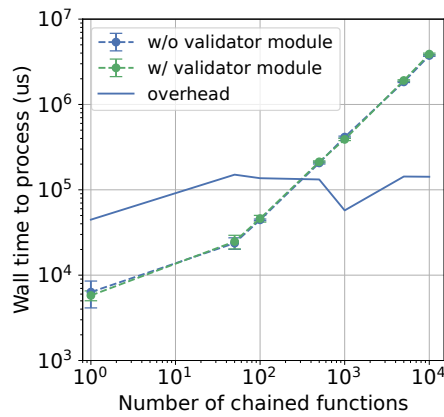


Fig. 6.12: Performance on the length of functions

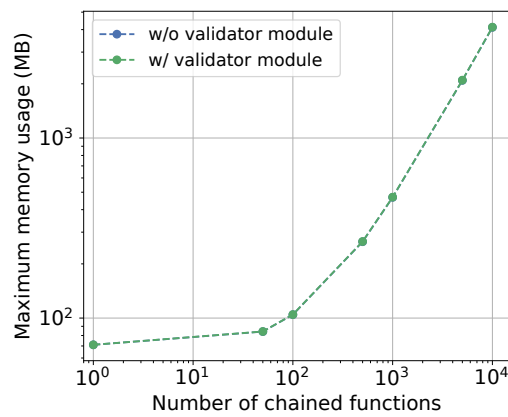


Fig. 6.13: Memory usage on the length of functions

yak chains function modules to process analysis procedures. The execution is serial, leading to increased processing time with longer function chains. I evaluated the relationship between chain length and analysis wall time. To measure function chaining performance, `id` was used as the function. It deserializes inputs and serializes them as an output. A column with 100,000 values was used as input. Fig. 6.12 presents the results. The processing time increased with the length of the function chain. The increase showed an almost linear relationship between time and the number of functions. This indicates that each function incurred serialization and deserialization overhead, affecting overall performance. However, the actual time with 10,000 functions did not reach 10 seconds, so the serialization or deserialization overhead is considered negligible. Fig. 6.13 shows the memory usage for different chain lengths. The usage increased almost linearly with chain length. At 10,000 functions with validator modules, the entire system consumed about 4.1 GB, meaning one function module consumed about 412 KB. Fig. 6.11 and Fig. 6.13 show that the number of function modules is the primary driver of memory usage. This result indicates that large procedures can require substantial memory resources.

Performance was measured with and without policy enforcement to evaluate the overhead caused by validator modules. The policy was defined to let the runner always call the corresponding validator module after the function module execution. To measure the validator module call overhead, `id` was used as the function. The validator module for `id` deserializes both input and output and verifies that column names remain unchanged. Fig. 6.12 presents the results. There was no significant overhead with validator modules. The validator modules only performed deserialization, which did not substantially affect performance compared to serialization

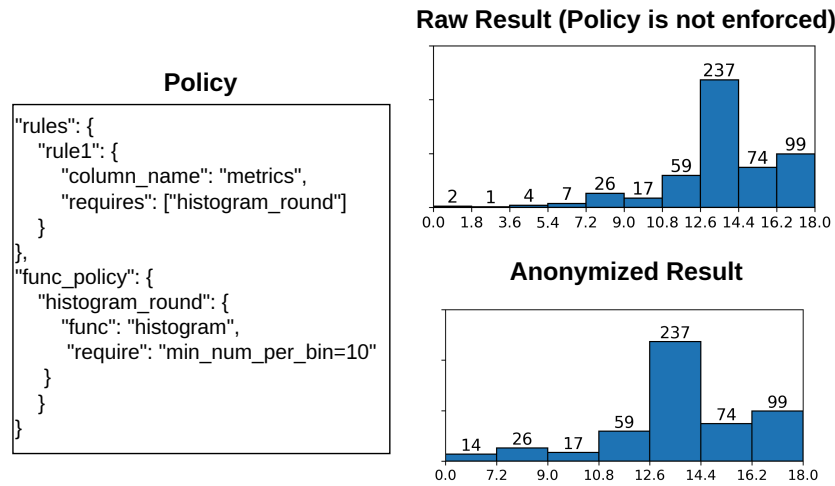


Fig. 6.14: Anonymization by policy: Histogram

in a function module. This benchmark indicates that calling validator modules themselves does not cause significant overhead. In real-world applications, validator modules may have suboptimal implementations. The development of efficient validator modules is outside the scope of this research.

These benchmarks indicated that yak incurred overhead compared to native Python execution. This overhead was primarily observed during serialization/deserialization and module chaining. It stemmed from yak's WebAssembly-based module design, which ensures data confidentiality and control over analysis; native Python execution does not provide these features. I consider this overhead an acceptable trade-off for these functionalities in secure data analysis.

6.7.2 Use Case: Anonymization based on Policy

Fig. 6.14 and Fig. 6.15 show use cases of policy-based anonymization. yak can enforce the anonymization of specific data by defining policies. I implemented validator modules to anonymize histogram and table data.

The SDC handbook [213] provides a method to anonymize histograms by merging bins to ensure each bin contains values exceeding a specified lower limit. I implemented this approach as a histogram validator module. Fig. 6.14 demonstrates a policy to anonymize a histogram, ensuring each bin contains at least 10 values. A raw result contains bins with fewer than 10 values. With the policy, yak automatically anonymizes the histogram using the implemented validator module, which provides an anonymized histogram.

Policy

```
"rules": {
  "rule1": {
    "column_name": "country",
    "table_validator": "tbl_k_anony"
  }
},
"table_validators": {
  "tbl_k_anony": {
    "validator": "tbl-k-anony",
    "arg": "{
      'k': 2,
      'mask': true,
      'quasi_columns': ['country', 'age']
    }"
  }
}
```

Raw Result (Policy is not enforced)

	country	age	purchased	payment
0	JPN	22.0	item1	100.0
1	USA	11.0	item1	100.0
2	UK	21.0	item2	150.0
3	JPN	24.0	item3	200.0
4	JPN	22.0	item4	50.0

Anonymized Result

	country	age	purchased	payment
0	JPN	22-24	item1	100.0
1	*	11-21	item1	100.0
2	*	11-21	item2	150.0
3	JPN	22-24	item3	200.0
4	JPN	22-24	item4	50.0

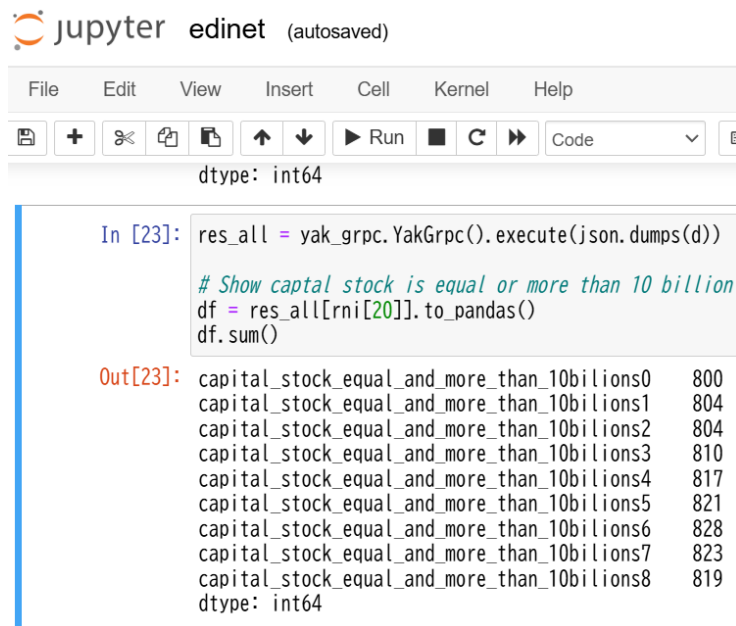
Fig. 6.15: Anonymization by policy: Table

k -anonymization is an anonymization method for tables. I implemented the Mondrian algorithm [222] for k -anonymization. This algorithm ensures that each record in a dataset is indistinguishable from at least $k-1$ other records. In a policy, the country and age columns were specified as quasi-identifiers, and k -anonymity ($k=2$) was required. The raw result displays all raw values. With the policy applied, country and age columns were masked to satisfy k -anonymity ($k=2$).

Fig. 6.14 and Fig. 6.15 illustrate a use case of yak's analysis control. Without implementing algorithms directly, data providers can anonymize their data by defining policies.

6.7.3 Case Study: Analysis of Financial Products

I performed analyses using EDINET [223] to evaluate usability in real-world scenarios. Using EDINET data, Japanese securities and stocks can be analyzed. I created a dedicated SQLite3 database from the datasets. It contains over 40 million entries, and the database's size is about 28 GiB. The database ran as a remote server, and yak requested datasets via a gRPC channel as described in Section 6.5.5. The datasets were provided remotely in response to the queries. The analysis included filtering data, calculating ratios with arithmetic operations, counting values, and displaying results as a concatenated table. Fig. 6.16 illustrates a portion of the analysis. yak provides a library for using yak via Jupyter Notebooks. The library and yak communicate through gRPC. With this library, analysts can write their procedures in the notebook and execute them with yak seamlessly. Although the procedure could be defined, there remains room for improvement. As the procedure became



The image shows a Jupyter Notebook interface with the title 'edinet (autosaved)'. The menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. Below the menu is a toolbar with icons for saving, adding cells, undo, redo, and running code. The code cell contains the following Python code:

```
dtype: int64

In [23]: res_all = yak_grpc.YakGrpc().execute(json.dumps(d))
         # Show capital stock is equal or more than 10 billion
         df = res_all[rni[20]].to_pandas()
         df.sum()
```

The output of the code cell is a table with two columns: the index and the sum of values. The data is as follows:

Index	Sum
capital_stock_equal_and_more_than_10bilions0	800
capital_stock_equal_and_more_than_10bilions1	804
capital_stock_equal_and_more_than_10bilions2	804
capital_stock_equal_and_more_than_10bilions3	810
capital_stock_equal_and_more_than_10bilions4	817
capital_stock_equal_and_more_than_10bilions5	821
capital_stock_equal_and_more_than_10bilions6	828
capital_stock_equal_and_more_than_10bilions7	823
capital_stock_equal_and_more_than_10bilions8	819

The output ends with 'dtype: int64'.

Fig. 6.16: Analysis of financial products

more complex, its readability diminished. Therefore, a more human-readable way to define procedures is required.

6.8 Discussion

6.8.1 Validity and Risk of Analysis Control

yak provides analysis control through data provider policies. Data characteristics depend on the data itself, and data providers can determine which data requires sealing or anonymization. External parties, including analysis platforms, analysts, and anonymization algorithms, lack this inherent knowledge. Consequently, there is no universal solution for protecting diverse data and providing analysis results without considering specific data attributes. yak offers general anonymization algorithms, such as k-anonymization for tables, but data providers remain responsible for selecting appropriate algorithms for data protection. Responsibility for data leakage rests with the data provider's policy, and data providers must define these policies meticulously.

However, data providers may make mistakes in policy configuration. As the last line of protection, analysis control has a “reject” policy and table validators. These prevent unintended function invocation and enable minimal anonymization so as

not to reveal raw values. Data providers should utilize them. Furthermore, a policy validator may help define better policies. The risk of data leakage depends on each anonymization algorithm, and showing an estimation of the risk would be useful. These enhancements are beyond the scope of this research and are a future task to leverage analysis control.

6.8.2 Usability for Data Providers

Data providers must define policies tailored to their data. As mentioned in Section 6.8.1, they must understand which data requires protection. This understanding is the initial step in policy definition, but it can be challenging for those unfamiliar with data anonymization. One solution is to utilize pre-defined policies as profiles. Certain organizations, particularly governments, define data formats for data collection. Experts can design policies that incorporate anonymization to comply with regulations using these formats.

Another approach involves algorithms that predict information leakage, as seen in differential privacy. These algorithms can estimate the “probability” of uniquely identifying an individual. Providers can use this metric to adjust the noise added to data for scrambling. If data providers can specify requirements for the probability, it will be possible to generate policies that meet these requirements using various algorithms. With yak and such policy generation methods, data providers can use differential privacy algorithms easily.

Data providers may provide less-structured datasets such as JSON. Analysis control assumes columnar data and cannot handle such datasets. Transforming such datasets into a pseudo-columnar representation within a data provider module would enable yak to handle a wider range of formats.

6.8.3 Analysis Performance

yak uses WebAssembly modules for a module system and the Apache Arrow IPC format for data exchange between modules. While these are necessary for module separation and a unified API, they introduce performance overheads compared to Python. Performance evaluations in Section 6.7 have demonstrated these overheads. They also indicated the possibility of significant memory consumption. Furthermore, the current yak implementation runs only one module concurrently. This is merely an implementation limitation. yak can be modified to run multiple modules

simultaneously while resolving dependencies, and this could improve performance. Performance optimization, including memory usage, is a future task.

6.9 Conclusion

In this chapter, I proposed **Analysis Control** and **yak** as its implementation. Analysis control prohibits or forces the execution of analysis functions, thereby controlling data provisioning and analysis. Data providers can control analysis with a policy and have results anonymized. yak is an extensible analysis platform that provides various analysis functions, enforces policies, and ensures data confidentiality and integrity. Evaluations using synthetic datasets demonstrated yak's feasibility with negligible overhead and confirmed its ability to enforce policies effectively.

Conclusion

Edge Computing (EC) is often delivered via platforms provided by cloud service providers. Cloud-based EC platforms are tightly coupled to the cloud and rely on centralized infrastructure for management, orchestration, and data processing. This architecture reduces the operational burden on developers and allows them to focus on application development. However, this architecture introduces availability risks due to its dependence on centralized services and raises privacy concerns.

In this dissertation, I advance Region-based Edge Computing (Region-based EC), which enables edge nodes and devices within a local region to operate independently of centralized platforms. With Region-based EC, edge infrastructure can continue to operate even when cloud services are unavailable, improving overall availability and resilience. Moreover, by processing data locally within the region, users can retain greater control over their data, enhancing privacy and reducing exposure to external services. Region-based EC empowers users with greater autonomy over their computing resources and data governance. However, it faces several challenges that must be addressed for practical adoption. The challenges are: securing intra-region communication in the presence of potentially untrusted devices; deploying and maintaining applications efficiently on resource- and bandwidth-constrained nodes; and enabling privacy-preserving collaboration with external computation and analytics.

This dissertation addresses these challenges with four contributions. CREBAS introduces a capability-based authorization architecture that provides fine-grained, user-manageable network access control among heterogeneous devices, mitigating risks posed by compromised or backdoored IoT components without requiring device-side retrofitting or expert-derived policy languages. `bypass4netns` improves the network performance of rootless containers through a socket-switching mechanism that reduces overhead in privilege-separated networking paths, enabling secure containerization without network performance penalties. D4C delivers efficient container image updates with merge-based delta generation and a filesystem that applies deltas lazily, significantly reducing data transfer volume and update latency under limited network bandwidth. `yak` provides analysis control inside Trusted Execution Environments, allowing data providers to specify prohibited operations

and enforce anonymization requirements directly on analytical queries, preventing exfiltration through malicious or data-leaking requests. These contributions enhance the practicality of Region-based EC by addressing key challenges in security, application deployment, and data privacy.

Future work on Region-based EC should focus on middleware and applications that can replace existing cloud-dependent services while leveraging the benefits of local autonomy. Application developers have benefited from managed, cloud-based EC platforms, but in Region-based EC, they must handle infrastructure management and orchestration themselves. Some orchestration tools for the edge, such as KubeEdge [61] and OpenYurt [224], can help manage edge nodes. Developing orchestration and management frameworks tailored for Region-based EC, including integration with services such as Function-as-a-Service, can lower the barrier for application developers and operators. For end users, enabling easy setup and management of edge nodes and devices will be important to promote the adoption of Region-based EC. Current cloud-based EC platforms and their applications provide web consoles and smartphone apps to simplify the management and monitoring of edge nodes and devices. In Region-based EC, similar user-friendly management tools will be necessary to facilitate broader adoption among non-expert users. Home Assistant [225] already provides a user-friendly interface for locally managing smart home devices, and it will play an important role in Region-based EC. With the foundational infrastructure for secure and efficient Region-based EC established, the next frontier is the support for high-level applications. Platform providers and vendors are actively developing cloud-based AI services that leverage powerful models hosted in the cloud. Bringing such cloud-grade AI capabilities to the edge while preserving local autonomy is crucial. Although computational resources are typically limited at the edge, the ongoing trend toward efficient hardware accelerators for resource-constrained environments (e.g., self-driving vehicles) will facilitate the deployment of high-level services in Region-based EC.

Region-based EC is in its infancy, and future work can draw on approaches and technologies developed in adjacent domains, including cloud computing and commodity hardware. This creates opportunities to develop and deploy a wide range of applications and services in a spirit of practical experimentation.

Bibliography

- [1]European Telecommunications Standards Institute. *GS MEC 003 - V3.2.1 - Multi-access Edge Computing (MEC); Framework and Reference Architecture*. Apr. 2024. URL: https://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/03.02.01_60/gs_mec003v030201p.pdf (visited on Oct. 27, 2025) (cit. on pp. 1, 7, 8).
- [2]SwitchBot. *SwitchBot's Post*. Oct. 2025. URL: <https://x.com/SwitchBot/status/1980212044281565246> (visited on Oct. 27, 2025) (cit. on pp. 2, 9).
- [3]Amazon Web Services, Inc. *Summary of the Amazon DynamoDB Service Disruption in the Northern Virginia (US-EAST-1) Region*. Oct. 2025. URL: <https://aws.amazon.com/jp/message/101925/> (visited on Oct. 27, 2025) (cit. on pp. 2, 9).
- [4]European Data Protection Board. *Binding Decision 1/2023 on the dispute submitted by the Irish SA on data transfers by Meta Platforms Ireland Limited for its Facebook service (Art. 65 GDPR)*. May 2023. URL: https://www.edpb.europa.eu/our-work-tools/our-documents/binding-decision-board-art-65/binding-decision-12023-dispute-submitted%5C_en (visited on Mar. 5, 2025) (cit. on pp. 2, 9, 101).
- [5]European Data Protection Board. *Dutch SA imposes a fine of 290 million euro on Uber because of transfers of drivers' data to the US*. Aug. 2024. URL: https://www.edpb.europa.eu/news/news/2024/dutch-sa-imposes-fine-290-million-euro-uber-be-cause-transfers-drivers-data-us%5C_en (visited on Mar. 5, 2025) (cit. on pp. 2, 9, 101).
- [6]Muhammad Baqer Mollah, Md. Abul Kalam Azad, and Athanasios Vasilakos. "Secure Data Sharing and Searching at the Edge of Cloud-Assisted Internet of Things". In: *IEEE Cloud Computing* 4.1 (2017), pp. 34–42. DOI: 10.1109/MCC.2017.9 (cit. on pp. 2, 17).
- [7]Craig Gentry. "A fully homomorphic encryption scheme". AAI3382729. PhD thesis. Stanford, CA, USA, 2009 (cit. on pp. 2, 17).
- [8]Thanh Dat Dang and Doan Hoang. "A data protection model for fog computing". In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. 2017, pp. 32–38. DOI: 10.1109/FMEC.2017.7946404 (cit. on pp. 2, 9).
- [9]Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. "A Large-Scale Analysis of the Security of Embedded Firmwares". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 95–110. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin> (cit. on pp. 3, 12).

- [10]Jingjing Ren, Daniel J. Dubois, David Choffnes, et al. “Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach”. In: *Proceedings of the Internet Measurement Conference*. IMC ’19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 267–279. DOI: 10.1145/3355369.3355577. URL: <https://doi.org/10.1145/3355369.3355577> (cit. on pp. 3, 12).
- [11]Yao Chiang, Yi Zhang, Hao Luo, et al. “Management and Orchestration of Edge Computing for IoT: A Comprehensive Survey”. In: *IEEE Internet of Things Journal* 10.16 (2023), pp. 14307–14331. DOI: 10.1109/JIOT.2023.3245611 (cit. on p. 7).
- [12]Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. “Fog computing and its role in the internet of things”. In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC ’12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 13–16. DOI: 10.1145/2342509.2342513. URL: <https://doi.org/10.1145/2342509.2342513> (cit. on p. 7).
- [13]Abderrahime Filali, Amine Abouaomar, Soumaya Cherkaoui, Abdellatif Kobbane, and Mohsen Guizani. “Multi-Access Edge Computing: A Survey”. In: *IEEE Access* 8 (2020), pp. 197017–197046. DOI: 10.1109/ACCESS.2020.3034136 (cit. on p. 8).
- [14]Amazon Web Services, Inc. *AWS IoT – Industrial, Consumer, Commercial, Automotive – AWS*. URL: <https://aws.amazon.com/iot/> (visited on Nov. 3, 2025) (cit. on p. 8).
- [15]Amazon Web Services, Inc. *Serverless Computing - AWS Lambda - Amazon Web Services*. URL: <https://aws.amazon.com/lambda/> (visited on Jan. 10, 2026) (cit. on p. 8).
- [16]Amazon Web Services, Inc. *Cloud Object Storage - Amazon S3 - Amazon Web Services*. URL: <https://aws.amazon.com/s3/> (visited on Jan. 10, 2026) (cit. on p. 8).
- [17]Amazon Web Services, Inc. *On Premises Private Cloud - AWS Outposts - AWS*. URL: <https://aws.amazon.com/outposts/> (visited on Feb. 22, 2026) (cit. on p. 10).
- [18]Oracle Corporation. *Oracle Alloy*. URL: <https://www.oracle.com/cloud/alloy/> (visited on Feb. 22, 2026) (cit. on p. 10).
- [19]Connectivity Standards Alliance. *Matter Specification Version 1.4*. Nov. 2024. URL: https://csa-iot.org/wp-content/uploads/2024/11/24-27349-006_Matter-1.4-Core-Specification.pdf (visited on Nov. 3, 2025) (cit. on p. 11).
- [20]Linghe Kong, Jinlin Tan, Junqin Huang, et al. “Edge-computing-driven Internet of Things: A Survey”. In: *ACM Comput. Surv.* 55.8 (Dec. 2022). DOI: 10.1145/3555308. URL: <https://doi.org/10.1145/3555308> (cit. on p. 11).
- [21]PeiYun Zhang, MengChu Zhou, and Giancarlo Fortino. “Security and trust issues in Fog computing: A survey”. In: *Future Generation Computer Systems* 88 (2018), pp. 16–27. DOI: <https://doi.org/10.1016/j.future.2018.05.008>. URL: <https://www.sciencedirect.com/science/article/pii/S0167739X17329722> (cit. on p. 11).
- [22]Nick McKeown, Tom Anderson, Hari Balakrishnan, et al. “OpenFlow: enabling innovation in campus networks”. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. DOI: 10.1145/1355734.1355746. URL: <https://doi.org/10.1145/1355734.1355746> (cit. on p. 12).

- [23] Syed Akbar Mehdi, Junaid Khalid, and Syed Ali Khayam. “Revisiting Traffic Anomaly Detection Using Software Defined Networking”. In: *Recent Advances in Intrusion Detection*. Ed. by Robin Sommer, Davide Balzarotti, and Gregor Maier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 161–180 (cit. on p. 12).
- [24] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. “FLOWGUARD: building robust firewalls for software-defined networks”. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 97–102. DOI: 10.1145/2620728.2620749. URL: <https://doi.org/10.1145/2620728.2620749> (cit. on p. 12).
- [25] Martin Casado, Michael J. Freedman, Justin Pettit, et al. “Ethane: taking control of the enterprise”. In: *SIGCOMM Comput. Commun. Rev.* 37.4 (Aug. 2007), pp. 1–12. DOI: 10.1145/1282427.1282382. URL: <https://doi.org/10.1145/1282427.1282382> (cit. on p. 12).
- [26] Nick Feamster. “Outsourcing home network security”. In: *Proceedings of the 2010 ACM SIGCOMM Workshop on Home Networks*. HomeNets ’10. New Delhi, India: Association for Computing Machinery, 2010, pp. 37–42. DOI: 10.1145/1851307.1851317. URL: <https://doi.org/10.1145/1851307.1851317> (cit. on pp. 12, 22, 28, 42).
- [27] The Eclipse Foundation. *OSGi Working Group*. URL: <https://www.osgi.org/> (visited on Nov. 3, 2025) (cit. on p. 13).
- [28] A. Fuggetta, G.P. Picco, and G. Vigna. “Understanding code mobility”. In: *IEEE Transactions on Software Engineering* 24.5 (1998), pp. 342–361. DOI: 10.1109/32.685258 (cit. on p. 13).
- [29] James Gosling and Henry McGilton. “The Java language environment”. In: *Sun Microsystems Computer Company* 2550 (1995), p. 38 (cit. on p. 13).
- [30] Luca Cardelli. “A language with distributed scope”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 286–297. DOI: 10.1145/199448.199516. URL: <https://doi.org/10.1145/199448.199516> (cit. on p. 13).
- [31] David Wong, Noemi Paciorek, and Dana Moore. “Java-based mobile agents”. In: *Commun. ACM* 42.3 (Mar. 1999), 92–ff. DOI: 10.1145/295685.295717. URL: <https://doi.org/10.1145/295685.295717> (cit. on p. 13).
- [32] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. “Efficient software-based fault isolation”. In: *SIGOPS Oper. Syst. Rev.* 27.5 (Dec. 1993), pp. 203–216. DOI: 10.1145/173668.168635. URL: <https://doi.org/10.1145/173668.168635> (cit. on p. 13).
- [33] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. “Efficient and language-independent mobile programs”. In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI ’96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 127–136. DOI: 10.1145/231379.231402. URL: <https://doi.org/10.1145/231379.231402> (cit. on p. 13).

- [34]Bennet Yee, David Sehr, Gregory Dardyk, et al. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. In: *2009 30th IEEE Symposium on Security and Privacy*. 2009, pp. 79–93. DOI: 10.1109/SP.2009.25 (cit. on p. 13).
- [35]W3C Web Assembly Community Group. *WebAssembly*. URL: <https://webassembly.org/> (visited on Mar. 5, 2025) (cit. on pp. 14, 103, 112).
- [36]Paul Barham, Boris Dragovic, Keir Fraser, et al. “Xen and the art of virtualization”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 164–177. DOI: 10.1145/945445.945462. URL: <https://doi.org/10.1145/945445.945462> (cit. on p. 14).
- [37]Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. “The Case for VM-Based Cloudlets in Mobile Computing”. In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23. DOI: 10.1109/MPRV.2009.82 (cit. on p. 14).
- [38]Keith Adams and Ole Agesen. “A comparison of software and hardware techniques for x86 virtualization”. In: *SIGOPS Oper. Syst. Rev.* 40.5 (Oct. 2006), pp. 2–13. DOI: 10.1145/1168917.1168860. URL: <https://doi.org/10.1145/1168917.1168860> (cit. on p. 14).
- [39]Yaozu Dong, Xiaowei Yang, Xiaoyong Li, et al. “High performance network virtualization with SR-IOV”. In: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 2010, pp. 1–10. DOI: 10.1109/HPCA.2010.5416637 (cit. on p. 14).
- [40]Filipe Manco, Costin Lupu, Florian Schmidt, et al. “My VM is Lighter (and Safer) than your Container”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. DOI: 10.1145/3132747.3132763. URL: <https://doi.org/10.1145/3132747.3132763> (cit. on p. 14).
- [41]Alexandru Agache, Marc Brooker, Alexandra Iordache, et al. “Firecracker: Lightweight Virtualization for Serverless Applications”. In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. URL: <https://www.usenix.org/conference/nsdi20/presentation/agache> (cit. on pp. 14, 15).
- [42]Gaurav Banga and Jeffrey C. Mogul. “Resource Containers: A New Facility for Resource Management in Server Systems”. In: *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*. New Orleans, LA: USENIX Association, Feb. 1999. URL: <https://www.usenix.org/conference/osdi-99/resource-containers-new-facility-resource-management-server-systems> (cit. on p. 14).
- [43]Poul-Henning Kamp and Robert NM Watson. “Jails: Confining the omnipotent root”. In: *Proceedings of the 2nd International SANE Conference*. 2000 (cit. on p. 14).
- [44]Open source container-based virtualization for Linux. URL: <https://openvz.org/> (visited on Nov. 3, 2025) (cit. on p. 14).

- [45]Daniel Price and Andrew Tucker. “Solaris Zones: Operating System Support for Consolidating Commercial Workloads”. In: *18th Large Installation System Administration Conference (LISA 04)*. Atlanta, GA: USENIX Association, Nov. 2004. URL: <https://www.usenix.org/conference/lisa-04/solaris-zones-operating-system-support-consolidating-commercial-workloads> (cit. on p. 14).
- [46]dlezcano et al. *LXC - Linux Containers*. 2008. URL: <https://github.com/lxc/lxc> (visited on May 17, 2024) (cit. on pp. 14, 49).
- [47]Docker, Inc. *Docker*. 2013. URL: <https://github.com/docker> (visited on Jan. 15, 2026) (cit. on p. 14).
- [48]Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. “Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors”. In: *SIGOPS Oper. Syst. Rev.* 41.3 (Mar. 2007), pp. 275–287. DOI: 10.1145/1272998.1273025. URL: <https://doi.org/10.1145/1272998.1273025> (cit. on p. 14).
- [49]Alexandru Ene. *wasi-threads*. 2024. URL: <https://github.com/WebAssembly/wasi-threads> (visited on Dec. 30, 2025) (cit. on p. 15).
- [50]Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, et al. “Unikernels: library operating systems for the cloud”. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. DOI: 10.1145/2451116.2451167. URL: <https://doi.org/10.1145/2451116.2451167> (cit. on p. 15).
- [51]Avi Kivity, Dor Laor, Glauber Costa, et al. “OSv—Optimizing the Operating System for Virtual Machines”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 61–72. URL: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity> (cit. on p. 15).
- [52]OASIS. *Virtual I/O Device (VIRTIO) Version 1.3*. Oct. 2023. URL: <https://docs.oasis-open.org/virtio/virtio/v1.3/csd01/virtio-v1.3-csd01.html> (visited on Dec. 30, 2025) (cit. on p. 15).
- [53]The QEMU Project Developers. *Device Emulation - QEMU documentation*. 2026. URL: <https://www.qemu.org/docs/master/system/device-emulation.html#device-pass-through> (visited on Jan. 13, 2026) (cit. on p. 15).
- [54]National Institute of Standards and Technology. *CVE-2019-5736*. 2019. URL: <https://nvd.nist.gov/vuln/detail/CVE-2019-5736> (visited on May 17, 2024) (cit. on pp. 16, 45).
- [55]giuseppe. *rootfs: umount all procfs and sysfs with -no-pivot*. 2019. URL: <https://github.com/opencontainers/runc/pull/1962> (visited on May 17, 2024) (cit. on pp. 16, 45).
- [56]National Institute of Standards and Technology. *CVE-2019-14271*. 2019. URL: <https://nvd.nist.gov/vuln/detail/CVE-2019-14271> (visited on May 17, 2024) (cit. on pp. 16, 45).

- [57]dmcgowan. *archive: check whiteout path before removal*. 2018. URL: <https://github.com/containerd/containerd/pull/2001> (visited on May 17, 2024) (cit. on pp. 16, 45).
- [58]Aleksa Sarai et al. *Rootless Containers*. 2017. URL: <https://rootlesscontainers.rs/> (visited on May 17, 2024) (cit. on pp. 16, 45).
- [59]Claus Pahl and Brian Lee. “Containers and Clusters for Edge Cloud Architectures – A Technology Review”. In: *2015 3rd International Conference on Future Internet of Things and Cloud*. 2015, pp. 379–386. DOI: 10.1109/FiCloud.2015.35 (cit. on p. 16).
- [60]Cloud Native Computing Foundation. *Kubernetes*. 2014. URL: <https://github.com/kubernetes/kubernetes> (visited on May 17, 2024) (cit. on pp. 16, 46).
- [61]*KubeEdge*. URL: <https://kubedge.io/> (visited on Nov. 3, 2025) (cit. on pp. 16, 128).
- [62]*KubeEdge. EdgeSite: Standalone Cluster at edge*. URL: <https://kubedge-docs.readthedocs.io/en/latest/modules/edgesite.html> (visited on Nov. 3, 2025) (cit. on p. 16).
- [63]Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. “Starlight: Fast Container Provisioning on the Edge and over the WAN”. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Apr. 2022, pp. 35–50. URL: <https://www.usenix.org/conference/nsdi22/presentation/chen-jun-lin> (cit. on pp. 16, 66, 68, 70, 72, 75, 84).
- [64]containerd. *eStargz*. 2022. URL: <https://github.com/containerd/stargz-snapshotter> (visited on July 31, 2024) (cit. on pp. 16, 66, 69, 70, 75).
- [65]Cynthia Dwork. “Differential Privacy”. In: *Automata, Languages and Programming*. Ed. by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–12 (cit. on p. 17).
- [66]Naoki Matsumoto, Daisuke Kotani, and Yasuo Okabe. “CREBAS: Enabling Network Access Control in a Home with One Click”. In: *Journal of Information Processing* 31 (2023), pp. 174–184. DOI: 10.2197/ipsjjip.31.174 (cit. on p. 19).
- [67]Carlo Vallati, Antonio Viridis, Enzo Mingozzi, and Giovanni Stea. “Mobile-Edge Computing Come Home Connecting things in future smart homes using LTE device-to-device communications”. In: *IEEE Consumer Electronics Magazine* 5.4 (2016), pp. 77–83 (cit. on p. 19).
- [68]ECHONET Consortium. *The ECHONET Lite Specification Version 1.13*. 2018. URL: https://echonet.jp/spec_v113_lite_en/ (visited on Jan. 15, 2026) (cit. on p. 19).
- [69]Netfix, Inc. *DIAL Discovery And Launch protocol specification Version 2.2.1*. 2021. URL: <https://www.dial-multiscreen.org/dial/protocol-specification> (visited on Jan. 15, 2026) (cit. on pp. 19, 32, 38).
- [70]R. Mortier, T. Rodden, T. Lodge, et al. “Control and understanding: Owning your home network”. In: *COMSNETS 2012*. 2012, pp. 1–10 (cit. on pp. 20, 22, 28, 42).

- [71]Eliot Lear, Ralph Droms, and Dan Romascanu. *Manufacturer Usage Description Specification*. RFC 8520. Mar. 2019 (cit. on pp. 20, 22, 28, 42).
- [72]Nick McKeown, Tom Anderson, Hari Balakrishnan, et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74 (cit. on p. 20).
- [73]Yaxing Yao, Justin Reed Basdeo, Smirity Kaushik, and Yang Wang. “Defending My Castle: A Co-Design Study of Privacy Mechanisms for Smart Homes”. In: *CHI ’19*. ACM, 2019, pp. 1–12 (cit. on pp. 20, 22, 23).
- [74]Michelle L. Mazurek, J. P. Arsenault, Joanna Bresee, et al. “Access Control for Home Data Sharing: Attitudes, Needs and Practices”. In: *CHI ’10*. ACM, 2010, pp. 645–654 (cit. on pp. 20, 21, 23, 24, 42).
- [75]Michelle L. Mazurek, Peter F. Klemperer, Richard Shay, et al. “Exploring Reactive Access Control”. In: *CHI ’11*. ACM, 2011, pp. 2085–2094 (cit. on pp. 20, 23, 24).
- [76]William Jang, Adil Chhabra, and Aarathi Prasad. “Enabling Multi-User Controls in Smart Home Devices”. In: *IoTS&P ’17*. Dallas, Texas, USA: ACM, 2017, pp. 49–54 (cit. on pp. 20, 23, 40).
- [77]Maia Jacobs, Henriette Cramer, and Louise Barkhuus. “Caring About Sharing: Couples’ Practices in Single User Device Access”. In: *GROUP ’16*. Sanibel Island, Florida, USA: ACM, 2016, pp. 235–243 (cit. on pp. 20, 23, 40).
- [78]Tara Matthews, Kerwell Liao, Anna Turner, et al. ““She’ll Just Grab Any Device That’s Closer”: A Study of Everyday Device & Account Sharing in Households”. In: *CHI ’16*. ACM, 2016, pp. 5921–5932 (cit. on pp. 20, 23, 40).
- [79]Serge Egelman, A.J. Bernheim Brush, and Kori M. Inkpen. “Family Accounts: A New Paradigm for User Accounts within the Home Environment”. In: *CSCW ’08*. San Diego, CA, USA: ACM, 2008, pp. 669–678 (cit. on pp. 20, 23, 41).
- [80]Ricardo Flores Moyano, David Fernandez Cambronero, and Luis Bellido Triana. “A User-Centric SDN Management Architecture for NFV-Based Residential Networks”. In: *Comput. Stand. Interfaces* 54.P4 (Nov. 2017), pp. 279–292 (cit. on pp. 22, 28, 42).
- [81]Amr Osman, Armin Wasicek, Stefan Köpsell, and Thorsten Strufe. “Transparent Microsegmentation in Smart Home IoT Networks”. In: *HotEdge ’20*. USENIX, June 2020, pp. 1–6 (cit. on pp. 22, 28, 42).
- [82]Henry M. Levy. “1 - Capability- and Object-Based System Concepts”. In: *Capability-Based Computer Systems*. Ed. by Henry M. Levy. Digital Press, 1984, pp. 1–20 (cit. on p. 23).
- [83]S. Gusmeroli, S. Piccione, and D. Rotondi. “IoT Access Control Issues: A Capability Based Approach”. In: *IMIS 2012*. 2012, pp. 787–792 (cit. on p. 23).
- [84]Scott Rose, Oliver Borchert, Stuart Mitchell, and Sean Connelly. *Zero Trust Architecture*. en. Aug. 2020. DOI: <https://doi.org/10.6028/NIST.SP.800-207>. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=930420 (cit. on p. 25).

- [85]Mohamed Boucadair, Reinaldo Penno, and Dan Wing. *Universal Plug and Play (UPnP) Internet Gateway Device - Port Control Protocol Interworking Function (IGD-PCP IWF)*. RFC 6970. July 2013. DOI: 10.17487/RFC6970. URL: <https://www.rfc-editor.org/info/rfc6970> (cit. on p. 31).
- [86]Sony. *IP Control BRAVIA Professional Displays*. 2019. URL: <https://pro-bravia.sony.net/develop/integrate/ip-control/index.html> (visited on Jan. 15, 2026) (cit. on pp. 32, 38).
- [87]F-Secure. *THE CONNECTED HOME'S NEXT WAVE*. 2020. URL: https://blog-assets.f-secure.com/wp-content/uploads/2020/07/23093355/2020-07-02_FS_CHS-report_A4_RGB_final.pdf (visited on Jan. 15, 2026) (cit. on p. 36).
- [88]Communications and Information network Association of Japan. *Usage Survey of Mobile Devices in 2020 (in Japanese)*. 2020. URL: <https://www.ciaj.or.jp/pressrelease2020/6280.html> (visited on Jan. 15, 2026) (cit. on p. 36).
- [89]Ministry of Internal Affairs and Communications, Japan. *WHITE PAPER 2021, Information and Communications in Japan*. 2021. URL: <https://www.soumu.go.jp/johotsusintokei/whitepaper/eng/WP2021/2021-index.html> (visited on Jan. 15, 2026) (cit. on p. 36).
- [90]Statistics Japan. *Statistical Handbook of Japan 2021*. 2021. URL: <https://www.stat.go.jp/english/data/handbook/pdf/2021all.pdf> (visited on Jan. 15, 2026) (cit. on p. 36).
- [91]Md Mainuddin, Zhenhai Duan, and Yingfei Dong. “Network Traffic Characteristics of IoT Devices in Smart Homes”. In: *ICCCN 2021*. 2021, pp. 1–11 (cit. on pp. 37, 38).
- [92]Netflix. *dial-reference*. 2013. URL: <https://github.com/Netflix/dial-reference> (visited on Jan. 15, 2026) (cit. on p. 39).
- [93]Martin Casado, Michael J. Freedman, Justin Pettit, et al. “Ethane: taking control of the enterprise”. In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '07. Kyoto, Japan: Association for Computing Machinery, 2007, pp. 1–12. DOI: 10.1145/1282380.1282382. URL: <https://doi.org/10.1145/1282380.1282382> (cit. on p. 39).
- [94]Naser Peiravian and Xingquan Zhu. “Machine Learning for Android Malware Detection Using Permission and API Calls”. In: *ICTAI 2013*. 2013, pp. 300–305 (cit. on p. 43).
- [95]Naoki Matsumoto and Akihiro Suda. “bypass4netns: Accelerating TCP/IP Communications in Rootless Containers”. In: *Proceedings of the 19th Asian Internet Engineering Conference*. AINTEC '24. Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 67–75. DOI: 10.1145/3674213.3674221. URL: <https://doi.org/10.1145/3674213.3674221> (cit. on p. 45).
- [96]Michael Crosby et al. *runc*. 2014. URL: <https://github.com/opencontainers/runc/> (visited on May 17, 2024) (cit. on p. 45).
- [97]Michael Crosby et al. *containerd*. 2016. URL: <https://github.com/containerd/containerd> (visited on May 17, 2024) (cit. on pp. 45, 56, 82).

- [98]Xin Lin, Lingguang Lei, Yuewu Wang, et al. “A Measurement Study on Linux Container Security: Attacks and Countermeasures”. In: ACSAC ’18. ACM, 2018, pp. 418–429. DOI: 10.1145/3274694.3274720 (cit. on p. 45).
- [99]Akihiro Suda et al. *RootlessKit*. 2018. URL: <https://github.com/rootless-containers/rootlesskit> (visited on May 17, 2024) (cit. on pp. 45, 47, 49).
- [100]Akihiro Suda et al. *slirp4netns*. 2018. URL: <https://github.com/rootless-containers/slirp4netns> (visited on May 17, 2024) (cit. on pp. 45, 48, 49).
- [101]Ryo Nakamura, Yuji Sekiya, and Hajime Tazaki. “Grafting sockets for fast container networking”. In: *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*. ANCS ’18. Ithaca, New York: Association for Computing Machinery, 2018, pp. 15–27. DOI: 10.1145/3230718.3230723. URL: <https://doi.org/10.1145/3230718.3230723> (cit. on pp. 46, 49, 51).
- [102]Sunyanan Choochotkaew, Tatsuhiko Chiba, Scott Trent, and Marcelo Amaral. “Bypass Container Overlay Networks with Transparent BPF-driven Socket Replacement”. In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 2022, pp. 134–143. DOI: 10.1109/CLOUD55607.2022.00033 (cit. on pp. 46, 49).
- [103]Michael Kerrisk et al. *seccomp_unotify(2)*. 2021. URL: https://man7.org/linux/man-pages/man2/seccomp_unotify.2.html (visited on May 17, 2024) (cit. on pp. 46, 48).
- [104]Akihiro Suda et al. *Usernetes*. 2018. URL: <https://github.com/rootless-containers/usernetes> (visited on May 17, 2024) (cit. on pp. 46, 54).
- [105]Michael Kerrisk et al. *seccomp(2)*. 2021. URL: <https://man7.org/linux/man-pages/man2/seccomp.2.html> (visited on May 17, 2024) (cit. on p. 48).
- [106]Serge Hallyn et al. *lxc-user-nic*. 2013. URL: <https://github.com/lxc/lxc/blob/master/doc/lxc-user-nic.sgml.in> (visited on May 17, 2024) (cit. on p. 49).
- [107]Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, et al. “Slim: OS Kernel Support for a Low-Overhead Container Overlay Network”. In: *NSDI 19*. USENIX, 2019, pp. 331–344 (cit. on pp. 49, 51).
- [108]National Institute of Standards and Technology. “CVE-2017-5985”. In: (2017). URL: <https://nvd.nist.gov/vuln/detail/CVE-2017-5985> (visited on May 17, 2024) (cit. on p. 49).
- [109]National Institute of Standards and Technology. *CVE-2018-6556*. 2018. URL: <https://nvd.nist.gov/vuln/detail/CVE-2018-6556> (visited on May 17, 2024) (cit. on p. 49).
- [110]Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. “SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM ’22. Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 780–794. DOI: 10.1145/3544216.3544259. URL: <https://doi.org/10.1145/3544216.3544259> (cit. on p. 50).

- [111]Akihiro Suda. *[CVE-2020-15257] Don't use -net=host. Don't use spec.hostNetwork.* 2020. URL: <https://medium.com/nttlabs/dont-use-host-network-namespace-f548aeef575> (visited on Jan. 15, 2026) (cit. on p. 51).
- [112]Itay Shakury et al. *tracee*. 2019. URL: <https://github.com/aquasecurity/tracee> (visited on May 17, 2024) (cit. on p. 51).
- [113]Cloud Native Computing Foundation. *Cluster Networking | Kubernetes*. 2024. URL: <https://kubernetes.io/docs/concepts/cluster-administration/networking/> (visited on May 17, 2024) (cit. on p. 54).
- [114]flannel-io. *flannel*. 2014. URL: <https://github.com/flannel-io/flannel> (visited on May 17, 2024) (cit. on p. 54).
- [115]Project Calico. *Calico*. 2014. URL: <https://github.com/projectcalico/calico> (visited on May 17, 2024) (cit. on p. 54).
- [116]Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, et al. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. Aug. 2014. DOI: 10.17487/RFC7348 (cit. on p. 54).
- [117]Cloud Native Computing Foundation. *Service | Kubernetes*. 2024. URL: <https://kubernetes.io/docs/concepts/services-networking/service/> (visited on May 17, 2024) (cit. on p. 55).
- [118]Akihiro Suda. *Usernetes Gen2 - Kubernetes in Rootless Docker, with Multiple Nodes*. 2024. URL: <https://github.com/AkihiroSuda/AkihiroSuda/blob/master/slides/2024/20240415> (visited on June 24, 2024) (cit. on p. 55).
- [119]Akihiro Suda et al. *nerdctl*. 2020. URL: <https://github.com/containerd/nerdctl/> (visited on May 17, 2024) (cit. on p. 56).
- [120]*apt (2.4.12)*. 2024. URL: <https://salsa.debian.org/apt-team/apt> (visited on May 17, 2024) (cit. on p. 59).
- [121]*iperf3 (3.16)*. 2023. URL: <https://iperf.fr/> (visited on May 17, 2024) (cit. on p. 59).
- [122]*nginx (1.18)*. 2020. URL: <https://nginx.org/> (visited on May 17, 2024) (cit. on pp. 59, 60).
- [123]*Redis (7.2.3)*. 2023. URL: <https://redis.io/> (visited on May 17, 2024) (cit. on pp. 59, 61).
- [124]*memcached (1.6.22)*. 2023. URL: <https://memcached.org/> (visited on May 17, 2024) (cit. on p. 59).
- [125]*RabbitMQ (3.12.10)*. 2023. URL: <https://www.rabbitmq.com/> (visited on May 17, 2024) (cit. on p. 59).
- [126]*etcd (3.3.25)*. 2020. URL: <https://etcd.io/> (visited on May 17, 2024) (cit. on p. 59).
- [127]*PostgreSQL (16.1)*. 2023. URL: <https://www.postgresql.org/> (visited on May 17, 2024) (cit. on p. 59).

- [128]MySQL (8.2.0). 2023. URL: <https://www.mysql.com/> (visited on May 17, 2024) (cit. on pp. 59, 61).
- [129]net/http - Go Packages (Go 1.21.3). 2023. URL: <https://pkg.go.dev/net/http> (visited on May 17, 2024) (cit. on p. 59).
- [130]Redis benchmark (7.2.3). 2023. URL: https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/ (visited on May 17, 2024) (cit. on pp. 59, 61).
- [131]memptier_benchmark (2.0.0). 2023. URL: https://github.com/RedisLabs/memptier_benchmark (visited on May 17, 2024) (cit. on p. 59).
- [132]rabbitmq-perf-tester (2.20.0). 2023. URL: <https://github.com/rabbitmq/rabbitmq-perf-test> (visited on May 17, 2024) (cit. on p. 59).
- [133]etcd/tools/benchmark (3.3.25). 2023. URL: <https://github.com/etcd-io/etcd/tree/main/tools/benchmark> (visited on May 17, 2024) (cit. on p. 59).
- [134]pgbench (16.1). 2023. URL: <https://www.postgresql.org/docs/16/pgbench.html> (visited on May 17, 2024) (cit. on p. 59).
- [135]sysbench (1.0.20). 2020. URL: <https://github.com/akopytov/sysbench> (visited on May 17, 2024) (cit. on pp. 59, 61).
- [136]Shengkai Lin, Peirui Cao, Tianyi Huang, et al. *XMasq: Low-Overhead Container Overlay Network Based on eBPF*. 2023. arXiv: 2305.05455 [cs.NI] (cit. on p. 62).
- [137]Common Weakness Enumeration. *CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition*. 2006. URL: <https://cwe.mitre.org/data/definitions/367.html> (visited on May 17, 2024) (cit. on p. 62).
- [138]Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. “MegaPipe: A New Programming Interface for Scalable Network I/O”. In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 135–148 (cit. on p. 63).
- [139]EunYoung Jeong, Shinae Wood, Muhammad Jamshed, et al. “mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems”. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 489–502 (cit. on p. 63).
- [140]Naoki Matsumoto, Daisuke Kotani, and Yasuo Okabe. “D4C: Fast and Efficient Container Image Updating for Edge with Delta Encodings”. In: *IEICE TRANSACTIONS on Communications E108-B.7* (July 2025), pp. 780–795. DOI: 10.23919/transcom.2024.EBT0008 (cit. on p. 65).
- [141]Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. 2019. DOI: 10.48550/ARXIV.1902.03383 (cit. on p. 65).
- [142]Arif Ahmed and Guillaume Pierre. “Docker Container Deployment in Fog Computing Infrastructures”. In: *2018 IEEE International Conference on Edge Computing (EDGE)*. 2018, pp. 1–8. DOI: 10.1109/EDGE.2018.00008 (cit. on pp. 65, 67, 69, 99).

- [143]Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “Slacker: Fast Distribution with Lazy Docker Containers”. In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Feb. 2016, pp. 181–195. URL: <https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter> (cit. on pp. 66, 70).
- [144]Lian Du, Tianyu Wo, Renyu Yang, and Chunming Hu. “Cider: a Rapid Docker Container Deployment System through Sharing Network Storage”. In: *2017 IEEE HPCC/SmartCity/DSS*. 2017, pp. 332–339. DOI: 10.1109/HPCC-SmartCity-DSS.2017.44 (cit. on pp. 66, 70).
- [145]Chao Zheng, Lukas Rupperecht, Vasily Tarasov, et al. “Wharf: Sharing Docker Images in a Distributed File System”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2018, pp. 174–185. DOI: 10.1145/3267809.3267836 (cit. on pp. 66, 70).
- [146]Lorenzo Civolani, Guillaume Pierre, and Paolo Bellavista. “FogDocker: Start Container Now, Fetch Image Later”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. 2019, pp. 51–59. DOI: 10.1145/3344341.3368811 (cit. on pp. 66, 70).
- [147]Shotaro Gotanda and Takahiro Shinagawa. “Short Paper: Highly Compatible Fast Container Startup with Lazy Layer Pull”. In: *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 2021, pp. 53–59. DOI: 10.1109/IC2E52221.2021.00019 (cit. on pp. 66, 70).
- [148]Valerio Persico, Alessio Botta, Pietro Marchetta, Antonio Montieri, and Antonio Pescapé. “On the performance of the wide-area networks interconnecting public-cloud datacenters around the globe”. In: *Computer Networks* 112 (2017), pp. 67–83. DOI: 10.1016/j.comnet.2016.10.013 (cit. on pp. 66, 70).
- [149]Xinlei Yang, Hao Lin, Zhenhua Li, et al. “Mobile Access Bandwidth in Practice: Measurement, Analysis, and Implications”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 114–128. DOI: 10.1145/3544216.3544237 (cit. on pp. 66, 67, 70, 91).
- [150]Open Container Initiative. *OCI Image Format Specification*. 2023. URL: <https://github.com/opencontainers/image-spec/blob/main/spec.md> (visited on Oct. 24, 2024) (cit. on p. 67).
- [151]Nannan Zhao, Vasily Tarasov, Hadeel Albahar, et al. “Large-Scale Analysis of the Docker Hub Dataset”. In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019, pp. 1–10. DOI: 10.1109/CLUSTER.2019.8891000 (cit. on p. 67).
- [152]Docker Inc. *Multi-stage builds* | *Docker Docs*. 2024. URL: <https://docs.docker.com/build/building/multi-stage/> (visited on Mar. 6, 2024) (cit. on p. 67).
- [153]Docker Community and MySQL Team. *mysql/Dockerfile.debian*. 2024. URL: <https://github.com/docker-library/mysql/blob/master/Dockerfile.debian> (visited on Mar. 6, 2024) (cit. on p. 67).
- [154]David Korn, Joshua P. MacDonald, Jeffrey Mogul, and Kiem-Phong Vo. *The VCDIFF Generic Differencing and Compression Data Format*. RFC 3284. July 2002. DOI: 10.17487/RFC3284. URL: <https://www.rfc-editor.org/info/rfc3284> (cit. on p. 68).

- [155]Josh Macdonald. *xdelta3*. Apr. 2016. URL: <https://github.com/jmacd/xdelta> (visited on July 2, 2024) (cit. on p. 68).
- [156]Colin Percival. *Naive Differences of Executable Code*. 2003. URL: <https://www.daemonology.net/papers/bsdifff.pdf> (visited on July 31, 2024) (cit. on pp. 68, 85).
- [157]Dimitris Skourtis, Lukas Rupperecht, Vasily Tarasov, and Nimrod Megiddo. “Carving Perfect Layers out of Docker Images”. In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. July 2019, pp. 1–8. URL: <https://www.usenix.org/conference/hotcloud19/presentation/skourtis> (cit. on p. 69).
- [158]Zhigang Lu, Jiwei Xu, Yuewen Wu, Tao Wang, and Tao Huang. “An Empirical Case Study on the Temporary File Smell in Dockerfiles”. In: *IEEE Access* 7 (2019), pp. 63650–63659. DOI: 10.1109/ACCESS.2019.2905424 (cit. on p. 69).
- [159]Nannan Zhao, Hadeel Albahar, Subil Abraham, et al. “DupHunter: Flexible High-Performance Deduplication for Docker Registries”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. July 2020, pp. 769–783. URL: <https://www.usenix.org/conference/atc20/presentation/zhao> (cit. on p. 69).
- [160]N Hardi, J Blomer, G Ganis, and R Popescu. “Making containers lazy with Docker and CernVM-FS”. In: *Journal of Physics: Conference Series* 1085.3 (2018), pp. 1–7. DOI: 10.1088/1742-6596/1085/3/032019 (cit. on p. 70).
- [161]Linux Foundation. *Dragonfly*. 2024. URL: <https://d7y.io/> (visited on July 31, 2024) (cit. on p. 70).
- [162]uber. *kraken*. 2024. URL: <https://github.com/uber/kraken> (visited on July 31, 2024) (cit. on p. 70).
- [163]Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. “FID: A Faster Image Distribution System for Docker Platform”. In: *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. 2017, pp. 191–198. DOI: 10.1109/FAS-W.2017.147 (cit. on p. 70).
- [164]Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, et al. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–17. DOI: 10.1145/2741948.2741964 (cit. on p. 70).
- [165]Soeren Becker, Florian Schmidt, and Odej Kao. “EdgePier: P2P-based Container Image Distribution in Edge Computing Environments”. In: *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. 2021, pp. 1–8. DOI: 10.1109/IPCCC51483.2021.9679447 (cit. on p. 70).
- [166]Google. *CRFS: Container Registry Filesystem*. 2019. URL: <https://github.com/google/crfs> (visited on July 31, 2024) (cit. on p. 70).
- [167]Huiba Li, Yifan Yuan, Rui Du, et al. “DADI: Block-Level Image Service for Agile and Elastic Application Deployment”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 727–740. URL: <https://www.usenix.org/conference/atc20/presentation/li-huiba> (cit. on p. 71).
- [168]dragonflyoss. *Nydus: Dragonfly Container Image Service*. 2026. URL: <https://github.com/dragonflyoss/nydus> (visited on Jan. 20, 2026) (cit. on p. 71).

- [169]containers. *containers-storage 1 "August 2024"*. 2026. URL: <https://github.com/containers/storage/blob/main/docs/containers-storage-zstd-chunked.md> (visited on Jan. 20, 2026) (cit. on p. 71).
- [170]Naoki Matsumoto, Daisuke Kotani, and Yasuo Okabe. "Efficient Container Image Updating in Low-bandwidth Networks with Delta Encoding". In: *2023 IEEE International Conference on Cloud Engineering (IC2E)*. 2023, pp. 1–10. DOI: 10.1109/IC2E59103.2023.00009 (cit. on p. 71).
- [171]Barron C. Housel and David B. Lindquist. "WebExpress: A System for Optimizing Web Browsing in a Wireless Environment". In: *Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking*. 1996, pp. 108–116. DOI: 10.1145/236387.236416 (cit. on p. 71).
- [172]Gaurav Banga, Fred Douglass, and Michael Rabinovich. "Optimistic Deltas for WWW Latency Reduction". In: *USENIX 1997 Annual Technical Conference (USENIX ATC 97)*. Jan. 1997. URL: <https://www.usenix.org/conference/usenix-1997-annual-technical-conference/optimistic-deltas-www-latency-reduction> (cit. on p. 71).
- [173]Hidetoshi Teraoka, Fumiharu Nakahara, and Kenichi Kurosawa. "Incremental Update Method for In-vehicle ECUs". In: *IPSJ transactions. CDS 7.2* (May 2017), pp. 41–50. URL: <https://cir.nii.ac.jp/crid/1050282812885095552> (cit. on p. 71).
- [174]Nikolai Samteladze and Ken Christensen. "DELTA++: Reducing the Size of Android Application Updates". In: *IEEE Internet Computing 18.2* (2014), pp. 50–57. DOI: 10.1109/MIC.2013.82 (cit. on p. 71).
- [175]Andrew Tridgell and Paul Mackerras. "The rsync algorithm". In: *The Australian National University* (1996), pp. 1–8. URL: <https://www.andrew.cmu.edu/course/15-749/READINGS/required/cas/tridgell96.pdf> (cit. on p. 71).
- [176]Wen Xia, Can Wei, Zhenhua Li, Xuan Wang, and Xiangyu Zou. "NetSync: A Network Adaptive and Deduplication-Inspired Delta Synchronization Approach for Cloud Storage Services". In: *IEEE Transactions on Parallel and Distributed Systems 33.10* (2022), pp. 2554–2570. DOI: 10.1109/TPDS.2022.3145025 (cit. on p. 71).
- [177]Athicha Muthitacharoen, Benjie Chen, and David Mazières. "A Low-Bandwidth Network File System". In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. 2001, pp. 174–187. DOI: 10.1145/502034.502052 (cit. on p. 71).
- [178]Andrea C. G. Mennucci. *The debdelta suite*. 2006. URL: http://debdelta.debian.net/html/x67.html#no_incremental (visited on July 31, 2024) (cit. on p. 76).
- [179]Ryozo Kiyohara, Koichi Tanaka, and Yoshiaki Terashima. "S/W upgrade for on-vehicle information devices". In: *2012 IEEE International Conference on Consumer Electronics (ICCE)*. 2012, pp. 19–20. DOI: 10.1109/ICCE.2012.6161718 (cit. on p. 78).
- [180]The kernel development community. *FUSE - The Linux Kernel documentation*. 2022. URL: <https://www.kernel.org/doc/html/latest/filesystems/fuse.html> (visited on July 31, 2024) (cit. on pp. 82, 93).
- [181]hanwen. *go-fuse*. 2017. URL: <https://github.com/hanwen/go-fuse> (visited on Oct. 17, 2024) (cit. on p. 82).

- [182]containerd developers. *containerd's document: Content Flow*. 2024. URL: <https://github.com/containerd/containerd/blob/d036988eec4ada7c313f0d111acf1551bae8a5be/docs/content-flow.md> (visited on Oct. 23, 2024) (cit. on p. 83).
- [183]containerd developers. *containerd's document: containerd Plugins*. 2024. URL: <https://github.com/containerd/containerd/blob/a3d84a172783c329968d08242680ffe788a088dd/docs/PLUGINS.md> (visited on Oct. 23, 2024) (cit. on p. 83).
- [184]Docker,Inc. *postgres - Official Image |Docker Hub*. 2024. URL: https://hub.docker.com/_/postgres (visited on July 18, 2024) (cit. on p. 83).
- [185]Docker,Inc. *redis - Official Image |Docker Hub*. 2024. URL: https://hub.docker.com/_/redis (visited on July 18, 2024) (cit. on p. 83).
- [186]Docker,Inc. *nginx - Official Image |Docker Hub*. 2024. URL: https://hub.docker.com/_/nginx (visited on July 18, 2024) (cit. on p. 83).
- [187]PyTorch. *pytorch/pytorch - Official Image |Docker Hub*. 2024. URL: <https://hub.docker.com/r/pytorch/pytorch> (visited on July 18, 2024) (cit. on p. 83).
- [188]Josh Macdonald. *xdelta3.c*. Apr. 2016. URL: https://github.com/jmacd/xdelta/blob/release3_1_apl/xdelta3/xdelta3.c (visited on July 2, 2024) (cit. on p. 86).
- [189]GNU Project. *GNU Gzip: General file (de)compression*. 2023. URL: <https://www.gnu.org/software/gzip/manual/gzip.html> (visited on July 2, 2024) (cit. on p. 86).
- [190]J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714 (cit. on p. 86).
- [191]Blesson Varghese, Eyal de Lara, Aaron Yi Ding, et al. "Revisiting the Arguments for Edge Computing Research". In: *IEEE Internet Computing* 25.5 (2021), pp. 36–42. DOI: 10.1109/MIC.2021.3093924 (cit. on p. 91).
- [192]The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. URL: <https://www.postgresql.org/> (visited on July 31, 2024) (cit. on p. 94).
- [193]The Linux Foundation. *PyTorch*. URL: <https://pytorch.org/> (visited on July 31, 2024) (cit. on p. 94).
- [194]The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 16: pgbench*. 2024. URL: <https://www.postgresql.org/docs/current/pgbench.html> (visited on July 18, 2024) (cit. on p. 94).
- [195]Robert J. Hanson. *TPC-B*. 1990. URL: <https://www.tpc.org/tpcb/> (visited on July 31, 2024) (cit. on p. 94).
- [196]Lukas Hedegaard. *PyTorch-Benchmark*. Version 0.3.5. 2022. DOI: 10.5281/zenodo.7223585. URL: <https://github.com/pytorch/benchmark> (visited on Jan. 15, 2026) (cit. on p. 94).

- [197]Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. DOI: <https://doi.org/10.48550/arXiv.1905.11946>. URL: <https://arxiv.org/abs/1905.11946> (visited on Jan. 15, 2026) (cit. on p. 94).
- [198]SATA-IO Board Members. *Serial ATA International Organization: Serial ATA Revision 3.1*. July 2011. URL: https://sata-io.org/system/files/specifications/SerialATA_Revision_3_1_Gold.pdf (visited on Oct. 23, 2024) (cit. on p. 99).
- [199]Naoki Matsumoto, Tomoya Kawanishi, Kenji Ohira, and Masahiro Kozuka. “yak: WebAssembly-based Data Analysis Platform with Data Providers Driven Analysis Control”. In: *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*. © 2025 IEEE. Reprinted, with permission. 2025, pp. 1116–1125. DOI: 10.1109/COMPSAC65507.2025.00144 (cit. on p. 101).
- [200]The European Parliament and The council of The European Union. “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)”. In: *Official Journal of the European Union* (2016) (cit. on p. 101).
- [201]Christian Priebe, Kapil Vaswani, and Manuel Costa. “EnclaveDB: A Secure Database Using SGX”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 264–278 (cit. on pp. 101–104).
- [202]Kai Li, Yuzhe Tang, Qi Zhang, Jianliang Xu, and Ju Chen. “Authenticated Key-Value Stores with Hardware Enclaves”. In: *Proceedings of the 22nd International Middleware Conference: Industrial Track*. Middleware ’21. Association for Computing Machinery, 2021, pp. 1–8 (cit. on pp. 101–104).
- [203]Rafael Pires, Daniel Gavril, Pascal Felber, Emanuel Onica, and Marcelo Pasin. “A Lightweight MapReduce Framework for Secure Processing with SGX”. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2017, pp. 1100–1107 (cit. on pp. 101–104).
- [204]Snowflake Inc. *Preserving Data Privacy in Life Sciences: How Snowflake Data Clean Rooms Make It Happen*. May 2024. URL: <https://www.snowflake.com/en/blog/data-privacy-life-sciences-clean-rooms/> (visited on Mar. 5, 2025) (cit. on p. 102).
- [205]LiveRamp. *Data Collaboration Platform*. 2025. URL: <https://liveramp.com/our-platform/> (visited on Mar. 5, 2025) (cit. on p. 102).
- [206]Nicolas Anciaux, Philippe Bonnet, Luc Bouganim, et al. “Personal Data Management Systems: The security and functionality standpoint”. In: *Information Systems* 80 (2019), pp. 13–35. DOI: <https://doi.org/10.1016/j.is.2018.09.002>. URL: <https://www.sciencedirect.com/science/article/pii/S0306437918304022> (cit. on pp. 102, 104).

- [207]Robin Carpentier, Floris Thiant, Iulian Sandu Popa, Nicolas Anciaux, and Luc Bouganim. “An Extensive and Secure Personal Data Management System Using SGX”. In: *EDBT 2022 - 25th International Conference on Extending Database Technology*. Mar. 2022 (cit. on pp. 102, 104).
- [208]Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding Applications from an Untrusted Cloud with Haven”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Oct. 2014, pp. 267–283 (cit. on p. 103).
- [209]Chia-che Tsai, Donald E. Porter, and Mona Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, July 2017, pp. 645–658 (cit. on p. 103).
- [210]Sergei Arnaudov, Bohdan Trach, Franz Gregor, et al. “SCONE: Secure Linux Containers with Intel SGX”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Nov. 2016, pp. 689–703 (cit. on p. 103).
- [211]J. Menetrey, M. Pasin, P. Felber, and V. Schiavoni. “Twine: An Embedded Trusted Runtime for WebAssembly”. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Apr. 2021, pp. 205–216 (cit. on p. 103).
- [212]Riad Ladjel, Nicolas Anciaux, Philippe Pucheral, and Guillaume Scerri. “A manifest-based framework for organizing the management of personal data at the edge of the network”. In: *ISD 2019 - 28th International Conference on Information Systems Development*. Aug. 2019 (cit. on p. 104).
- [213]Emily Griffiths, Carlotta Greci, Yannis Kotrotsios, et al. “Handbook on Statistical Disclosure Control for Outputs”. In: *Safe Data Access Professionals* (June 2020). DOI: <https://doi.org/10.6084/m9.figshare.9958520> (cit. on pp. 106, 121).
- [214]Felix Ritchie and Mark Elliot. “Principles- Versus Rules-Based Output Statistical Disclosure Control In Remote Access Environments”. In: *IASSIST Quarterly* 39 (Dec. 2014). DOI: 10.29173/iq778 (cit. on p. 106).
- [215]Apache Software Foundation. *Arrow Columnar Format*. URL: <https://arrow.apache.org/docs/format/Columnar.html> (visited on Mar. 5, 2025) (cit. on p. 113).
- [216]Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: <https://www.rfc-editor.org/info/rfc7519> (visited on Mar. 5, 2025) (cit. on p. 115).
- [217]Advanced Micro Devices, Inc. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. 2020. URL: <https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf> (visited on Mar. 5, 2025) (cit. on p. 116).
- [218]Jeremy Powell. *AMD SEV-SNP Attestation: Establishing Trust in Guests*. Sept. 2022. URL: <https://www.amd.com/content/dam/amd/en/documents/developer/lss-snp-attestation.pdf> (visited on Mar. 5, 2025) (cit. on pp. 116, 117).

- [219]Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanaivanon, Michael Steiner, and Gene Tsudik. “VRASED: A Verified Hardware/Software Co-Design for Remote Attestation”. In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Aug. 2019, pp. 1429–1446 (cit. on p. 116).
- [220]Eric Rescorla. *Keying Material Exporters for Transport Layer Security (TLS)*. RFC 5705. Mar. 2010. DOI: 10.17487/RFC5705. URL: <https://www.rfc-editor.org/info/rfc5705> (cit. on p. 117).
- [221]Advanced Micro Devices, Inc. *Versioned Chip Endorsement Key (VCEK) Certificate and KDS Interface Specification*. Jan. 2025. URL: <https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/57230.pdf> (visited on Mar. 5, 2025) (cit. on p. 117).
- [222]K. LeFevre, D.J. DeWitt, and R. Ramakrishnan. “Mondrian Multidimensional K-Anonymity”. In: *22nd International Conference on Data Engineering (ICDE’06)*. 2006, pp. 25–25. DOI: 10.1109/ICDE.2006.101 (cit. on p. 122).
- [223]Financial Services Agency of Japan. *EDINET*. 2025. URL: <https://disclosure2.edinet-fsa.go.jp/week0020.aspx> (visited on Mar. 5, 2025) (cit. on p. 122).
- [224]*OpenYurt*. URL: <https://openyurt.io/> (visited on Nov. 18, 2025) (cit. on p. 128).
- [225]The Home Assistant project. *Home Assistant*. 2026. URL: <https://www.home-assistant.io/> (visited on Jan. 13, 2026) (cit. on p. 128).

List of Figures

1.1	Cloud-based Edge Computing architecture	1
1.2	Region-based Edge Computing architecture	2
1.3	Improvements toward a region-based edge computing architecture . . .	4
2.1	Architecture of Region-based Edge Computing	10
3.1	Overview of CREBAS	21
3.2	Overview of the authorization architecture	25
3.3	Example capability	25
3.4	Example binary expression tree	26
3.5	Flows of Authorization	27
3.6	Overview of the PEP	29
3.7	Loading twin devices	30
3.8	Access control methods	31
3.9	Flow of network access control	33
3.10	WebUI of the prototype implementation	34
3.11	Average throughput (iperf3)	35
3.12	Average RTT (ping)	35
3.13	CPU usage in payload-based control with iperf3 (A dotted line is ideal throughput)	36
3.14	CPU usage in proxy-based control with iperf3 (A dotted line is ideal throughput)	36
3.15	CPU usage in payload-based control	37
3.16	CPU usage in proxy-based control	37
4.1	Overview of the rootless container network	47
4.2	The syscall filtering flow in Seccomp Notify	48
4.3	The overview of bypass4netns	50
4.4	Flow of socket switching	53
4.5	Publishing ports with bypass4netns	54
4.6	Multi-node communication approach	55
4.7	Switch overhead	58
4.8	Benchmark with iperf3	60
4.9	Benchmark with nginx	60

4.10	Benchmark with Redis	61
4.11	Benchmark with MySQL	61
5.1	Comparison of delta encodings	68
5.2	Overview of D4C	73
5.3	Overview of delta bundle generation	74
5.4	Structure of a delta bundle	75
5.5	Example DAG-based merge graph	77
5.6	Format of a delta file in bsdiff	79
5.7	Example of DeltaMerging	81
5.8	Reading files with Di3FS	81
5.9	Comparison of delta bundle size	84
5.10	Comparison of time required to generate delta bundles	85
5.11	Effect of multithreaded delta bundle generation	85
5.12	Comparison of time to merge delta bundles and the ratio	87
5.13	Effect of multithreaded merge	88
5.14	Merge performance	88
5.15	Ratio of merging delta file operations in number and time	89
5.16	DeltaMerging performance with file size and number of blocks	90
5.17	Breakdown of merge delta block operations	90
5.18	Time to mount delta bundles	91
5.19	Time to pull update bundles	91
5.20	Open latency comparison between Di3FS and Native	93
5.21	Read performance comparison between Di3FS and Native	93
5.22	Performance benchmark with PyTorch	95
6.1	Analysis platform with data provider's control	105
6.2	Analysis control with data provider's policy	110
6.3	Overview of yak	112
6.4	Data chaining among modules	113
6.5	Analysis procedure is defined with DAG	114
6.6	Handshake and authentication between an analyst and data provider	115
6.7	Remote attestation protocol for data providers	117
6.8	Function var performance	119
6.9	Function LinearRegression performance	119
6.10	Function id performance	119
6.11	Memory usage on the number of values	119

6.12 Performance on	
the length of functions	120
6.13 Memory usage on	
the length of functions	120
6.14 Anonymization by policy: Histogram	121
6.15 Anonymization by policy: Table	122
6.16 Analysis of financial products	123

List of Tables

2.1 Comparison with offloading approaches	15
3.1 Comparison with other network access control methods	22
3.2 Example operations in DIAL	39
4.1 Comparison with socket switching approaches	49
4.2 Traced syscalls related to sockets	52
4.3 Tested Applications	59
5.1 Comparison between applications with delta encoding	71
5.2 Types of entry in metadata	76
5.3 Operations for MergeDeltaBundle	78
5.4 Operations to merge delta blocks	81
5.5 Container images used for evaluations	83
5.6 Evaluation Environment (Virtual Machine)	84
5.7 Files with the greatest size reduction in postgres 13.1→13.2 with bsdiff	86
5.8 Files with the least size reduction in postgres 13.1→13.2 with bsdiff	86
5.9 TPC-B benchmark comparison in postgres 13.2	94
6.1 Comparison of different privacy-preserving data analysis approaches	102

