Simplifying the Deployment of Intrusion-Tolerant Systems By Leveraging Cloud Resources

by

Maher Khan

BSc. in Computer Science, BSc. in Information Systems

Carnegie Mellon University, 2017

Submitted to the Graduate Faculty of the

School of Computing and Information in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2024

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Maher Khan

It was defended on

April 4, 2024

and approved by

Dr. Amy Babay, Department of Computer Science

Dr. Daniel Mossé, Department of Computer Science

Dr. Adam Lee, Department of Computer Science

Dr. Balaji Palanisamy, Department of Informatics and Networked Systems

Copyright © by Maher Khan
2024

Simplifying the Deployment of Intrusion-Tolerant Systems By Leveraging Cloud Resources

Maher Khan, PhD

University of Pittsburgh, 2024

Abstract: The rise of cyberattacks on high-value systems has led to a growing interest in intrusion-tolerant systems as a means of ensuring resilience. An intrusion-tolerant system can guarantee that it can continue to operate correctly even when parts of the system are compromised. The research community has developed techniques for intrusion-tolerant systems based on Byzantine Fault-Tolerant (BFT) replication. However, these systems are still not widely used in industry. One of the main obstacles is the technical expertise and infrastructure investment required for deploying and managing these systems. Cloud resources can help with this but are currently not feasible for many system operators due to concerns about maintaining the confidentiality of sensitive information.

We address this issue by developing novel systems that allow system operators to deploy intrusion-tolerant applications by partially or fully outsourcing the responsibility of the BFT replication protocol to a cloud service while maintaining the privacy of the application's state and algorithms. We define a hybrid management model for joint management of intrusion-tolerant applications by system operators and cloud service providers, separating responsibilities. Only the replicas managed by the system operator execute the application logic, and the replicas managed by the cloud service provider participate in the BFT replication protocol to provide the needed resilience and only have access to encrypted state.

Finally, we introduce three concrete service models for offering Intrusion-Tolerance as a Service (ITaaS) on top of existing cloud services. We enable an ITaaS provider to cost-effectively deploy such a service by designing a framework for optimizing the distribution of replicas of different applications across shared cloud resources. Overall, this approach has the potential to make intrusion-tolerant systems more accessible to system operators while maintaining the confidentiality of sensitive information.

Table of Contents

Preface . . . xii

1.0 Introduction . . . 1

1.1 Motivation and Problem Statement . . . 1

1.2 Dissertation Overview . . . 3

1.3 List of Contributions . . . 6

2.0 Background . . . 8

2.1 BFT Basics . . . 8

2.2 BFT and Network Attacks . . . 10

2.3 Separating Agreement from Execution . . . 12

2.4 Cloud-Based BFT and Confidentiality . . . 13

2.5 Scalability of Byzantine Fault Tolerant Applications . . . 14

3.0 System and Threat Models . . . 16

3.1 Basic System Model . . . 16

3.2 Basic Threat Model . . . 18

3.3 Service Properties . . . 20

4.0 Confidentiality in Partially Cloud-Based BFT System . . . 24

4.1 Overview . . . 24

4.2 System and Threat Model . . . 25

4.2.1 System Model . . . 25

4.2.2 Threat Model . . . 25

4.2.3 Service Properties . . . 26

4.3 Partially Cloud-Based BFT Architecture . . . 27

4.3.1 System Architecture . . . 30

4.3.2 Replica Distribution . . . 30

4.4 Protocols for Partially Cloud-Based BFT . . . 33

4.4.1 Introducing Client Updates . . . 34

4.4.2 Ordering Updates and Disseminating Results 36
4.4.3 Checkpoints and State Transfer 36
4.4.4 Key Renewal 38
4.5 Confidential Spire Implementation 41
4.5.1 Confidentiality-Preserving Intrusion Tolerant Middleware 42
4.5.2 Encryption and Decryption Details 42
4.5.3 Checkpointing and State Transfer Implementation 43
4.6 Evaluation 44
4.6.1 Performance Overhead of Confidentiality 44
4.6.2 Attack Evaluation of Confidential Spire 46
5.0 A Cloud-Based Hybrid Management Approach to Deploying Resilient Systems 49
5.1 Overview 49
5.2 Architecture Outline 50
5.2.1 Simplifying Interfaces via Threshold Signatures 51
5.2.2 Strengthening Confidentiality via Privacy Firewalls 52
5.3 Threat Model 53
5.3.1 Management Domain Failures: 54
5.3.2 Service Properties: 54
5.4 System Configuration 56
5.4.1 Calculating the Number of Required Cloud Replicas 58
5.5 Protocols for Decoupled Intrusion-Tolerant System 61
5.5.1 Introducing New Client Requests 61
5.5.1.1 Preventing Replay Attacks 63
5.5.1.2 Updating Encryption Keys 66
5.5.2 Ordering and Executing Client Requests 66
5.5.3 Batching of Client Requests 67
5.5.4 Checkpoints and Nearest-First Recovery 68
5.5.4.1 Checkpoint Creation 69
5.5.4.2 Nearest-First Recovery 69

5.6 Implementation . . . 71
5.6.1 Decoupled Spire Components . . . 71
5.6.2 Separating Agreement and Execution . . . 71
5.7 Evaluation . . . 72
5.7.1 Normal Operation Performance ($f = 1$) . . . 72
5.7.2 Increasing the Number of Tolerated Intrusions . . . 74
5.7.3 Performance during Failures and Recovery . . . 74
5.7.4 Discussion on Throughput . . . 76
5.7.5 Diversity Analysis . . . 76
6.0 Optimizing Deployment of Intrusion-Tolerance as a Service . . . 78
6.1 Overview . . . 78
6.2 Challenges of Optimizing Intrusion-Tolerance as a Service . . . 80
6.2.1 Limitation on Sharing Physical Machines . . . 80
6.2.2 New Challenges . . . 81
6.2.2.1 Calculation of Number of BFT Replicas . . . 81
6.2.2.2 Latency Constraint for BFT Replication Protocol . . . 81
6.2.2.3 Scheduling Proactive Recovery Across Applications . . . 82
6.3 Service Models . . . 83
6.3.1 Service Model 1 (SM1): Virtual-Machine-based ITaaS . . . 84
6.3.2 Service Model 2 (SM2): Dedicated-Server-based ITaaS . . . 86
6.3.3 Service Model 3 (SM3): Colocation-based ITaaS . . . 87
6.4 Strategies for Optimizing Replica Placement of Intrusion-Tolerant Applications across Cloud Resources . . . 89
6.4.1 Optimal Solutions using Mixed-Integer Linear Programming . . . 89
6.4.1.1 Service Model 1 (VM-based ITaaS) MILP formulation . . . 90
6.4.1.2 Service Model 2 (Dedicated-Server-based ITaaS) MILP formulation . . . 94
6.4.1.3 Service Model 3 (Colocation-based ITaaS) MILP formulation . . . 95
6.4.2 Heuristic Optimization Algorithms . . . 100
6.4.2.1 Calculation of Number of Sites and Replicas . . . 105

6.4.2.2 Strategies for Selecting Sites 105
6.4.2.3 Consideration of Maximum Latency Constraint 106
6.5 Experimental Setup and Implementation 107
6.5.1 Synthetic Application Data and Available Resources 107
6.5.2 ITaaS Optimizer, Placements Validator, and Quality Calculator 109
6.5.3 ITaaS Evaluator 110
6.6 Evaluation 111
6.6.1 Overall Feasibility 111
6.6.2 Efficiency for Large-Scale Deployments 115
6.6.3 Processing Time of Optimization Algorithms 120
6.6.4 Cost Analysis of Large-Scale Deployments 123
6.6.4.1 Service Model 1: AWS EC2 Instance Deployment 123
6.6.4.2 Service Model 2: AWS EC2 Dedicated Host Deployment . . . 125
6.6.4.3 Service Model 3: Colocation Deployment 127
7.0 Conclusion 129
7.1 Closing Statement 129
7.2 Lessons Learned 130
7.3 Future Work 131
Bibliography 133

List of Tables

1 Comparison of Threat Models 23

2 Configurations tolerating a proactive recovery, disconnected site, and 1-3 intrusions. 31

3 Spire and Confidential Spire normal operation performance for 36,000 Updates over 1 hour 45

4 Spire, Confidential Spire and Decoupled Spire normal operation performance for 36,000 updates over 1 hour 73

5 Spire, Confidential Spire and Decoupled Spire diversity analysis (number of diverse variants) 76

6 Cloud Optimization Algorithms 104

7 Experimental Setup Details for Generating Synthetic Application Data 109

List of Figures

1	Technology Contribution Map	3
2	Background map of relevant BFT topics	8
3	BFT system with 4 replicas: tolerates 1 intrusion	9
4	BFT system with 6 replicas: tolerates 1 intrusion and 1 proactive recovery	9
5	Network-Attack-Resilient Intrusion-Tolerant SCADA System [15]: tolerates 1 intrusion, 1 proactive recovery, and 1 site disconnection	11
6	Basic System Model	17
7	Supervisory Control and Data Acquisition (SCADA) System	18
8	*Adaptive Denial of Service Attack:* as soon as one on-premises site is able to reconnect, the other is targeted and disconnected	28
9	System architecture overview, showing 2 on-premises sites (each containing 4 replicas) and 2 cloud sites (each containing 3 replicas).	29
10	Partially Cloud-based BFT System with configuration "4+4+3+3": tolerates 1 intrusion, 1 proactive recovery, and 1 site disconnection	32
11	Client Request Flow in Partially Cloud-based BFT System	34
12	Latency of Confidential Spire in the presence of proactive recoveries and site disconnections based on configuration "4+4+3+3"	47
13	Cloud-based Hybrid Management Intuition	51
14	Decoupled Intrusion-Tolerant Architecture	57
15	Client Request Flow in Decoupled Intrusion-Tolerant System	62
16	Message Formats in Decoupled Intrusion-Tolerant System	65
17	Performance of Decoupled Spire During Attack Recovery	75
18	Service Model 1: Virtual-Machine-based ITaaS	84
19	Service Model 2: Dedicated-Server-based ITaaS	86
20	Service Model 3: Colocation-based ITaaS	88
21	ITaaS Experimental Setup	108

22 Service Model 1 Average Verification Score 112
23 Service Model 2 Average Verification Score 113
24 Service Model 3 Average Verification Score 114
25 Service Model 1 Average Number of Apps 115
26 Service Model 1 Average Number of Replicas 116
27 Service Model 2 Average Number of Apps 117
28 Service Model 2 Average Number of Machines 118
29 Service Model 3 Average Number of Apps 119
30 Service Model 3 Average Number of Machines 119
31 Service Model 1 Average Execution Time 121
32 Service Model 2 Average Execution Time 122
33 Service Model 3 Average Execution Time 122
34 Service Model 1 Average Monthly Total Cost 123
35 Service Model 1 Average Monthly Cost per App 124
36 Service Model 2 Average Monthly Total Cost 125
37 Service Model 2 Average Monthly Cost per App 126
38 Service Model 3 Average Monthly Total Cost 126
39 Service Model 3 Average Monthly Cost per App 127

Preface

I am overwhelmed with an immense sense of gratitude for the incredible journey that has led me to this moment. Completing this dissertation marks the culmination of years of dedication, passion, and perseverance, and I am profoundly grateful to all those who have supported me along the way.

First and foremost, I want to express my sincere gratitude to my advisor, Dr. Amy Babay. Dr. Amy's guidance, wisdom, and dedication have played a pivotal role in shaping both my academic journey and personal development. Through her encouragement, patience, and unwavering belief in my abilities, I have been motivated to surpass my own limitations and embrace new challenges. Dr. Amy's mentorship has not only enhanced the quality of my research but has also left a profound impact on various aspects of my life, for which I am deeply thankful.

I am also indebted to my esteemed committee members, Dr. Daniel Mossé, Dr. Adam Lee, and Dr. Balaji Palanisamy, for their invaluable insights, constructive feedback, and scholarly expertise. Their rigorous intellectual engagement and unwavering support have been a cornerstone of this endeavor, and I am deeply grateful for their contributions.

To my beloved wife, Maliha, words cannot express the depth of my gratitude for your unwavering love, encouragement, and understanding. Your unwavering support has been my rock throughout this journey, and I am endlessly grateful for your patience, sacrifice, and belief in me. You have been my source of strength and inspiration, and I am blessed to have you by my side.

Last but certainly not least, I extend my heartfelt thanks to my family and friends for their endless love, encouragement, and unwavering belief in my abilities. Your unwavering support has sustained me through the challenges and triumphs of this journey, and I am profoundly grateful for each and every one of you.

With great appreciation,

Maher Khan, PhD

1.0 Introduction

"Achieving reliability in the face of arbitrary malfunctioning is a difficult problem, and its solution seems to be inherently expensive." — Lamport et al. [44]

1.1 Motivation and Problem Statement

Cyberattacks on high-value systems continue to increase, with power grid, pipeline, and hospital systems (among others) experiencing high profile attacks [33, 52, 2]. In this hostile environment, *intrusion-tolerant* or Byzantine Fault Tolerant (BFT) replication can improve the attack resilience of such systems, allowing them to operate correctly even while partially compromised by an attacker [21]. In a BFT system, multiple *replicas* of the same application communicate with each other to reach a consensus on the correct state of the system, even in the presence of Byzantine (i.e., arbitrary or malicious) faults. However, despite a long history of research on BFT replication, such protocols have not been widely adopted in industry. There has been progress in making BFT replication easier to integrate into practical systems, with libraries such as UpRight [24] and BFT-SMaRt [16, 20], but this does not address the challenges of *deploying and managing* BFT-replicated systems.

A key barrier to deployment is that integrating an existing application with a BFT library is not enough to build a system that is resilient to sophisticated attacks. A practical intrusion-tolerant system must not only employ BFT agreement, but must also use proactive recovery to periodically refresh system replicas [21, 65], employ diversity to ensure replicas cannot be compromised by shared vulnerabilities [38, 55], and be deployed across multiple geographic sites to overcome sophisticated network attacks that can isolate a site [15]. Such a system becomes complex to manage and extremely expensive to build.

For critical infrastructure applications (e.g., power grid, pipeline, water treatment, and healthcare systems), it is unlikely that each system operator will be able to build such a system for themselves. Moreover, even if this was feasible, a system built by any single operator has inherent fragility in that the entire system is under a single management do-

main, and therefore subject to shared vulnerabilities and/or misconfigurations. In practice, a single-operator system is also likely to be more limited in its geographic span and physical redundancy due to the cost of building dedicated infrastructure. On top of these challenges, once the system is set up, maintaining and managing it over time requires specialized expertise to reason about the underlying BFT replication protocols.

Commercially available cloud infrastructure offers a promising solution to these challenges. Cloud service providers invest in building highly resilient data center infrastructure, since costs are amortized over many applications. Existing cloud service providers typically manage widely geographically distributed data centers, with each data center having connectivity from multiple Internet Service Providers (ISPs). In addition, cloud service providers typically have expertise in designing and operating fault-tolerant distributed systems.

A tempting solution to making high-value systems intrusion tolerant is to host them in the cloud, allowing the cloud service provider to fully manage the infrastructure needed for resilience. However, for many applications, this is not an option. First, for certain applications (e.g., critical infrastructure like the power grid), system operators are unwilling to store potentially sensitive data in the cloud. Furthermore, some systems may need specialized hardware or client communication infrastructure (e.g., clients that do not communicate over IP) that the cloud infrastructure does not support. Finally, even if fully hosting the application on cloud infrastructure is feasible, there is still a question of who is responsible for system administration (e.g., performing software updates, managing OS configurations, etc.). If we consider a traditional application that is replicated by linking with a BFT library, there are two possibilities: (1) the system operator is responsible, and the cloud is simply used for providing the physical infrastructure, or (2) the cloud service provider is responsible and offers a fully managed service. However, neither of these options is satisfactory. The first requires that the system operator has a high level of expertise in deploying BFT-replicated systems, while the second requires the cloud service provider to have deep knowledge of the applications they support. The core challenge is that experts in a specific application domain are unlikely to be experts in designing and deploying intrusion-tolerant systems, and conversely, experts in intrusion-tolerant systems are unlikely to be experts in specific application domains.

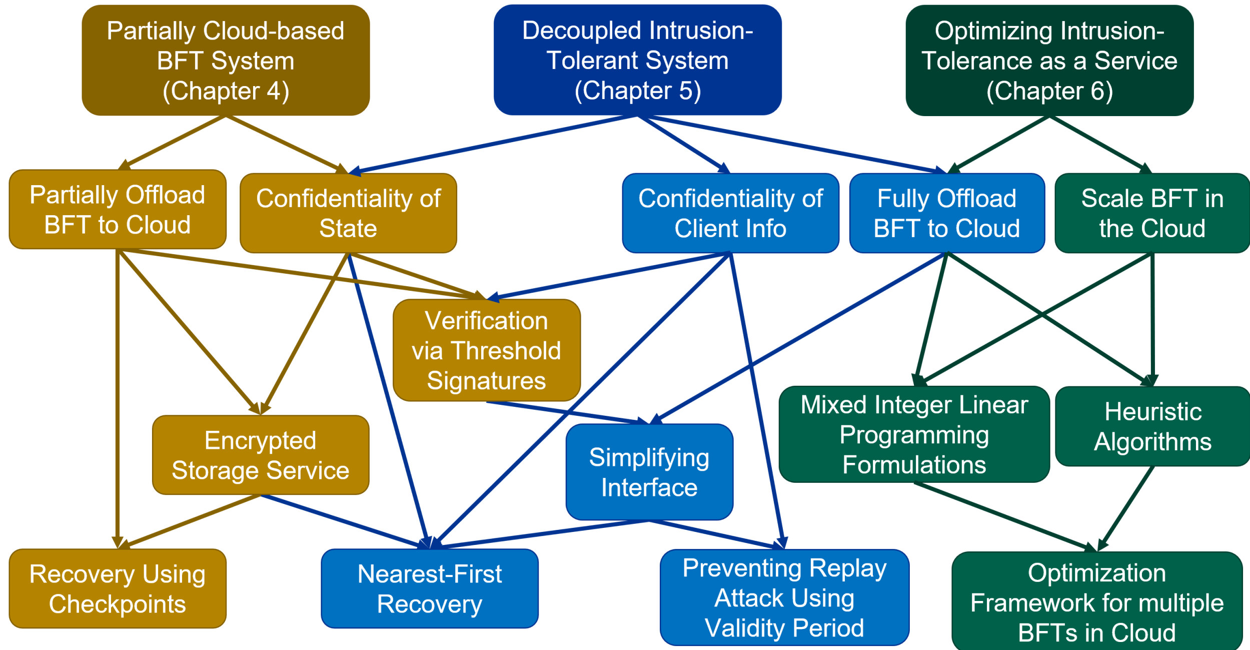


Figure 1: Technology Contribution Map

1.2 Dissertation Overview

In this dissertation, our main contribution lies in introducing a pioneering *hybrid management* approach to deploying intrusion-tolerant systems. We develop innovative systems that enable system operators to deploy intrusion-tolerant applications by partially or fully outsourcing the responsibility of the BFT replication protocol to a cloud service, all while maintaining the privacy of the application's state and algorithms. By embracing our novel concept of *intrusion-tolerance as a service*, wherein specialized cloud service providers manage the intricacies of intrusion-tolerant systems, individual system operators can fortify their high-value systems against cyber threats. We simplify the deployment and management processes of intrusion-tolerant systems for system operators, marking a notable step forward in cybersecurity research. Our key technological contributions are visualized in Figure 1.

To address the issue of storing sensitive data in the cloud, in Chapter 4, we present a Partially Cloud-based BFT System, which is a new model for BFT systems that moves toward offering "intrusion-tolerance as a service". Under this model, application logic and data are only exposed to servers hosted on the system operator's premises. However, the

intrusion-tolerant system architecture can be designed by a cloud service provider, and additional offsite servers can be hosted in cloud sites managed by the cloud service provider to provide the needed resilience to system compromises and network attacks. These offsite servers participate in the BFT replication protocol, but do not execute application logic and only store encrypted state and requests. In Chapter 4, we show that our Partially Cloud-based BFT System is able to provide the same resilience to system compromises and network attacks as the state-of-the-art [15], *without requiring application state and logic to be exposed to cloud replicas*.

Deploying and managing BFT-replicated systems in practice requires both specialized technical expertise and substantial investment in additional physical infrastructure. To separate the responsibilities between system operators (who are experts in their application domains) and cloud service provider (who specializes in the deployment of intrusion-tolerant systems), in Chapter 5, we introduce a Decoupled Intrusion-Tolerant System with a *hybrid management model* that allows for specialization. System operators manage their applications, but management of the intrusion-tolerant replication service and the additional physical infrastructure needed to support it is done by the cloud service provider. This model aims to make intrusion-tolerance *accessible*, by allowing system operators to essentially purchase intrusion-tolerant-replication-as-a-service. It also enables a level of resilience that was not previously possible by introducing *management diversity*, which enables resilience to a new class of failure that affects an entire management domain.

Because prior work has assumed that the entire system operates under a single administrative domain, it has not needed to consider how to divide responsibilities between a system operator and a cloud service provider, or how to define the interfaces between them. We address this in Chapter 5 with our Decoupled Intrusion-Tolerant System: we define an intrusion-tolerant ordering and encrypted state storage service that can support intrusion-tolerant, network-attack-resilient, and management-domain-failure-recoverable applications, while keeping deployment as simple as possible for the system operator. In Chapter 5, we show that our Decoupled Intrusion-Tolerant System meets application performance requirements in terms of request latency.

Overall, the Decoupled Intrusion-Tolerant System in Chapter 5 requires *more* resources

than the Partially Cloud-based BFT System in Chapter 4. However, a system operator using the Decoupled Intrusion-Tolerant System needs to deploy and manage *fewer* resources, while the rest, including the BFT replication engine, are managed by the cloud service provider. If a system operator would prefer to implement and manage their own intrusion-tolerant system, including the BFT replication engine, while still utilizing cloud sites to host additional replicas, then the Partially Cloud-based BFT System will require fewer total resources (thereby, costing less) than the Decoupled Intrusion-Tolerant System. Additionally, the Partially Cloud-based BFT System also supports lower latency than the Decoupled Intrusion-Tolerant System when tolerating a similar threat model.

The architecture of our Decoupled Intrusion-Tolerant System relies on a cloud service provider offering an intrusion-tolerant replication service. Hence, in Chapter 6, we explore how an entity can deploy *Intrusion-Tolerance as a Service (ITaaS)*. We envision an *ITaaS provider* to be a new entity that builds our ITaaS, including the BFT Replication Engine, on top of the cloud resources provided by an *infrastructure provider*, who offers the necessary cloud resources as a service (e.g. Amazon Web Services offers Elastic Compute Cloud [63] and Equinix offers colocation in their data centers [36]). We make the deployment of our solution easier and cost-effective for ITaaS provider by designing a framework for optimizing the distribution of replicas of different applications across shared cloud resources, while guaranteeing safety, liveness, and supporting proactive recovery. We develop heuristic optimization algorithms and Mixed-Integer Linear Programming (MILP) formulations for three separate service models, where each service model is based on the type of cloud resources being used (i.e., virtual machines, dedicated servers, or colocated servers), and the level of trust between the ITaaS provider and the infrastructure provider. Evaluating these algorithms reveals their effectiveness in maximizing the number of applications deployed, while minimizing cloud resource usage and overall costs, all while meeting application requirements and constraints unique to ITaaS.

1.3 List of Contributions

The overall contributions of this dissertation are:

- We design a Partially Cloud-based BFT System, which is the first BFT system that can leverage offsite cloud sites to achieve resilience to simultaneous network attacks and system compromises, without requiring confidential state or algorithms to be exposed to cloud servers. This allows for improved security and availability of critical applications without weakening their confidentiality.
- We extend the basic Partially Cloud-based BFT System to provide well-defined confidentiality guarantees in the case that an on-premises server is compromised (i.e. an attacker gains access to a system operator's on-premises server).
- We implement and evaluate the Partially Cloud-based BFT System in the context of SCADA for the power grid. This is a critical application area, as SCADA systems are responsible for monitoring and controlling the power grid, and are therefore subject to a wide range of security threats. We show that in our system the performance overhead of providing confidentiality is acceptable, and the system can meet the latency requirements of power grid SCADA.
- We define a hybrid management model for intrusion-tolerant systems that enables system operators to leverage intrusion-tolerant ordering and encrypted storage services from a cloud service provider. This is an innovative approach to intrusion-tolerant systems, as it allows system operators to retain control over their applications while leveraging cloud-based services.
- We design a Decoupled Intrusion-Tolerant System, which is the first system architecture that enables system operators to deploy intrusion-tolerant applications while completely offloading the BFT replication protocol to the cloud, preserving confidentiality, and providing resilience to a broad threat model.
- We show that the Decoupled Intrusion-Tolerant System architecture can provide resilience to a broad threat model that includes intrusions and network attacks and is able to recover from management domain failures that affect all replicas hosted by the sys-

tem operator (on-premises). This provides confidence in the system's ability to provide robustness and security in real-world deployments.

- We implement and evaluate the Decoupled Intrusion-Tolerant System architecture in the context of an industrial control application, showing that while it increases latency by about 9ms (18%) compared to a fully system-operator-managed BFT system, the request latency meets application performance requirements. This demonstrates the practicality of the system in a real-world application, and provides evidence that it can be deployed in a way that meets the needs of system operators.
- We show how an entity can offer *Intrusion-Tolerance as a Service (ITaaS)* by building on top of cloud resources provided by an infrastructure provider. We define three service models based on the type of cloud resources being used, and the level of trust between the ITaaS provider and the infrastructure provider.
- We design and implement a framework for optimizing the distribution of replicas of different applications across shared cloud resources, while guaranteeing safety, liveness, and supporting proactive recovery. We develop heuristic algorithms, and Mixed-Integer Linear Programming (MILP) formulations for this framework.
- We evaluate our optimization framework in terms of feasibility, efficiency, performance and cost analysis. We show that optimal solutions and select heuristic algorithms consistently exhibit high effectiveness in minimizing cloud resource usage and overall costs, particularly in scenarios involving a large number of applications, all while meeting the necessary application requirements.

In summary, we present innovative solutions for simplifying the deployment of intrusion-tolerant systems, crucial for securing high-value critical applications against cyber threats. We address the challenges of integrating Byzantine Fault Tolerant (BFT) replication into practical systems by leveraging cloud resources effectively. Through the introduction of the Partially Cloud-based BFT System and the Decoupled Intrusion-Tolerant System, we offer flexible deployment models catering to diverse application needs and system operator expertise levels. Additionally, the exploration of Intrusion-Tolerance as a Service (ITaaS) offers a new approach to making resilience accessible while optimizing resource utilization and cost-effectiveness.

2.0 Background

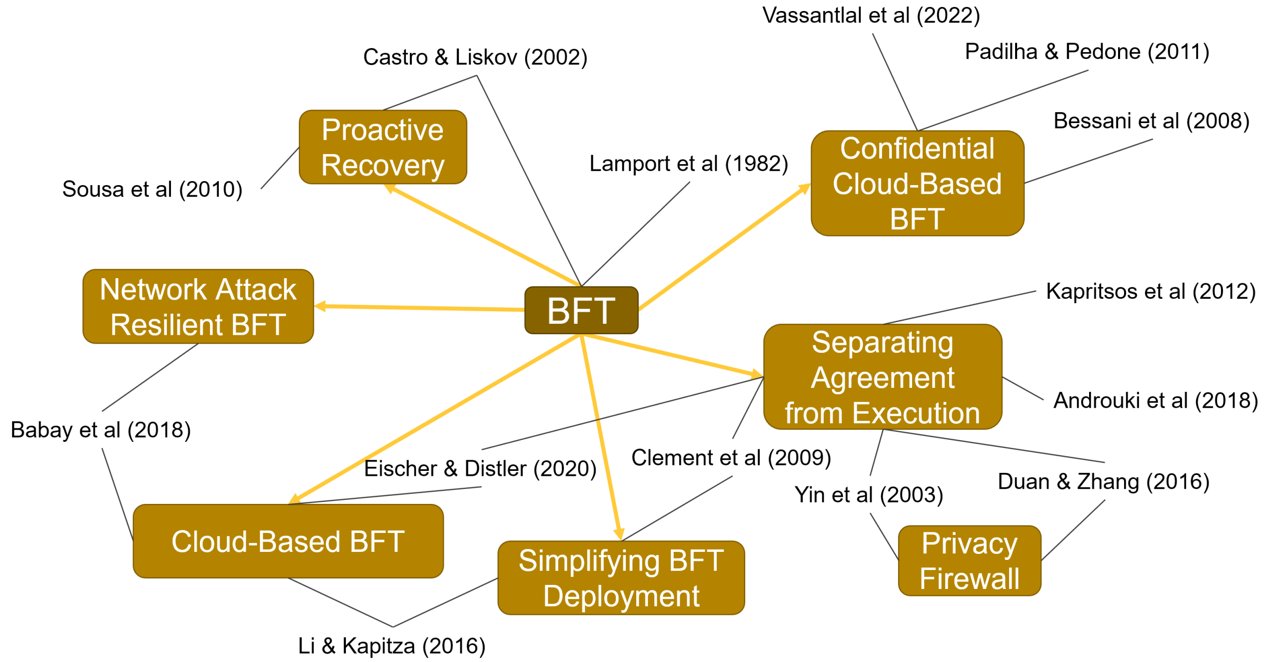


Figure 2: Background map of relevant BFT topics

2.1 BFT Basics

The concept of Byzantine Fault Tolerance (BFT) originated from the Byzantine Generals Problem [44], which is a metaphor that describes the difficulty of achieving consensus in a distributed system where components may fail arbitrarily. Byzantine Fault Tolerant (BFT) state machine replication is a well-known technique to provide intrusion-tolerance, enabling a system to guarantee safety (correctness and consistency of the system state) and liveness (progress in processing updates) even if up to some threshold number of replicas are compromised (e.g. [21]). State machine replication involves replicating the execution of a state machine across multiple nodes, ensuring that all replicas reach the same state in the same order by applying the same sequence of inputs. In BFT state machine replication, replicas

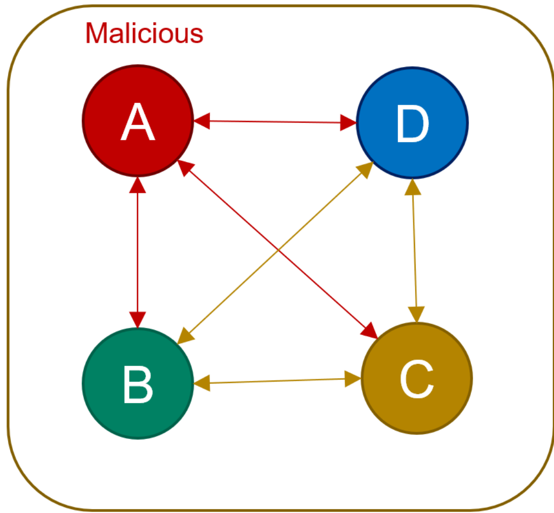


Figure 3: BFT system with 4 replicas: tolerates 1 intrusion

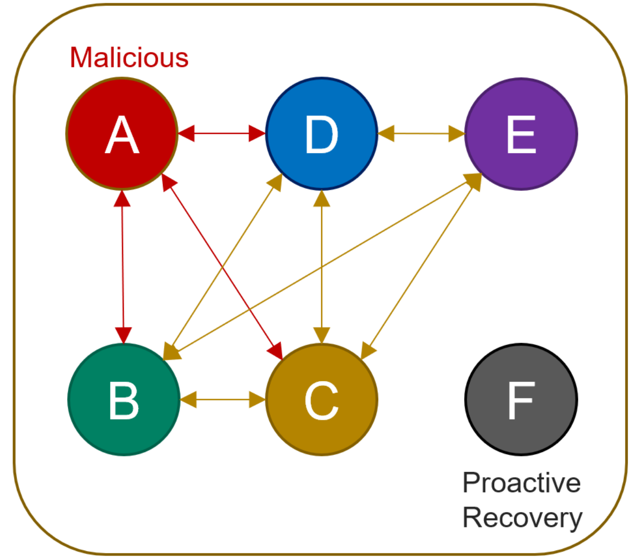


Figure 4: BFT system with 6 replicas: tolerates 1 intrusion and 1 proactive recovery

communicate with each other to reach a consensus on the correct state of the system, even in the presence of Byzantine faults, which include arbitrary behaviors such as sending incorrect messages or omitting messages altogether. The number of tolerated compromises is most often f out of $3f + 1$ total replicas [21], although some systems can tolerate f out of $2f + 1$ total replicas with additional assumptions or trusted hardware [27, 23, 28, 73].

To prevent an attacker from simultaneously compromising more than one replica using the *same* attack procedure, replicas must employ *diversity*. Diversity helps by increasing the complexity for attackers, making it harder for them to find vulnerabilities that apply uniformly across all replicas. This can include strategies such as N-version programming [13, 43], OS diversity [37], compile-time diversification [54], and/or use of different physical hardware and Internet Service Providers (ISPs). Figure 3 shows a BFT system with 4 replicas capable of tolerating 1 replica being compromised. Some systems additionally guarantee *performance* under attack, rather than only liveness [9, 25, 72, 49].

To support long system lifetimes, it is necessary to employ *proactive recovery*, which allows replicas to be periodically taken down and restored to a known clean state [21, 59]. Providing continuous availability with proactive recovery typically requires $3f + 2k + 1$ total replicas to simultaneously tolerate up to f compromised replicas and k recovering

replicas [65]. The required number of replicas increases compared to a BFT system without proactive recovery because if the sum of compromised replicas and recovering replicas exceeds f, the system risks failing to progress without additional replicas. Figure 4 shows a BFT system with 6 replicas capable of tolerating simultaneously 1 replica being compromised and 1 replica going through proactive recovery.

2.2 BFT and Network Attacks

One motivation for our work comes from the recent Network-Attack-Resilient Intrusion-Tolerant SCADA System for the power grid [15, 69], which showed that at least three geographically distributed sites are needed to withstand sophisticated network attacks that can target and isolate a site from the rest of the network. The intuition for this is the following: since BFT replication protocols require more than half of the system replicas (in fact, $2f+k+1$ out of $3f+2k+1$) to be up, correct, and connected in order to make progress, any system that distributes replicas across fewer than three sites can be prevented from making progress by isolating a single site. Clearly, if all replicas are located in a single site, a denial of service attack targeting that site can prevent it from communicating with remote clients and thus render it unable to receive and process their updates. If replicas are split across only two sites, targeting the larger of the two sites will disconnect a majority of the system replicas, leaving the rest unable to make progress without them.

Figure 5 shows a Network-Attack-Resilient Intrusion-Tolerant SCADA System with 12 replicas distributed evenly across 4 geographically distributed sites capable of tolerating 1 compromised replica, 1 replica going through proactive recovery, and 1 site being disconnected from the rest of the network. Due to the high expense of constructing additional control centers with full capabilities for communicating with Remote Terminal Units (RTUs) and Programmable Logic Controllers (PLCs), and controlling power grid equipment, the Network-Attack-Resilient Intrusion-Tolerant SCADA System introduced an architecture that uses two power grid control centers (which typically exist today for fault tolerance purposes) and supplements them with additional data center sites that do not need to com-

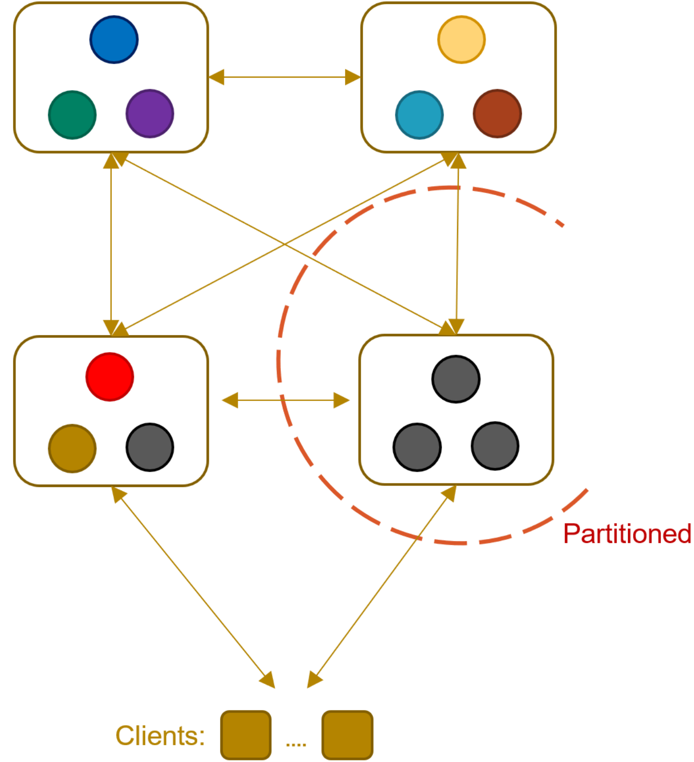


Figure 5: Network-Attack-Resilient Intrusion-Tolerant SCADA System [15]: tolerates 1 intrusion, 1 proactive recovery, and 1 site disconnection

municate with RTUs and PLCs. The use of data centers can also reduce the management overhead of the higher number of replicas that Spire needs to support its threat model (12 total replicas to support $f = 1$ and 1 disconnected site).

However, data center replicas are still required to maintain a full copy of the system state and execute application logic to process incoming updates. This raises confidentiality concerns, as it requires SCADA operators to expose their private system state and algorithms to offsite replicas potentially hosted by a third party. Today, if a system operator wants to avoid trusting a third party with this information, they must take on the responsibility for managing the full deployment (and constructing their own additional sites to host system replicas).

In this dissertation, we address this issue through a new hybrid model for intrusion-tolerant systems: system operators host and manage "on-premises" replicas distributed across one or more geographic sites that they manage and control, while a cloud service

provider hosts and manages additional replicas located in cloud sites. In our model, not only do cloud-service-provider-managed replicas not need to communicate with clients, but they only see encrypted state and do not execute application logic.

2.3 Separating Agreement from Execution

Separating agreement from execution [75] is integral to our overarching goal of simplifying the deployment and management of intrusion-tolerant systems. Instead of using a single set of replicas that both order and execute requests, replicas are divided into two clusters: an *agreement cluster*, which runs the BFT replication protocol to totally order client requests, and an *execution cluster*, which executes requests in the order determined by the agreement cluster and generates client responses [75]. This separation also makes it possible to prevent compromised replicas from leaking confidential data by inserting a *privacy firewall* between the execution and agreement clusters [75, 34]. The privacy firewall can filter messages sent by the execution replicas to prevent confidential data from being sent in client replies.

In this work, by separating the ordering of client requests (agreement) from executing these requests and generating client responses (execution), we introduce a level of modularity and flexibility into the design of our intrusion-tolerant systems. Our Decoupled Intrusion-Tolerant System in Chapter 5 builds on the separation of agreement and execution but introduces the separation of *management*. There are other works that also separate agreement from execution (e.g. HyperLedger Fabric [11], UpRight [24], Spider [35], and Eve [41]), and some recent BFT protocols that separate dissemination of requests from ordering [42, 30, 67], but these all assume a single management domain. In our Decoupled Intrusion-Tolerant System, not only are the agreement and execution clusters run on physically separate sets of hosts, but they are managed by different entities and can be in different geographic locations.

The separation of management introduces strict privacy concerns: in many cases, system operators are unlikely to be willing to expose confidential data, algorithms, or client information to the cloud service provider managing the replication service. The work in [75] and [34] partially addresses privacy concerns by encrypting client requests and replies, and

using a privacy firewall to filter messages sent by the execution cluster. However, in those works, only the data itself is considered confidential. Client identities, locations, and request patterns are not considered to be confidential. In fact, the clients communicate directly with the agreement cluster. In our Decoupled Intrusion-Tolerant System, the agreement cluster runs in the cloud and should not have any access to clients, or information about them.

2.4 Cloud-Based BFT and Confidentiality

Prior work has deployed BFT systems partially or fully in the cloud to reduce costs and/or to simplify deployment. However, none of the existing works fully meet our aims of (1) providing intrusion-tolerance for arbitrary state machine replication applications (2) partially or fully offloading the BFT replication protocol to the cloud, and (3) enforcing confidentiality of application state and proprietary algorithms.

Prior work has investigated cloud-based BFT replication from the perspective of ease of deployment (e.g. BFT-Dep [45]) and performance (e.g. Spider [35]), but these works have assumed the system operator is not concerned with exposing application state and logic to the cloud and will run the entire application in the cloud.

Other work has considered the confidentiality implications of cloud-based BFT systems, developing secret-sharing approaches to maintain confidentiality for BFT-replicated applications in the cloud (e.g. DepSpace[18], Belisarius [53], and COBRA [71]). These systems use $(f + 1, n)$–threshold schemes, in which data is encoded into n shares, and each share is stored in a separate replica. Since $f + 1$ shares are needed to reconstruct the original data, they guarantee confidentiality as long as no more than f replicas are compromised. This approach has been used to build BFT-replicated storage systems, such as DepSky [17], SCFS [19], and RockFS [48].

While these systems offload BFT replication to the cloud and support data confidentiality, they do not support arbitrary state machine replication applications. Practical secret-sharing schemes today support limited operations such as key-value storage [71], tuple space operations [18], or addition on stored values [53]. Secure multiparty computation or homo-

morphic encryption could enable general operations on encrypted data, but these approaches are computationally expensive and, more fundamentally, do not keep the application logic confidential (only the data it operates on). Recent approaches using Trusted Execution Environments (TEEs) to provide confidential computing in the cloud via encrypted VMs [60] could keep both data and logic confidential. However, we would like to minimize assumptions of trusted hardware, and this approach does not resolve the question of who is responsible for managing the encrypted VMs in the cloud. Alternatively, we could use a BFT storage solution to replicate encrypted state and consider the application as a client of the storage system, but in that case the application itself is not intrusion-tolerant.

2.5 Scalability of Byzantine Fault Tolerant Applications

A cloud service provider's focus is to minimize cost while maintaining the required performance and service level agreements (SLAs). Research such as [64] explore ways to optimize the cost for cloud service provider while minimizing SLA violations. In comparison, our work focuses specifically on the scalability and cost optimization of virtual or physical machines running Byzantine Fault Tolerant (BFT) systems.

Authors in [50, 74] acknowledge that running multiple replicas of the same BFT application on the same physical machine is not possible due to shared vulnerabilities. Although the same authors [50, 74] show how to run multiple BFT applications on the same physical machines (e.g., by running each replica inside a separate VM, and allocating multiple VMs from separate BFT applications on the same physical machine), they naively distribute replicas of the same BFT applications across physical machines without considering different application requirements or optimization of given physical resources.

Although there are numerous scheduling algorithms [39, 3], none of them are feasible for scheduling multiple BFT applications on the same physical machines. The primary distinction arises from the nature of these BFT replicas, which run continuously (except when temporarily halted for proactive recovery), unlike jobs or tasks that eventually terminate. Thus, algorithms designed for scheduling jobs/tasks in distributed clusters are not well suited

for our problem. However, there are scheduling algorithms for optimizing the placement for continuously running processes, such as web servers [58], but they do not address the three new challenges for our BFT systems: calculation of number of BFT replicas (Section 6.2.2.1), latency constraint for replication protocol (Section 6.2.2.2), and scheduling proactive recovery across BFT applications (Section 6.2.2.3).

In Chapter 6, we design an optimization framework, with heuristic algorithms and Mixed Integer Linear Programming (MILP) formulations, for optimizing the distribution of replicas of different BFT applications across shared cloud resources, while guaranteeing safety, liveness, and supporting proactive recovery. We evaluate these algorithms in terms of feasibility, efficiency and cost, and show that the optimal solutions and certain heuristic algorithms reveal their effectiveness in maximizing the number of applications deployed, while minimizing cloud resource usage and overall costs, all while meeting application requirements and constraints unique to BFT applications.

3.0 System and Threat Models

In this chapter, we introduce our novel hybrid-managed system model and then present an overview of our threat model, and the service properties our systems provide.

3.1 Basic System Model

To support intrusion-tolerance as a service, we introduce a new system model in which system management is shared between *system operators* and *cloud service providers*, as shown in Figure 6. System operators are responsible for managing their application and are typically experts in the application domain. Cloud service providers manage cloud resources that help support intrusion-tolerance, while reducing the amount of infrastructure the system operator needs to build and manage. At a minimum, the cloud service provider offers data center hosting capabilities to reduce the number of physical sites the system operator needs to manage (see Chapter 4). However, the cloud service provider may offer additional services that further simplify deployment for system operators (see Chapter 5).

In our model, a system is physically deployed across locations owned and operated by the system operator (*on-premises sites*) and locations operated by the cloud service provider (*cloud sites*). Both on-premises sites and cloud sites may host system replicas. We refer to replicas located in on-premises sites as *on-premises replicas* and replicas located in cloud sites as *cloud replicas*.

We consider clients and on-premises replicas to be part of the same *on-premises domain*, and clients communicate only with the on-premises replicas. This is a good fit for many industrial control or enterprise applications where the on-premises replicas and the clients are managed by the same entity. System operators may have privacy concerns regarding allowing clients to communicate with entities outside the on-premises sites, or clients may face feasibility constraints preventing communication beyond the on-premises sites. For example, in an industrial control context, on-premises replicas may be replicas of a Supervisory Control

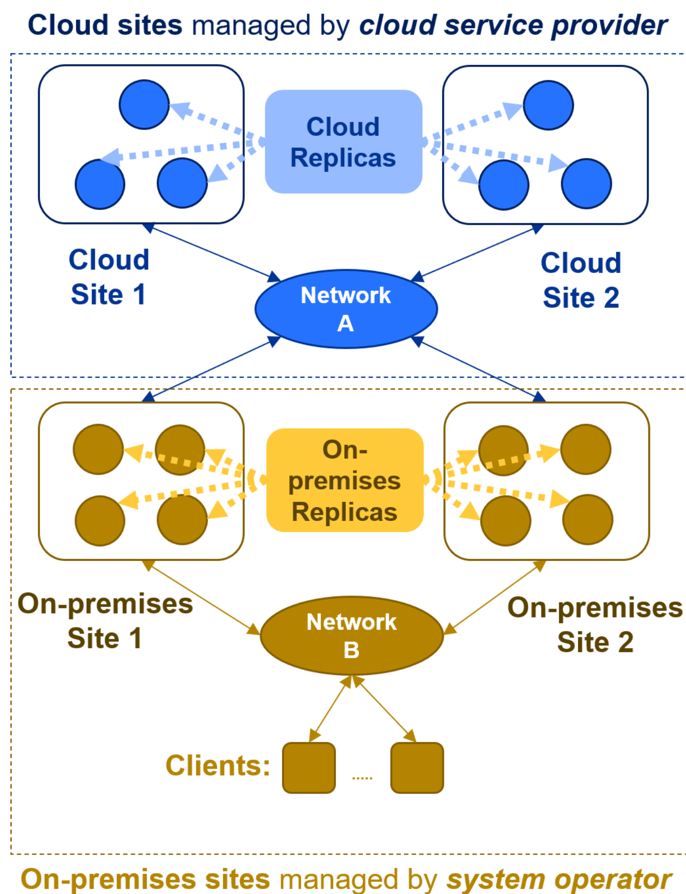


Figure 6: Basic System Model

and Data Acquisition (SCADA) server (Figure 7), and clients may be Remote Terminal Units or Programmable Logic Controllers that send data to and receive commands from the SCADA system. Or, we could consider replicas of an Electronic Health Record (EHR) database, and clients that are authorized devices or users accessing the service via a VPN. Clients authenticate themselves to the system by signing their requests using private keys; the corresponding public keys are known by the on-premises replicas.

The cloud replicas represent the *cloud domain*, managed by the cloud service provider. For certain applications (e.g., critical infrastructure like the power grid), system operators are unwilling to store potentially sensitive data in the cloud. Therefore, to preserve confidentiality of on-premises domain's application state and proprietary algorithms, we do not expose them to the cloud domain.

Due to the same privacy concerns, only on-premises replicas *execute* client requests and

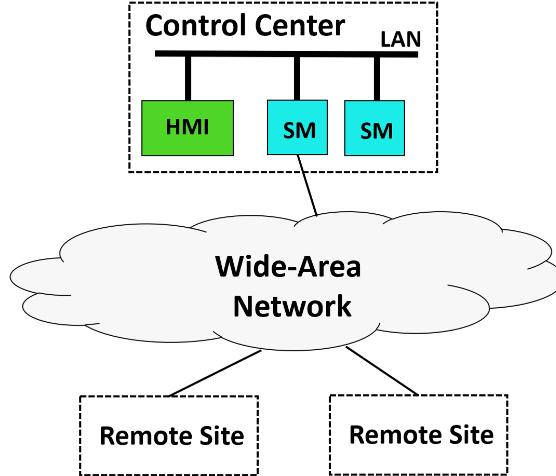


Figure 7: Supervisory Control and Data Acquisition (SCADA) System

maintain *active* system state. On-premises replicas *encrypt* (with encryption/decryption keys only available to on-premises replicas) client requests before sharing them with cloud replicas. Cloud replicas participate in total ordering of *encrypted* client requests using an intrusion-tolerant BFT replication protocol (on-premises replicas also participate in total ordering, but we limit this in Chapter 5 to simplify management). In addition to performing ordering, the cloud replicas store each ordered encrypted request. They periodically garbage collect these requests, replacing them with encrypted state checkpoints from the on-premises replicas. This enables on-premises replicas to recover their state entirely from the cloud, making it possible to tolerate sophisticated network attacks (see details in Section 4.3), as well as *management domain failures* in which *all* on-premises replicas lose their state (see details in Section 5.3).

3.2 Basic Threat Model

Our threat model builds on the work in [15], which considered both system-level compromises of the server replicas and network-level attacks that aim to disrupt communication among replicas and/or between replicas and clients. This model includes a broad range of network attacks, but, as in [15], we reduce this to a simpler model through the use of an

intrusion-tolerant overlay network to connect the sites to one another [51, 68]. The intrusion-tolerant network defends against extensive network disruption, combats malicious routing attacks, and significantly enhances the effort and resources needed to carry out a successful denial-of-service attack. With the use of the intrusion-tolerant network, the network attacks we still need to address are reduced to sophisticated (resource-intensive) denial of service attacks that can target and isolate a geographic site. We assume that at any time, up to a tolerated number of sites may be subject to such an attack and thus disconnected from the rest of the network.

As in other intrusion-tolerant replicated systems (e.g., [21]), we assume that up to a threshold number of replicas may be compromised (e.g., we assume f number of tolerated compromised replicas in Chapter 4). Compromised replicas may behave arbitrarily and collude with one another. As in prior work, we employ proactive recovery to allow the system to tolerate up to the threshold number of compromises within a limited time window, as opposed to over the entire system lifetime. Proactively recovering replicas become unavailable during the recovery process (the replica is taken down and restored to a known good state) [21, 59]. Thus, at any time our threat model includes up to a threshold number of compromised replicas, up to a threshold number of replicas that are unavailable because they are going through proactive recovery, and up to a threshold number of disconnected (or otherwise unavailable) sites.

In Chapter 5, we introduce the new concept of a *management domain failure*. This scenario involves the loss of state for *all* on-premises replicas managed by a system operator. Management domain failure can address practical threats like ransomware. If on-premises replicas are encrypted in a ransomware attack, they can be shut down, cleaned, and restarted. The latest state can then be restored from cloud replicas, allowing seamless operation continuity, albeit with a temporary outage. Our Decoupled Intrusion-Tolerant System in Chapter 5 can recover from a management domain failure (more details in Section 5.3.1).

To support the assumption that the attacker does not exceed the tolerated threshold number of system compromises, the system must employ diversity (e.g. using N-version programming [13, 43], OS diversity [37], compile-time diversification [54], and/or different hardware and ISPs). We assume each replica has access to a hardware-protected private key

(e.g. using the TPM) that it can use for signing, but that cannot be deleted, modified, or exfiltrated from the machine. Finally, we assume an attacker cannot break cryptographic protocols.

3.3 Service Properties

Our systems are designed to provide three types of guarantees: Safety, Liveness / Bounded Delay, and Confidentiality. Our liveness guarantee is same as the one specified in [21], and our safety and bounded delay guarantees are essentially the same as those specified in [15], although we adapt them to a generic replicated system; where [15] specifically considered SCADA Masters, HMIs, RTUs, and PLCs, we state our guarantees in terms of generic servers and clients. Extending those guarantees to additionally support confidentiality is a novel contribution of this work.

Definition 1 (Safety)**.** *If two correct on-premises replicas execute the ith update, then those updates are identical, and the state resulting from the execution of that update at the two on-premises replicas is also identical.*

Our Partially Cloud-based BFT System (Chapter 4) guarantees safety as long as no more than f replicas are simultaneously compromised. For the Decoupled Intrusion-Tolerant System (Chapter 5), safety is guaranteed as long as: (1) no more than f_o on-premises replicas in each on-premises site are compromised simultaneously, and (2) no more than f_c cloud replicas are compromised simultaneously. Furthermore, safety is guaranteed in our Decoupled Intrusion-Tolerant System even when all on-premises replicas under the control of a system operator lose their state (i.e., *management domain failure*), and this system can recover the latest state after such a failure as long as the conditions under which safety is guaranteed are not violated.

Definition 2 (Liveness)**.** *Clients eventually receive replies to their requests [21].*

Our Partially Cloud-based BFT System guarantees liveness, as well as Bounded Delay, which is a stronger guarantee than just liveness (see Definition 3 below). To guarantee

liveness for the Decoupled Intrusion-Tolerant System (Chapter 5), we require that: there are at most f_o compromised on-premises replicas per on-premises site and f_c compromised cloud replicas, at most k_o on-premises replicas per on-premises site and k_c cloud replicas performing proactive recovery, and at most d_o on-premises sites and d_c cloud sites are disconnected from the network.

Definition 3 (Bounded Delay)**.** *The latency for an update introduced by a correct authorized client to be executed by at least $f + 1$ correct on-premises replicas (and thus have its effects made visible) is upper bounded.*

Bounded Delay essentially gives a stronger guarantee than just liveness, and it was introduced in [10]. To guarantee bounded delay in our Partially Cloud-based BFT System (Chapter 4), we require that the conditions of our threat model are met: at most f replicas are compromised, at most one replica is undergoing proactive recovery, and at most one site is downed or disconnected due to network attack. In addition, the remaining replicas (i.e. all correct, non-recovering replicas located outside the disconnected site) must be able to communicate with one another, and the remaining correct *on-premises* replicas must be able to communicate with clients. Finally, communication among the remaining correct replicas must meet the network stability requirements of Prime [10], which is used as our underlying agreement protocol and requires that the latency variance between each pair of correct servers is bounded (see [15] for additional discussion). Note that, our Decoupled Intrusion-Tolerant System (Chapter 5) can also guarantee Bounded Delay when Prime [10] is used as the underlying BFT replication engine.

Definition 4 (Complete Confidentiality)**.** *System state and state manipulation algorithms remain confidential (known only to on-premises replicas).*

Our Partially Cloud-based BFT System provides this guarantee as long as no *on-premises replica* is compromised. An unlimited number of cloud replicas may be compromised without violating confidentiality. Similar to the Partially Cloud-based System, our Decoupled Intrusion-Tolerant System also guarantees complete confidentiality, even if an unlimited number of cloud replicas are compromised. However, unlike the Partially Cloud-based System, in the Decoupled Intrusion-Tolerant System, we maintain confidentiality even when up

to f_o on-premises replicas per on-premises site are compromised, provided that the privacy firewalls (see details in Section 5.2.2) are operational and correctly configured.

We summarize our overall threat models for the Partially Cloud-based BFT System, Decoupled Intrusion-Tolerant System, and compare them to the prior work that introduced the Network-Attack-Resilient Intrusion-Tolerant SCADA System [15] in Table 1.

Table 1: Comparison of Threat Models

	Network-Attack-Resilient Intrusion-Tolerant SCADA System [15]	Partially Cloud-based BFT System (Chapter 4)	Decoupled Intrusion Tolerant System (Chapter 5)
Number of Intrusions Tolerated	f	f	f_o on-premises f_c cloud
Number of Simultaneous Proactive Recoveries	1	1	k_o on-premises k_c cloud
Number of Site Disconnections Tolerated	1	1	d_o on-premises d_c cloud
Tolerates Management Domain Failure	No	No	Yes, if no more than f_c cloud intrusions
Safety Guarantee	Met if no more than f intrusions	Met if no more than f intrusions	Met if no more than f_o and f_c intrusions
Liveness Guarantee	Met if no more than f intrusions, 1 recovery, and 1 site disconnection	Met if no more than f intrusions, 1 recovery, and 1 site disconnection	Met if no more than f_o intrusions, k_o recoveries, d_o site disconnections, f_c intrusions, k_c recoveries, and d_c site disconnections
Bounded Delay Guarantee	Yes	Yes	Depends on underlying BFT protocol
Confidentiality Guarantee	No	Met if no on-premises intrusion	Met if no more than f_o on-premises intrusions

4.0 Confidentiality in Partially Cloud-Based BFT System

4.1 Overview

In this chapter, we present a Partially Cloud-based BFT System, which is a new Byzantine Fault Tolerant replicated system that moves toward "intrusion tolerance as a service". In this system, application logic and data are only exposed to servers (on-premises replicas) hosted on the system operator's on-premises sites. Additional offsite servers (cloud replicas) hosted in cloud sites can support the needed resilience without executing application logic or accessing confidential state.

To make this system work, there are a few challenges: (1) the offsite servers need to ensure that each request submitted to the BFT replication protocol is a valid request submitted by an authorized client without decrypting the request (since they do not have access to encryption/decryption key), and (2) to tolerate site disconnections (see Section 4.3): servers hosted on the system operator's premises need to be able to recover the latest application state from just the offsite servers even though the offsite servers do not execute requests, nor maintain an active application state.

We have implemented the Partially Cloud-based BFT System architecture in the open-source Spire system, and our evaluation shows that the performance overhead can be less than 4% in terms of latency.

The contributions of this chapter are:

- The design of the first BFT system that can leverage cloud sites to achieve resilience to simultaneous network attacks and system compromises, without requiring confidential state or algorithms to be exposed to cloud servers.
- Extensions to the basic system to provide well-defined confidentiality guarantees in the case that an on-premises server is compromised. (See our Complete Confidentiality definition in Section 3.3)
- An implementation and evaluation of the system in the context of SCADA for the power grid. We show that the performance overhead of providing confidentiality is acceptable,

and the system can meet the latency requirements of power grid SCADA.

4.2 System and Threat Model

4.2.1 System Model

Our basic system model is detailed in Chapter 3, with system management shared between a system operator who manages on-premises sites and a cloud service provider who manages cloud sites. In this chapter, we assume exactly two on-premises sites for our Partially Cloud-based BFT System, where on-premises replicas from both sites participate in intrusion-tolerant ordering and executing of client requests. We design our architecture to avoid constructing any additional on-premises sites. This is because in many applications that require fault tolerance, operators are likely to already maintain two on-premises sites (e.g. for primary-backup). In contrast to adding servers to an existing site, creating a new one involves provisioning the physical location/building to house it, hiring management personnel (since fault independence requires a sufficient geographical distance from existing sites), and for some applications, provisioning specialized equipment to communicate with client sites. Thus, this assumption lets us provide strong intrusion tolerance guarantees while minimizing the amount of new infrastructure system operators need to deploy and manage. The work in [15] similarly assumes exactly two power grid control centers, arguing that for their power grid SCADA application, it is not feasible to construct additional control (on-premises) sites.

4.2.2 Threat Model

Our basic threat model is detailed in Chapter 3. Specifically, in this chapter, at any time our threat model for Partially Cloud-based BFT System includes up to f compromised replicas, up to *one* replica that is unavailable because it is going through proactive recovery, and *one* disconnected (or otherwise unavailable) geographic site. We assume that replicas are recovered one at a time, and that one replica's recovery finishes before the next replica's

recovery starts.[1]

4.2.3 Service Properties

The definitions of the service properties is detailed in Chapter 3, but here we specify under what conditions they are met for our Partially Cloud-based BFT System.

Safety (Definition 1, Section 3.3): Our system guarantees safety as long as no more than f replicas are simultaneously compromised. Note that while safety as defined above is maintained in the presence of an unlimited number of compromised clients, compromised clients may still cause the system to take incorrect actions by submitting malicious updates; we only guarantee that all replicas will observe and execute these updates in a consistent way (this is a general limitation in BFT replication). We do not consider cloud replicas as executing updates here, as we only care about the state as it is visible to clients.

Bounded Delay (Definition 3, Section 3.3): To guarantee bounded delay, we require that the conditions of our threat model are met: at most f replicas are compromised, at most one replica is undergoing proactive recovery, and at most one site is downed or disconnected due to network attack. In addition, the remaining replicas (i.e. all correct, non-recovering replicas located outside the disconnected site) must be able to communicate with one another, and the remaining correct *on-premises* replicas must be able to communicate with clients. Finally, communication among the remaining correct replicas must meet the network stability requirements of Prime [10], which is used as our underlying agreement protocol and requires that the latency variance between each pair of correct servers is bounded (see [15] for additional discussion).

Our new contribution is to combine the above guarantees with the *confidentiality* property defined in Chapter 3.

[1] Assuming that one replica's recovery finishes before the next replica's recovery starts requires certain synchrony assumptions: an attacker must not be able to arbitrarily prolong a replica's recovery (see [66]). However, in practice these can be met: simple trusted devices can trigger recovery by cycling the power to a replica, and recovery intervals on the order of one replica per day are sufficient [55]. If an adversary can prevent a replica from collecting messages needed for recovery for a full day, that replica is effectively disconnected. The intrusion-tolerant overlay makes such disconnections very difficult, and our system technically allows recoveries of replicas in a disconnected site to overlap, as long as the total number of recovering replicas is no more than the size of the largest site plus one.

Complete Confidentiality (Definition 4, Section 3.3): Our base system provides this guarantee as long as no *on-premises replica* is compromised. An unlimited number of cloud replicas may be compromised without violating confidentiality. Note that a compromised client may always leak *its own* state or updates; our model does not prevent this, nor does any other confidential BFT work we are aware of. When we refer to system state in Definition 4, we refer to the full state of the system maintained by the on-premises replicas.

Note that our Complete Confidentiality guarantee is not comparable to those of the confidential BFT systems discussed in Section 2.4: if any on-premises server is compromised (over the entire lifetime of the system), it can cause confidentiality to be violated. However, we argue that the novel combination of guarantees we provide represents a significant advance over the state of the art. The Spire system [15] provided a level of attack resilience in terms of safety and performance guarantees that was not possible before, but introduced a trade-off in terms of confidentiality. For a baseline system that provides fault tolerance through standard primary-backup mechanisms hosted fully on-premises, transitioning to the Spire architecture offers much stronger safety and performance guarantees, but at the cost of somewhat weaker confidentiality guarantees. In the baseline system, confidentiality may be violated if an on-premises server is compromised. However, if cloud sites are introduced, confidentiality is violated if *either* a cloud server or an on-premises server is compromised, and even in the case where no server is compromised, certain information is made accessible to the cloud service provider managing the cloud servers. Our architecture eliminates this trade-off: the strictly improved safety and performance guarantees are provided while maintaining *the same* level of confidentiality as in the baseline system. This is likely to substantially increase its acceptability to system operators. In Section 4.4.4, we discuss how the system can at least limit the amount of state that can be disclosed if an on-premises server is compromised.

4.3 Partially Cloud-Based BFT Architecture

The key observation behind our system design in this chapter is that network-attack resilience requires system state to be stored in at least three distinct geographic sites. As

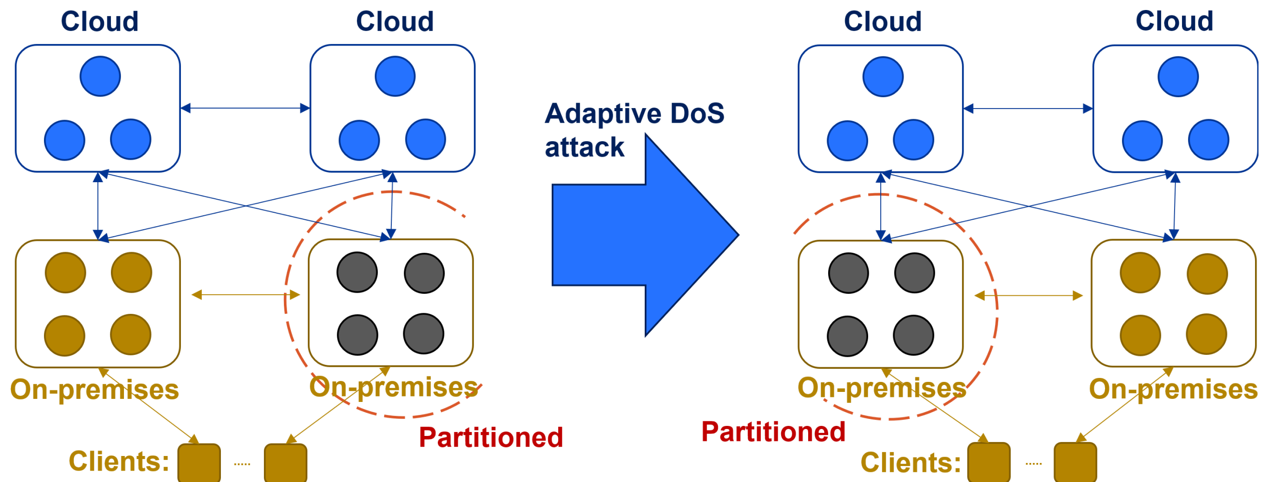


Figure 8: *Adaptive Denial of Service Attack:* as soon as one on-premises site is able to reconnect, the other is targeted and disconnected

discussed in Section 2.2, the work in [15] observed that because BFT replication protocols require (more than) a majority of replicas to be connected in order to safely make progress and order updates, they cannot guarantee continuous availability in the presence of network attacks unless at least 3 sites are used: otherwise a network attack targeting a single site can isolate a majority of the system, leaving the remainder unable to make progress, and rendering the system unavailable.

In fact, exactly the same observation applies to the storage of system state. To see why this is the case, consider a system with exactly two on-premises sites (Figure 8). Under our threat model, any one site may be disconnected at any time, so the system must be able to make progress with only a single on-premises site up and connected to the cloud sites. Consider that on-premises site A is up, connected to cloud sites and client sites, and receiving, submitting for ordering, and executing incoming client updates, while on-premises site B is under denial-of-service attack and isolated from the rest of the network. Then, the attacker shifts focus to instead target on-premises site A: site A is now isolated, while site B rejoins the network and is now connected to the cloud sites and client sites. As before, the conditions of our threat model are met, so the system *should* be able to process updates and make progress. However, on-premises site B has missed all of the client

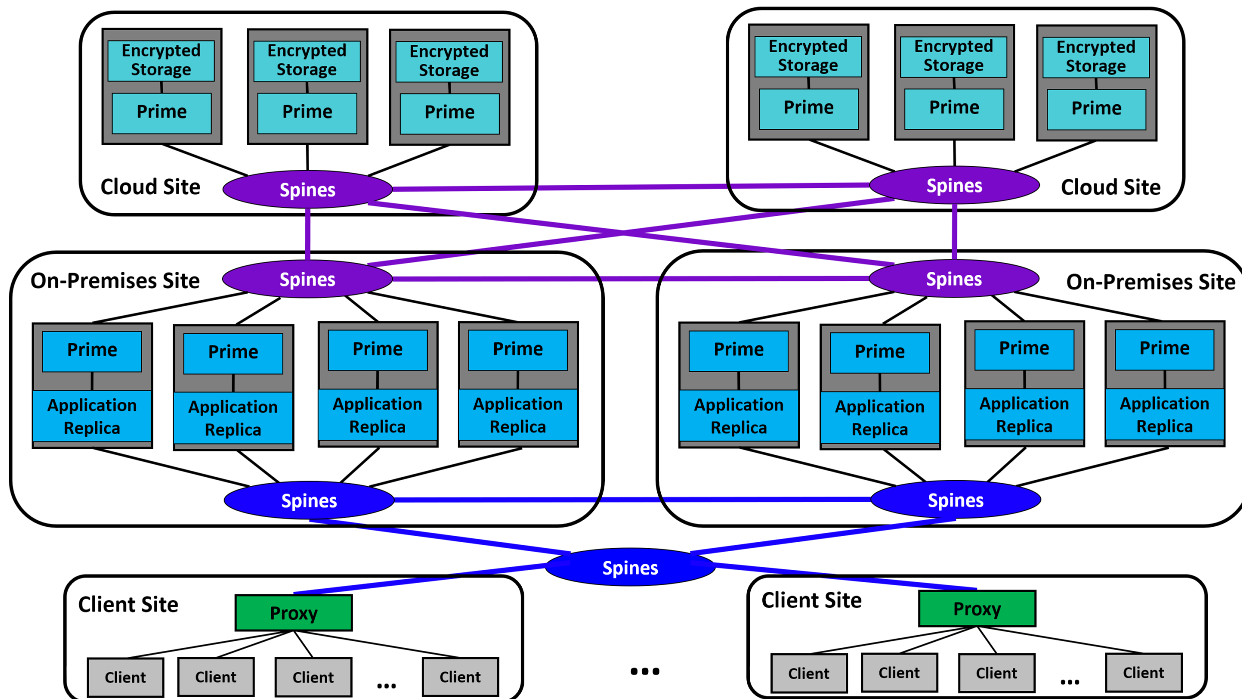


Figure 9: System architecture overview, showing 2 on-premises sites (each containing 4 replicas) and 2 cloud sites (each containing 3 replicas).

updates that were processed while it was disconnected. If cloud sites do not store any system state, it is impossible for the replicas in site B to catch up and recover the state to resume safely executing updates. In order to support our threat model, a disconnected on-premises site must be able to rejoin the network, catch up, and resume processing updates without communicating with the other on-premises site.

Therefore, our approach is for cloud replicas to store *encrypted* updates and state checkpoints. By encrypting updates and checkpoints with keys known only to the on-premises replicas, we can allow cloud replicas to store them, without being able to decrypt and interpret them. This allows a disconnected on-premises site to rejoin the network, collect state, and resume processing updates based only on information obtained from cloud replicas, but without requiring cloud replicas to access unencrypted state or perform any application-specific logic.

4.3.1 System Architecture

An overview of our architecture is shown in Figure 9. Our high-level architecture is based on the Spire architecture [15]. System replicas are distributed across two on-premises sites and a configurable number of cloud sites. Sites are connected through an instance of the Spines intrusion tolerant network [51, 68] to provide resilience to a broad range of network attacks (as detailed in Section 3.2). On-premises sites are additionally connected to client sites through a separate Spines instance. Proxies support clients that cannot be modified to use a BFT protocol. Clients in a single physical location may be grouped behind a single proxy, or each client can have its own proxy. A proxy collects updates from its respective client(s), digitally signs them so that server replicas can verify their authenticity, and submits them to the system by sending them to on-premises servers. Client proxies also validate responses received from the server replicas: specifically, server replicas generate threshold signatures on responses using an $(f+1, n)$-threshold scheme, so the proxy can verify a single signature to confirm that at least one correct replica agreed on the message.

The key difference from the Spire architecture [15] is the separation of functionality between on-premises and cloud replicas. In our system, all replicas host an instance of the Prime intrusion-tolerant replication engine [56] and participate in the replication protocol. However, only on-premises servers host application replica instances. Client updates received by on-premises servers are encrypted before being submitted for ordering and sent to cloud servers. Post-ordering, updates are decrypted and executed at (only) the on-premises application replicas, while those same updates are stored in encrypted form at the cloud servers.

4.3.2 Replica Distribution

In configuring the system, replicas must be distributed across sites such that the system is able to safely process updates and meet its bounded delay guarantee under the full threat model we consider. Prime (when configured to support proactive recovery, as in [15]) requires a total of $3f+2k+1$ replicas to tolerate f compromised replicas and k unavailable replicas, where a replica may be unavailable either because it is going through proactive recovery or

Table 2: Configurations tolerating a proactive recovery, disconnected site, and 1-3 intrusions.

		2 on-premises + 1 cloud sites	**2 on-premises + 2 cloud sites**	**2 on-premises + 3 cloud sites**
Partially Cloud -based BFT System	$f = 1$	6+6+6 (18)	4+4+3+3 (14)	4+4+2+2+2 (14)
	$f = 2$	9+9+9 (27)	6+6+5+4 (21)	6+6+3+3+3 (21)
	$f = 3$	12+12+12 (36)	8+8+6+6 (28)	8+8+4+4+4 (28)
Network Attack Resilient System [15]	$f = 1$	6+6+6 (18)	3+3+3+3 (12)	3+3+2+2+2 (12)
	$f = 2$	9+9+9 (27)	5+5+5+4 (19)	4+4+3+3+3 (17)
	$f = 3$	12+12+12 (36)	6+6+6+6 (24)	5+5+4+4+4 (22)

because it has been disconnected from the network (or because it has simply crashed). In order to guarantee progress, with bounded delay, at least $2f + k + 1$ of those replicas must be up, correct, and connected (with sufficient network stability).

The work in [15] showed that in order to ensure $2f + k + 1$ correct replicas are always available, it is necessary to ensure that no single site contains more than $k - 1$ servers: otherwise the disconnection of a single site, plus an ongoing proactive recovery elsewhere in the system could cause more than k replicas to become unavailable at the same time, preventing the system from making progress. That work shows that providing this guarantee requires setting $k \geq \lceil \frac{3f+S+1}{S-2} \rceil$, where S is the total number of sites (on-premises + cloud), and distributing replicas as evenly as possible across sites [15, 70].

However, an additional constraint under our threat model comes from the separation of functionality between on-premises and cloud replicas: only on-premises replicas can execute updates and communicate with clients. To verify that a received message is correct, a client must be able to confirm that $f + 1$ servers agreed to it (to ensure at least one correct server was involved). This means that generating verifiable responses requires that $f + 1$ *on-premises* replicas are available at any time: cloud replicas cannot participate in generating client responses, as this requires knowledge of update contents, system state, and application logic.

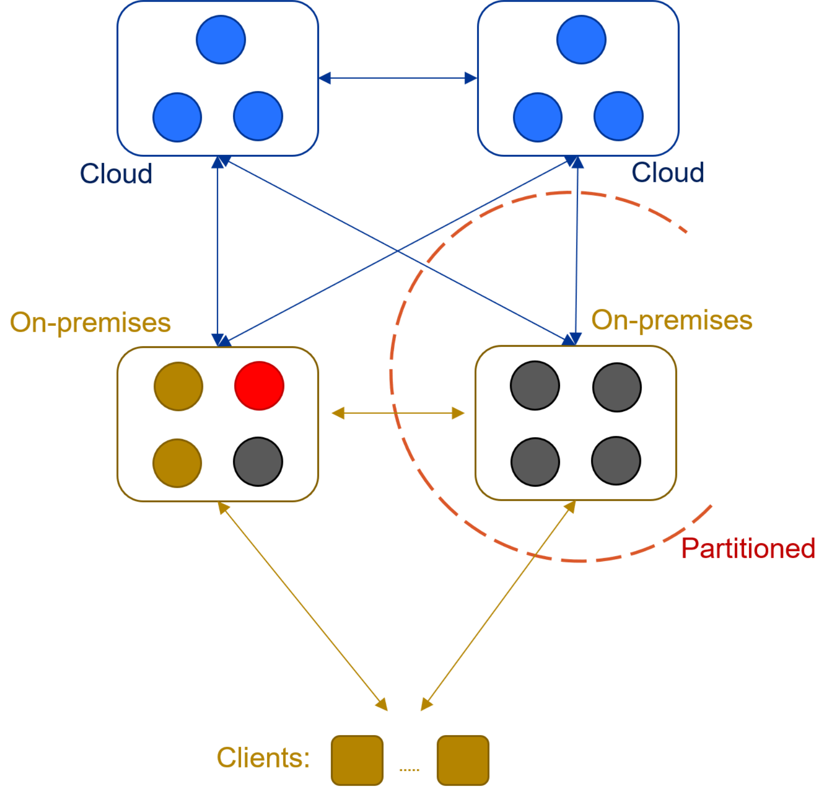


Figure 10: Partially Cloud-based BFT System with configuration "4+4+3+3": tolerates 1 intrusion, 1 proactive recovery, and 1 site disconnection

In the worst case, under our threat model, one of the two on-premises sites may be disconnected, and the other may contain f compromised replicas and one replica undergoing proactive recovery. Therefore, in order to ensure that $f+1$ correct replicas are available at all times, *each* of the two on-premises sites must contain at least $2f+2$ total replicas ($2f+2$ replicas $-\,1$ recovering replica $-\,f$ compromised replicas $=f+1$ available correct replicas). However, since we must still have k strictly greater than the size of the largest site, this adds the restriction $k \geq 2f+3$.

Together, the two above restrictions give us the requirement:

$$k \geq \max\left(2f+3, \left\lceil \frac{3f+S+1}{S-2} \right\rceil\right)$$

After finding the minimal value of k using this formula, the total number of required replicas is calculated from the original formula: $n = 3f + 2k + 1$. To distribute these

replicas across the sites, we must first ensure that at least $2f + 2$ replicas are placed in each on-premises site, and then distribute the remaining replicas across the sites such that the total number of replicas per site is as even as possible. The results of this process for several different system options are shown in Table 2. In Table 2, we consider configurations tolerating 1-3 intrusions ($f = 1$, $f = 2$, and $f = 3$), with replicas distributed across two on premises sites and 1-3 cloud sites. The first 2 numbers per cell denote the number of replicas in each on-premises site, while the following numbers represent the number of replicas in the cloud sites, and the final number in parentheses represents the total number of replicas. For example, configuration "4+4+3+3" (Figure 10) represents 4 replicas in each on-premises site, and 3 replicas in each cloud site, for a total of 14 required replicas.

While the total number of replicas is considerably higher than the typical $3f + 1$, this is because we (1) support proactive recovery (which requires $3f + 2k + 1$ replicas) and (2) provide stronger guarantees, tolerating not only f compromises, but also network attacks that can disconnect an entire site. This threat model was first introduced in [15], which showed that for the case of $f = 1$, 12 replicas are needed. We slightly increase that number to 14, but we believe this is justified to provide the confidentiality needed to trust a cloud service provider and thus avoid the need for the system operator to manage the large set of sites and replicas themselves. If we consider a system operator who already supports fault tolerance, deploying primary and backup sites, each of which includes primary and backup replicas, we only require that they add 2 on-premises servers per site: the remaining sites and replicas are fully managed by the cloud service provider.

4.4 Protocols for Partially Cloud-Based BFT

Having described how to distribute replicas and set up the system, we next describe the protocols used to submit and process updates. We visualize the client request flow in Figure 11.

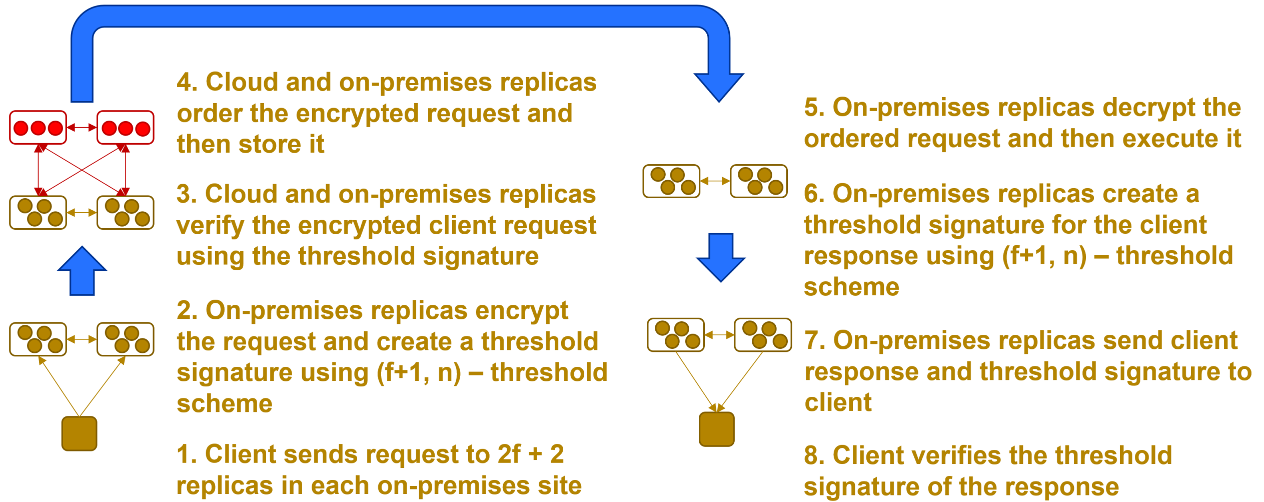


Figure 11: Client Request Flow in Partially Cloud-based BFT System

4.4.1 Introducing Client Updates

Clients submit updates to the system through proxies. These proxies digitally sign each update using their private keys before forwarding them to the on-premises servers. Each client update sent from the proxy to the on-premises server includes the client's identification number, request sequence number, message body, and the digital signature created by the proxy. On-premises servers can then verify the signature on the update, encrypt it, and inject it into Prime for ordering.

However, our new model introduces a challenge, as cloud replicas need to verify that each update submitted for ordering actually came from a correct client (and was not maliciously generated by an adversary), yet cloud replicas do not have the ability to decrypt client updates. In fact, they should not be required to maintain any information about client identities or public keys (in some cases, client IP addresses or locations may be a sensitive type of state that system operators would like to avoid revealing [14]). Requiring updates to be signed by the on-premises server injecting them is not sufficient, as any individual server could be compromised and manufacture a large number of spurious updates, forcing the system to work to order the bogus updates.

Our approach is for on-premises servers to cooperate to generate a threshold signature on each introduced update. To do this, we require each client to send its updates to $2f + k + 1$

on-premises replicas, which guarantees that at least $f + 1$ correct replicas will receive the request. Upon receiving a client request, the on-premises replica first checks its validity, then encrypts the request message body. Note that we do not encrypt client identification number and client request sequence number, since these numbers are needed to prevent replay attacks (otherwise, cloud replicas will not be able to distinguish old requests from new requests). In Chapter 5, we *fully* encrypt client requests (including client identification number and client request sequence number), and develop a novel method to prevent replay attacks that does not rely on any client information.

Each *encrypted client update* consists of the encrypted message body, while client identification number, request sequence number and digital signature are left in the clear. The on-premises replica creates a partial threshold signature (using an $(f + 1, n)$-threshold scheme) for the encrypted client update and multicasts this partial threshold signature to all other on-premises replicas. Upon collecting $f + 1$ partial signatures, an on-premises replica combines them to form the full threshold signature, and injects the threshold-signed encrypted client update into Prime for ordering. All other replicas, including the cloud replicas, can verify the full threshold signature to validate that a request is legitimate.

Client update encryption presents one remaining challenge: the individual on-premises replicas all need to perform the encryption independently but come up with *the same encrypted content*, so that the threshold signature shares that they independently generate will combine correctly. To do this, we assume on-premises replicas maintain two shared secret keys per client: the first is the shared key used to perform symmetric encryption and decryption of updates for that client, while the second is used as one of the inputs to a pseudorandom function used to generate initialization vectors, similarly to the approach in [34]. Details about our implementation can be found in Section 4.5.2. For now, we assume that all of these per-client key pairs are stored in persistent read-only memory and reloaded from there after proactive recovery, although we discuss how to weaken this restriction in Section 4.4.4.

4.4.2 Ordering Updates and Disseminating Results

Once an on-premises server generates a full threshold signature on an encrypted client update and injects it into Prime, it is assigned an *ordinal*, or sequence number in the global total ordering through the Prime agreement protocol [10], and then delivered to the application to be decrypted and executed (in the case of on-premises servers) or simply stored in encrypted form along with its ordinal (at cloud servers).

As part of executing an ordered update, application replicas may generate a response message that needs to be sent to a client. To generate a single response that can be verified by a client proxy based on a single service public key, application replicas generate a threshold signature on the response, again using an $(f+1, n)$-threshold scheme to ensure the message is agreed on by at least one correct replica. This is the same approach as in [15], but in our case, only on-premises replicas can participate in generating the response. Our replica distribution framework (Section 4.3.2) guarantees that it is always possible to generate such a signature under the conditions of our threat model.

4.4.3 Checkpoints and State Transfer

Since storing every ordered client request will eventually exhaust replicas' storage capacity, we keep only a limited number of the latest encrypted client requests and replace older requests by encrypted checkpoints. At specified checkpoint intervals (i.e. every C ordered updates), each on-premises replica creates and encrypts a checkpoint that represents its state up through the execution of the last ordered client update (similar to [21] and others). Note that an on-premises replica does not consider itself to have fully executed a particular update until it has generated and sent a threshold-signed client response message for any outgoing messages that were generated as a result of its execution. The latest (threshold-signed) outgoing message for each client is included in the system state, since these may need to be retransmitted.

After generating an encrypted checkpoint, the on-premises replica then creates and signs a checkpoint message that contains the encrypted checkpoint, as well as the (cleartext) sequence number it corresponds to (the global sequence number of the last ordered update

that was executed and reflected in the state). The replica then multicasts this checkpoint message to all other replicas (including both on-premises and cloud replicas). When a replica (on-premises or cloud) receives $f + 1$ identical encrypted checkpoints from different replicas for the same sequence number, then this encrypted checkpoint can be marked as *correct*: at least 1 correct replica has agreed that this checkpoint represents the system state at the given sequence number.

Cloud replicas do not create their own checkpoints. Instead, when a cloud replica collects a *correct* encrypted checkpoint, it creates and signs a checkpoint message containing that encrypted checkpoint, and then multicasts this checkpoint message to all other replicas. When a replica (on-premises or cloud) receives $2f + k + 1$ identical encrypted checkpoints from different replicas for the same sequence number, then this encrypted checkpoint can be marked as *stable*: even if f replicas sending checkpoints are malicious, and k immediately become disconnected/unavailable, at least $f + 1$ correct replicas still remain that can help another replica catch up to this checkpoint.

Upon collecting a *stable* checkpoint for a given sequence number, a replica may safely garbage collect stored updates and checkpoints for all prior sequence numbers (similar to [21] and others), as long as it has also fully executed all sequence numbers up through the sequence of the stable checkpoint (note that since cloud replicas do not participate in generating client responses, they consider an update to be fully executed as soon as it is ordered).

When a replica detects that it has fallen behind (e.g. because it went through proactive recovery, or was disconnected and missed some updates), it submits a state transfer request to Prime for ordering. When this request is ordered, the other replicas (including cloud replicas) execute it by sending the recovering replica their *stable* encrypted checkpoint, *digests* for any correct checkpoints they have with sequence numbers higher than the stable checkpoint, and the list of ordered, encrypted client requests with sequence numbers between the *stable* checkpoint and the global sequence number of the state transfer request from this recovering replica.

In order to catch up to the latest state, the recovering replica waits to receive a set of state transfer responses such that it has (1) a *correct* checkpoint, with at least $f+1$ matching

checkpoints/digests to guarantee its validity, and (2) a set of updates such that for every sequence number between its latest correct checkpoint and its target recovery ordinal (i.e. the sequence number at which its state transfer request was ordered), it has $f+1$ identical updates from distinct replicas. Once these requirements are met, if the recovering replica is a cloud replica, then it simply stores the latest correct checkpoint and all following correct updates in the already encrypted format. If the recovering replica is an on-premises replica, it additionally decrypts the encrypted checkpoint and the list of client requests, and then applies the decrypted checkpoint and client requests in order of increasing global ordinals to bring its application state up to date.

4.4.4 Key Renewal

As described so far, a single on-premises compromise can leak encryption keys. If the keys are sent to a cloud replica, it will be able to decrypt all following updates and checkpoints. While system operators could recover from such a situation (if it was detected) by manually backing up the system, taking replicas down, bringing them back up with new keys, and re-instantiating the system, this is a labor intensive operation that is likely to require system downtime.

Therefore, we extend the basic protocol with an automatic key renewal mechanism that, combined with proactive recovery, limits the amount of confidential state a compromised on-premises replica can disclose. The basic idea is that on-premises servers maintain a separate shared symmetric encryption key and shared pseudorandom function key for each client in the system, and a given client key pair (encryption key + pseudorandom function key) is only valid for a fixed, predetermined range of client update sequence numbers. When servers get near the end of the range of sequence numbers the current client key pair is valid for, they each (independently) randomly generate a new pair of keys and propose their generated key pair by injecting it into Prime for ordering together with the proposed client sequence number range it should be valid for.

Since all correct replicas observe the ordered stream of messages from Prime in the same way, they can use the global ordering of proposals to determine the new key pair in

a consistent way. For example, we can determine the keys as a combination of the first $f+1$ proposals, guaranteeing that it includes random input from at least one correct replica, so that the process cannot be controlled solely by malicious replicas. Since the replica distribution process described in Section 4.3.2 guarantees that $f+1$ correct on-premises are always available under our threat model, it is always possible to collect $f+1$ proposals, so this approach is live. A correct server will not agree to inject a client message for ordering (i.e. will not generate its signature share as described in Section 4.4.1) unless it has received $f+1$ valid proposals ordered by Prime for the sequence range covering that message and thus determined the correct key to use for encryption.

While this process allows replicas to agree on new keys to use for encrypting client updates such that all (correct) replicas will apply the same new keys starting at the same client sequence number, there are still several issues to resolve to provide well defined confidentiality guarantees.

Encrypting Key Proposals. First, the new key proposals themselves must also be encrypted, since they are disseminated to cloud replicas as part of the ordering process. It is not possible to avoid storing these updates at the cloud replicas for exactly the same reason that cloud replicas must store general client updates: in order to ensure continuous availability under our threat model, on-premises replicas that have been disconnected and are rejoining the system must be able to recover the state and resume executing updates based only on input from the cloud replicas.

But, what keys can we use to encrypt the key proposal messages? Clearly, it is not safe to use the previous client encryption key, since the purpose of the key refresh is to recover from the case where the previous key was compromised. But, if some other key is used, then rejoining/recovering on-premises replicas must be able to recover that key from the cloud sites, which means that key needs to be stored in encrypted form, and we have the same problem again.

To solve this issue, we rely on a hardware-based root of trust. We assume each on-premises replica is configured at the time the system is set up with a shared symmetric encryption key that can only be accessed from within trusted hardware (e.g. TPM or Intel SGX [29]) and persists across reboots. An attacker who compromises a server but does not

have physical access may use the key for encryption while it has access to the machine, but cannot exfiltrate, modify, or delete the key. This permanent key is used to encrypt new key proposals: with this approach, they cannot be decrypted by cloud replicas (or external observers), but can be decrypted by recovering/rejoining on-premises replicas (without requiring the recovering/rejoining replicas to retrieve keys from cloud replicas). We note that this assumption of a limited degree of trusted hardware is not an unreasonable requirement, as proactive recovery already requires each replica to maintain a persistent hardware-based (TPM) asymmetric private signing key that it uses to authenticate itself and establish new session-level signing keys during its recovery process.

Adapting State Transfer. Given that key proposal messages will eventually be garbage collected, we must also extend state checkpoints to additionally include the current encryption and pseudorandom function keys for each client and their validity periods (i.e. the highest sequence number they can be used for), as well as any valid pending key proposal messages. By *pending* key proposal message, we mean a key proposal that has been ordered, but not yet used to generate a new key, as not enough proposals for the same client and validity period were ordered before the checkpoint was taken. With this extension, checkpoints are also encrypted using the hardware-protected symmetric key (although it is also possible to treat checkpoints as another logical client, with a new session-level key agreed on for each checkpoint. In this case, it is only necessary to encrypt the part of the checkpoint containing the session keys with the persistent hardware-protected key).

Limiting Disclosure. Finally, in order to guarantee limits on the amount of confidential state a compromised on-premises replica can expose, we must ensure that compromised replicas cannot control the selection of *future* encryption keys that will be used after they have gone through the proactive recovery process and been restored to a correct state. To do this, we enforce that new key proposals are only accepted as valid if they are introduced at the correct logical time. That is, we define a sequence number slack parameter x that represents how far in advance of the sequence range a key is intended to be active for it can be proposed. For example, if we consider $x = 10$ and a key validity period of 100 updates, a new key proposal for range 101-200 will not be considered as valid (and included in the computation of the actual new key) unless it is ordered *after* update 90 for the relevant client

in the global total ordering created by Prime. Since all correct replicas observe the ordered stream of updates in the same way, all will make the same decision as to a key proposal's validity.

An additional concern may be that a compromised replica could, while it is compromised, generate, encrypt, and sign proposals for future client sequence numbers, and send them to a malicious external collaborator to inject at the appropriate time. However, since such messages are required to be signed with the replica's session-level signing key, which is refreshed following a proactive recovery, this is not a problem.

Our key renewal procedure does not provide complete confidentiality (in the sense of Definition 4) in the presence of a compromised on premises replica, but it limits the damage such a replica can do. In particular, for a client key validity period V and slack parameter x, it guarantees that any keys leaked by a compromised replica will only be able to decrypt a maximum of $V + x$ updates per client that are issued *after* the replica is recovered (of course, the compromised replica may leak all updates issued while it is compromised). In addition, since checkpoints are encrypted with keys that cannot be exfiltrated from their physical machine, no checkpoint constructed after the replica is recovered can be decrypted using keys it leaked while compromised. Thus, as long as replicas are periodically proactively recovered and clients continue to issue updates, the system will eventually return to a situation where its state is fully confidential, if no new on-premises compromises occur. Unfortunately, this does not apply to state manipulation algorithms: since those are likely to change rarely, once a replica with access to those algorithms is compromised, we can no longer provide guarantees of their confidentiality.

4.5 Confidential Spire Implementation

We have implemented our architecture and protocols in Confidential Spire, a SCADA system for the power grid that provides the Safety, Bounded Delay, and Confidentiality guarantees defined in Section 3.3 under the threat model stated in Section 4.2.2. Our Confidential Spire implementation is built on the open source Spire version 1.2 [69], which implements

the architecture described in [15], and provides Safety and Bounded Delay (but not Confidentiality) under our same threat model. Confidential Spire has been incorporated into the publicly available open source Spire release as of version 2.0 [69].

In Confidential Spire, SCADA control centers serve as the on-premises sites, and the clients submitting updates to the system are Remote Terminal Units (RTUs) and Programmable Logic Controllers (PLCs) that interact with the power grid equipment, and Human Machine Interfaces (HMIs) that operators use to issue commands and view the system state.

The Spire 1.2 implementation already includes a SCADA master application and RTU or PLC proxies. Its system components communicate over the Spines intrusion-tolerant network [68], and updates are ordered using the Prime intrusion tolerant replication engine [56]. An intrusion-tolerant communication library (the Intrusion-Tolerant Reliable Channel, or ITRC) manages communications between client proxies and the control center servers, as well as between Prime and the SCADA Master application.

4.5.1 Confidentiality-Preserving Intrusion Tolerant Middleware

Confidential Spire adapts and extends Spire's intrusion-tolerant communication library into a Confidentiality-Preserving Intrusion-Tolerant Middleware (CP-ITM). While the CP-ITM serves the same basic functions as Spire's ITRC, it additionally supports encryption and decryption of client updates, the creation (and encryption) of periodic checkpoints, and a new checkpoint-based state transfer protocol. The CP-ITM is intended to be a generic middleware that can handle client communication and state management/transfer for any application.

4.5.2 Encryption and Decryption Details

The CP-ITM encrypts client requests before injecting them into Prime and decrypts them before delivering them to the SCADA Master application. For each client, the CP-ITM maintains a shared symmetric encryption key and a pseudorandom function key (which can be periodically refreshed as described in Section 4.4.4, though this is not currently imple-

mented). To encrypt a request, the CP-ITM generates a hash-based message authentication code (HMAC) based on the update request itself and the shared pseudorandom function key for that client, following the approach of [34]. Then, the client update request is encrypted using AES-256 in CBC mode with this HMAC as the initialization vector (IV) and the client's shared encryption key.

Since the encryption key and pseudorandom function key for each client are shared across all control center CP-ITM instances, they all generate the same encrypted result for a client request by using the above method. We note that even if a client issues the same request multiple times, it will not result in the same encrypted output, as the client sequence number is included in the message content over which the HMAC is generated and in the content that is encrypted. The CP-ITM can decrypt encrypted content using the shared encryption key for that client and the IV (HMAC) which is included in the message header as cleartext.

4.5.3 Checkpointing and State Transfer Implementation

When the CP-ITM running in a control center replica determines that a new checkpoint is needed (i.e. that C updates have been ordered since the previous checkpoint), it requests the SCADA master to package and send back a snapshot of the current state of the system. Before the CP-ITM multicasts this checkpoint to other replicas, it encrypts the checkpoint using the same method as described in Section 4.5.2 (the associated ordered sequence number is not encrypted since this is needed to distinguish an old encrypted checkpoint from a new one). Every CP-ITM instance maintains an additional shared pseudorandom key and encryption key (in addition to the client key pairs) for encrypting and decrypting the checkpoints (which can be hardware-protected, as discussed in Section 4.4.4). In this way, all control center CP-ITM instances can independently generate identical encrypted checkpoints.

When a replica requires a state transfer, its CP-ITM collects the correct encrypted checkpoint and the correct set of updates following the protocol in Section 4.4.3. When the CP-ITM is done collecting, if it is running on a cloud replica, then it simply stores the encrypted checkpoint and updates and continues operations in normal status. However, if the CP-ITM is running on a control center replica, it decrypts and sends the correct checkpoint

to the SCADA Master to apply, and then decrypts and sends each collected update request in the order of their sequence numbers to the SCADA Master. Finally, it does the same with any new ordered encrypted client requests that were pending while waiting to collect state, and resumes normal operations.

4.6 Evaluation

We first evaluate the overhead of providing confidentiality in our approach by comparing our Confidential Spire implementation to Spire 1.2 [69], and then evaluate our implementation's performance under particular types of attacks.

For all experiments, we emulate a power grid SCADA setup with control centers (on-premises sites) and cloud sites spanning about 250 miles of the US East Coast. Experiments are conducted in a local area network, but latencies between sites are emulated to reflect this geographic distribution. We emulate ten power grid substations each injecting updates via proxies at a rate of one per second per substation.

4.6.1 Performance Overhead of Confidentiality

To assess the performance overhead of our approach, we compare Confidential Spire to Spire 1.2 in two different configurations: one tolerating one compromised replica ($f = 1$) and one tolerating two compromised replicas ($f = 2$). Both configurations additionally tolerate a proactive recovery and disconnected site to support our full threat model.

We consider configurations using two control center sites and two cloud sites, as these were shown to be the most practical for Spire [15]. The 4-site configurations are also the most reasonable for Confidential Spire, as they allow us to use fewer total replicas compared to configurations using only one cloud site, but using additional cloud sites beyond two does not provide further benefits, due to the requirements on the number of replicas per control center (see Table 2).

Therefore, for the $f = 1$ configurations, we evaluate configuration "3+3+3+3" (three

Table 3: Spire and Confidential Spire normal operation performance for 36,000 Updates over 1 hour

	f	Setup	Avg Latency	0.1 percentile	1 percentile	50 percentile	99 percentile	99.9 percentile
Spire	1	3+3+3+3	51.7 ms	39.7 ms	41.0 ms	51.7 ms	62.4 ms	63.9 ms
	2	5+5+5+4	54.4 ms	42.5 ms	43.6 ms	54.4 ms	65.6 ms	67.7 ms
Confidential Spire	1	4+4+3+3	53.6 ms	41.6 ms	42.8 ms	53.6 ms	64.2 ms	66.1 ms
	2	6+6+5+4	61.2 ms	46.0 ms	47.5 ms	61.1 ms	78.4 ms	86.2 ms

replicas in each of 2 control centers and 2 cloud sites) for Spire 1.2 and configuration "4+4+3+3" (four replicas in each of 2 control centers and 3 replicas in each of 2 cloud sites) for Confidential Spire. For tolerating 2 simultaneous intrusions, we use the "5+5+5+4" configuration for Spire 1.2, and the equivalent "6+6+5+4" configuration in Confidential Spire. We ran each configuration for 1 hour and report the resulting update latencies in Table 3.

From the results for the $f = 1$ configurations, we can see that Confidential Spire adds a small constant latency overhead of about 2ms. This increase in overhead is small because it avoids adding any new wide-area communication on the critical path compared with Spire 1.2. While Confidential Spire requires control center replicas to cooperate to generate a threshold signature on each incoming client request, it is always possible for a replica to collect the needed $f + 1$ signature shares from replicas within its own site, since each on-premises site contains $2f + 2$ replicas. Hence, Confidential Spire only utilizes the local-area network for the added communications. The sum of computational overhead, to compute the signatures and to encrypt/decrypt the requests, and the local-area network communications overhead is small compared to the multiple rounds of wide-area network message exchanges needed for the agreement protocol. While Confidential Spire also adds computation and communication for checkpoint creation and exchange, this occurs off the critical path of request processing and thus does not have a significant effect on latency.

In the $f = 2$ case, we can see that Confidential Spire's "6+6+5+4" configuration adds

somewhat more overhead, increasing average latency by about 6.8ms as compared to Spire's "5+5+5+4" configuration. This is about 3.5 times the average latency increase noted above in the $f = 1$ case. This can be explained by increasing communication overheads, due to the all-to-all communication patterns, as the number of replicas increases. However, this is still acceptable, as the results show that our Confidential Spire implementation still meets the timeliness requirements for power grid SCADA systems (processing updates within 100ms), even while tolerating 2 intrusions. The observed 99.9 percentile latency is 86.2ms, and no update crossed the 100ms threshold. We expect that these can even be further reduced through improved engineering of the communication protocols to reduce the rate at which traffic is sent, and of the cryptographic mechanisms to reduce processing overheads. As shown by [55] and noted in [15], the majority of the advantages of proactive recovery can be obtained by tolerating two intrusions, instead of only one, making "6+6+5+4" a useful configuration to support.

4.6.2 Attack Evaluation of Confidential Spire

We next evaluate Confidential Spire's ability to meet the timeliness requirements of power grid SCADA systems while under attack. Such systems require responses within 100ms in the normal case but may tolerate latencies up to 200ms in certain situations [40], [31]. We consider the effect of proactive recovery on performance, as well as network attacks that cause a site to be disconnected. While we do not explicitly evaluate malicious actions by protocol replicas, we note that many types of malicious actions closely resemble proactive recovery of the leader replica in terms of performance: once the leader takes a malicious action (e.g. sending conflicting messages), a view change is triggered to elect a new leader.[2] The Confidential Spire "4+4+3+3" configuration's performance under all combinations of recoveries and site disconnections is illustrated in Figure 12. We can see that proactive recovery of a leader replica, which occurs between 1:00 and 1:30, causes one client update to spike over 100ms, when the system must perform a view change. Recovery of a non-leader

[2]It is possible to reduce the performance impact of proactive recovery by preemptively changing the leader, but since our implementation does not do that, our experimental results accurately reflect the case where timeouts must expire before changing the leader.

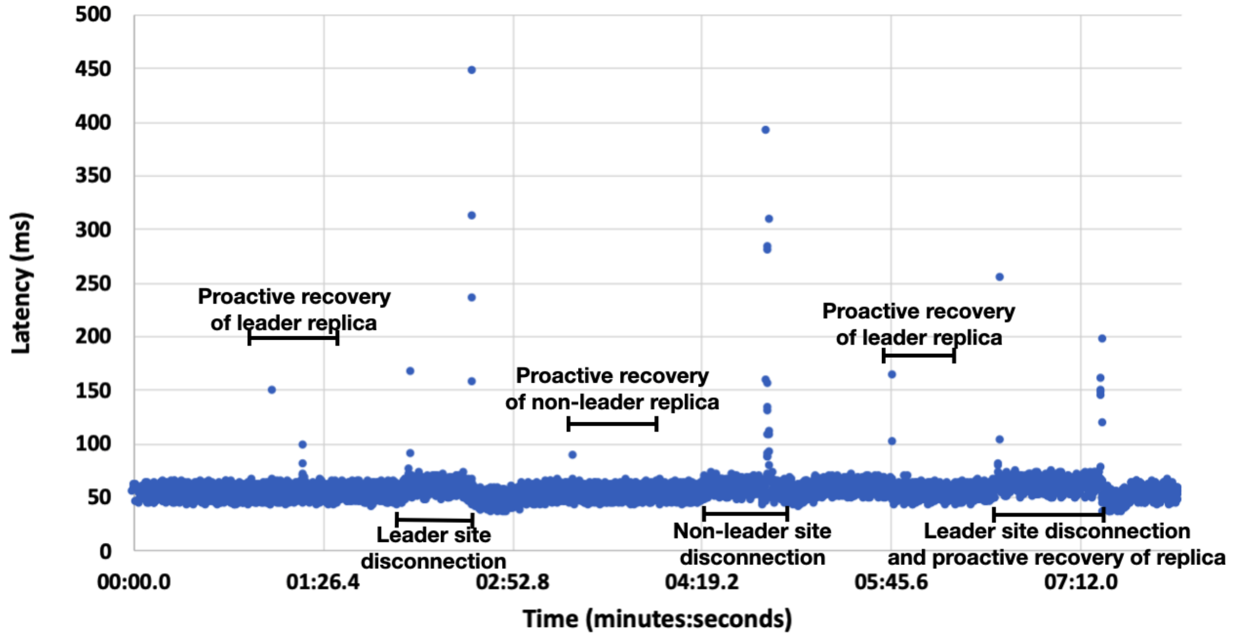


Figure 12: Latency of Confidential Spire in the presence of proactive recoveries and site disconnections based on configuration "4+4+3+3"

replica (the more common case), which occurs between 3:15 and 3:45, has almost no impact on performance. In this case, we only see one client update with higher than average latency, but it is still below the 100ms threshold.

Similarly, there is no latency spike when we disconnect a non-leader site, at 4:19, since there is no view change. However, we can see a few client updates spiking, with one rising above 100ms, though still under 200ms, when the leader-site is disconnected at 2:00, since this requires a view change. We also note a small (but still acceptable) increase in average latency for the duration of the time either the leader or non-leader site is disconnected in this experiment, as in both cases it renders the fastest quorum of replicas unavailable, and requires communication with a more distant site to occur on the critical path. However, when reconnecting a disconnected site, we can see significant latency spikes, crossing the 200ms threshold and reaching up to 450ms (e.g. at 2:30 and 5:00). This is due to the large number of checkpoint messages and update messages being sent over the network from the correct replicas to help catch up the lagging replicas in the site which just rejoined the network. While this is a limitation of our current implementation, we note that this is not

an inherent limitation of the protocol, and should be fixable by engineering a better message flow control for the checkpoint messages and update messages that are being sent to catch up the other replicas. Overall, these results indicate that, even with the overhead of providing confidentiality, our system can provide the necessary performance, even while under attack.

5.0 A Cloud-Based Hybrid Management Approach to Deploying Resilient Systems

5.1 Overview

To provide intrusion tolerance for general applications, while offloading part of the system management to a cloud provider, the work in Chapter 4 developed an architecture for partially cloud-based BFT systems, which maintains confidentiality of application state, application logic, and client locations. But, it still exposes client IDs and request patterns to the cloud, and, even more importantly is an integrated system, where the assumption is that the system operator manages all of the software and simply uses cloud resources to avoid deploying additional physical sites.

Deploying and managing BFT-replicated systems in practice requires both specialized technical expertise and substantial investment in additional physical infrastructure. In this work, we address this gap by designing a hybrid management model: while the system operator manages their application, a service provider hosts and manages the BFT replication service using cloud infrastructure.

To make this approach feasible, there are two core challenges: (1) to make using the cloud acceptable, we should enforce strict confidentiality, such that no application state, state manipulation algorithms, or client information is exposed to the cloud, and (2) to make system management practical, the cloud and application operators should each be able to manage their systems *independently*, without needing to know the internals of each other's systems or coordinate with one another.

We develop the protocols to support this system architecture, without revealing application state, algorithms, or client information to the cloud provider, even when application servers are compromised. We implement and evaluate concrete configurations of our hybrid management model in the context of an industrial control system and show that they meet their performance and resilience requirements.

The main contributions of this chapter are:

- We define a new hybrid management model for intrusion-tolerant systems, where system operators control their applications, but leverage *intrusion-tolerant ordering* and *encrypted storage* services from a cloud provider.
- We design a concrete system architecture that implements the hybrid management model and enforces the confidentiality of application state, algorithms, and client request patterns.
- We show that this system architecture can provide resilience to a broad threat model that includes intrusions and network attacks, and is able to recover from management domain failures that affect all replicas hosted by the system operator (on-premises).
- We implement and evaluate the architecture in the context of an industrial control application. We show that, while it increases latency by about 9ms (18%) compared to a fully system-operator-managed BFT system, it still meets the application's performance requirements.

5.2 Architecture Outline

In this chapter, we introduce a new Decoupled Intrusion-Tolerant System, which is designed to provide intrusion tolerance for any application that can work with the state machine replication model. The basic system model is detailed in Chapter 3. Additionally, in this chapter, we enforce a strict separation of responsibilities between the cloud and on-premises sites, as shown in Figure 13. The cloud service provider is responsible for deploying and managing the *BFT Replication Engine* which runs *only* in the cloud. The BFT Replication Engine consists of a set of *cloud replicas (CRs)* running a BFT replication protocol (any one of the many existing BFT protocols can be used). It provides an **intrusion-tolerant ordering and encrypted state maintenance service** that is used to turn the application, which runs only in the on-premises site(s), into an *Intrusion-Tolerant Application*. Note that the cloud service provider running the BFT replication engine may be an existing cloud provider, or, we envision that our new architecture can allow a specialized *intrusion-tolerance-as-a-service (ITaaS) provider* to emerge. The cloud service provider can scale their

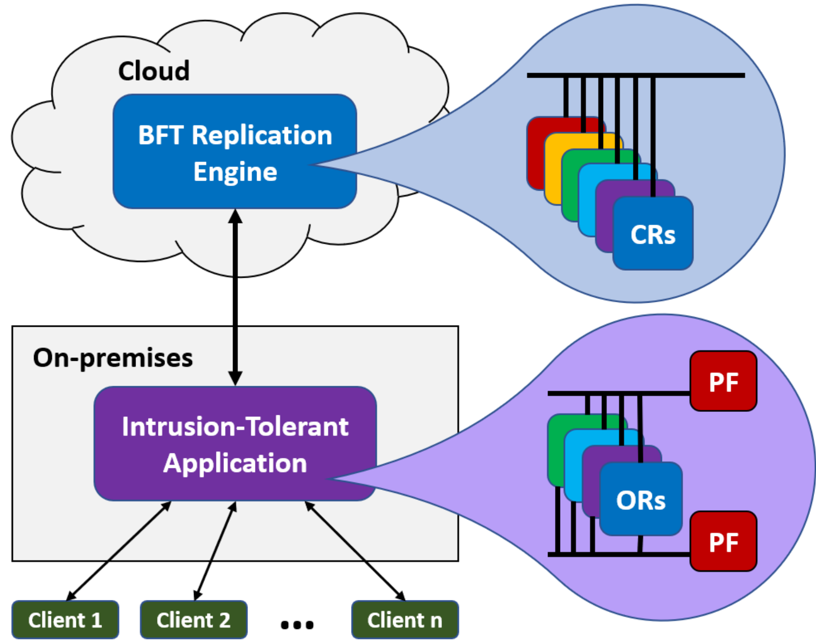


Figure 13: Cloud-based Hybrid Management Intuition

service effectively by using the same infrastructure and BFT Replication Engine to support many applications. Leakage between applications is not an issue since data is encrypted by the on-premises replicas before being shared with the cloud replicas (encryption/decryption keys are only available in on-premises replicas).

The system operator is responsible for deploying and managing the Intrusion-Tolerant Application which consists of a set of *on-premises replicas (ORs)*. On-premises replicas accept incoming requests from *clients* and forward them to the BFT Replication Engine for ordering. The cloud replicas establish a total order on the requests, and the on-premises replicas execute requests according to this total order and return responses to clients.

5.2.1 Simplifying Interfaces via Threshold Signatures

To make the interface between the cloud and on-premises domains as simple as possible and avoid requiring either domain to know internal configurations of the other, we use threshold signatures. In our architecture, *all* messages between domains must be threshold-signed. With this approach, the cloud replicas only know a single *on-premises public key* to

authenticate messages sent by on-premises replicas, and the on-premises replicas similarly only need to know a single *cloud public key* to authenticate messages sent by the cloud replicas.

We use an $(f_o + 1, n_o)$-threshold scheme for each on-premises site, where $f_o + 1$ shares out of n_o total shares are needed to generate a valid signature (where n_o is the number of on-premises replicas per site). Thus, a valid threshold signature guarantees that at least one correct on-premises replica agreed to the content of the message. Key shares can be refreshed without changing the public service key [77, 59]. Hence, there is no need to involve or update the cloud domain when system operators refresh the key shares in their on-premises replicas. Similarly, messages sent from the cloud replicas to the on-premises replicas are signed using an $(f_c + 1, n_c)$-threshold scheme (where n_c is the total number of cloud replicas).

5.2.2 Strengthening Confidentiality via Privacy Firewalls

A privacy firewall (PF) prevents leak of confidential data from a site by filtering out messages that a *correct* replica will not endorse [75, 34]. We strategically place privacy firewalls in our networks such that *malicious* on-premises replicas cannot leak confidential data. We envision two separate networks: one connecting the clients with the on-premises sites, and the other connecting the cloud sites with the on-premises sites. Ideally, each on-premises replica connects to each of these networks using separate network interfaces, with another interface used for local-area communication with the other replicas in its site. To ensure that confidential data does not leave an on-premises site, even in the presence of a compromised on-premises replica, we can insert privacy firewalls (PFs) [75, 34] between the on-premises replicas and each of the two wide-area networks. The *cloud-side PF* filters messages sent from on-premises replicas to the cloud, and the *client-side PF* filters messages sent from the on-premises replicas to the clients. Since all messages that leave an on-premises site must be threshold signed, the privacy firewall implementation is simple: each privacy firewall knows the relevant public key and only forwards outgoing messages that have a valid threshold signature. A valid threshold signature ensures at least one *correct* replica in the on-premises site endorsed the message content.

We consider each privacy firewall to be a black box, which can be a complex configuration from prior work (e.g. [75, 34]) or a single node. Note that while a single-node privacy firewall may appear to be a single point of failure, since we consider a wide-area setting, availability already depends on the router for the site, and a privacy firewall could be integrated with the site router, so it does not meaningfully expand the system's attack surface (see Section 5.3 for details).[1]

5.3 Threat Model

Our basic threat model is detailed in Chapter 3. The main difference between the threat model of Decoupled Intrusion-Tolerant System (Chapter 5) and Partially Cloud-based BFT System (Chapter 4) (and also [15]) is that in Decoupled Intrusion-Tolerant System we have a *separate* threat model for Intrusion-Tolerant Application and BFT Replication Engine, whereas Partially Cloud-based BFT System and [15] have a *single* threat model for both. This is due to the decoupled architecture in Decoupled Intrusion-Tolerant System which enforces strict separation between the Intrusion-Tolerant Application and BFT Replication Engine, while the architectures in Partially Cloud-based BFT System and [15] integrates them together.

Due to the full separation of the Intrusion-Tolerant Application and the BFT Replication Engine, the number of tolerated server compromises and site disconnections, as well as the number of supported proactive recoveries, can be configured separately for each of them.

On-premises Threat Model: Simultaneously, in each on-premises site, f_o on-premises replicas can be compromised, and k_o on-premises replicas can be performing proactive recovery. At the same time, d_o on-premises sites can be disconnected.

Cloud Threat Model: Simultaneously, f_c cloud replicas can be compromised, k_c cloud replicas can be performing proactive recovery, and d_c cloud sites can be disconnected.

[1] If weaker confidentiality guarantees are acceptable, the privacy firewall can be left out of the architecture. In that case, our confidentiality guarantees are the same as in Chapter 4. In our specification, privacy firewalls forward incoming messages to replicas within a site, but we can remove them by having on-premises replicas directly join the relevant multicast group. In our implementation, we use Spines [68] for inter-site multicast.

5.3.1 Management Domain Failures:

Our Decoupled Intrusion-Tolerant System makes it possible to recover from a *management domain failure*, in which *all* on-premises replicas managed by a system operator lose their state. This threat model can capture relevant practical attacks such as ransomware: if a ransomware attack on the on-premises replicas causes them all to lose access to their state (because it has been maliciously encrypted), the replicas can be taken down, cleaned, restarted, and the latest state restored from the cloud replicas. The system will suffer an outage during the recovery, but can seamlessly resume operations without losing the results of any previously executed requests.[2]

We do not tolerate cloud domain failures, as recovering cloud replicas may not be able to establish the latest ordinal executed by an on-premises replica (e.g. only a single replica has executed the ordinal, but it then becomes unreachable). However, highly resilient systems can still be built by ensuring sufficient resilience of the cloud domain. Cloud service providers can deploy replicas across multiple data centers and specify the risk of simultaneous unavailability in their SLA. A specialized *intrusion-tolerance-as-a-service (ITaaS) provider* can increase management diversity by deploying replicas across infrastructure managed by different underlying cloud infrastructure providers. This type of cloud-of-clouds approach is also considered in other works [17, 19], which explicitly store data across multiple cloud providers (e.g. Amazon, Azure, etc).

5.3.2 Service Properties:

The definitions of the service properties is detailed in Chapter 3, but here we specify under what conditions they are met.

Safety (Definition 1, Section 3.3): Our system guarantees safety as long as no more than f_o on-premises replicas in each on-premises site and no more than f_c total cloud replicas are compromised simultaneously (once a compromised replica goes through proactive recovery and is restored to a correct state, it is no longer compromised). Note that as in

[2]Even if the attack resulted from executing one of the updates, since the cloud replicas do not execute updates, they are not affected by the same attack. Thus, the operator could recover on-premises replicas to the latest checkpoint, and manually omit the update that triggered the attack.

Chapter 4, the safety definition only includes on-premises replicas, as cloud replicas do not execute updates.

In general, we assume that correct (non-compromised) replicas follow the protocol correctly and do not lose their state. However, we also maintain safety in the case of an on-premises management domain failure (i.e. all on-premises replicas lose their state), as long as no more than f_c cloud replicas are compromised (and no correct cloud replicas lose their state). Compromised privacy firewalls cannot affect safety.

Liveness (Definition 2, Section 3.3): To guarantee liveness, we require that the conditions of both the on-premises and cloud threat models are met: at most f_o on-premises replicas per on-premises site and f_c total cloud replicas are compromised, at most k_o on-premises replicas per on-premises site and k_c cloud replicas are undergoing proactive recovery, and at most d_o on-premises sites and d_c cloud sites are disconnected from the network. We also require that the privacy firewalls are up and correct (note that we can tolerate failed or compromised privacy firewalls, but, since a failed/compromised firewall effectively disconnects its site from the network, such failures count against the d_o tolerated on-premises disconnections).

Liveness also requires that correct system components that are not in the disconnected sites are able to communicate successfully. Specifically, we require that all correct on-premises replicas in a given on-premises site are able to communicate with each other and with the privacy firewalls for that site; all cloud replicas are able to communicate with all other correct, not-disconnected cloud replicas and with the cloud-side privacy firewalls for the not-disconnected on-premises sites; and the client-side privacy firewall in each on-premises site is able to communicate with clients. Communication between the cloud replicas must meet any network synchrony requirements of the specific BFT protocol being used.

Complete Confidentiality (Definition 4, Section 3.3): We guarantee complete confidentiality even if an unlimited number of cloud replicas are compromised. We maintain this guarantee when up to f_o on-premises replicas per on-premises site are compromised, as long as the privacy firewalls are up and correct. If a privacy firewall and an on-premises replica in the same site are compromised, confidentiality can be violated.

Note that when an on-premises replica is compromised, it may be able to use side-

channel attacks to potentially reveal confidential information, but, in contrast to all prior work, our architecture can *only* leak information if such attacks are successful. The work in Chapter 4 cannot maintain complete confidentiality in the presence of compromised on-premises replicas; solutions in [75, 34] separate agreement from execution and uses privacy firewalls to keep state confidential in the execution replicas, but their architectures inherently reveal client information to agreement nodes, which we want to avoid; and in secret sharing based solutions [53, 71], replicas similarly communicate directly with clients, and execute state manipulation algorithms. We consider side-channel attacks outside the scope of this paper. However, the privacy firewall can provide mechanisms to make them more difficult (as discussed in [75]).

5.4 System Configuration

Our decoupled system architecture can be configured based on the threat model the operator wants to tolerate. For the on-premises threat model, we require the total number of on-premises sites $S_o \geq d_o + 1$, and the number of replicas in each on-premises site $n_o \geq 2f_o + k_o + 1$. This guarantees that at least one site with $f_o + 1$ correct replicas is always available, which is the minimum needed to generate valid threshold signatures.

The required number of cloud sites and replicas depends on the BFT protocol used, but for protocols that normally use $3f + 2k + 1$ replicas to withstand f compromises and k proactive recoveries (the most common setting), we adapt the replica distribution formula from [15]. We require the total number of cloud sites $S_c \geq 2d_c + 1$ and set the total number of cloud replicas $n_c = 3f_c + 2\left\lceil \frac{3f_c d_c + d_c + S_c k_c}{S_c - 2d_c} \right\rceil + 1$, with replicas distributed evenly across the sites. This guarantees that a quorum of cloud replicas is always available under our threat model. Note however, that if the n_c replicas do not divide evenly among the cloud sites, additional replicas may be needed (see Section 5.4.1 for details and derivation).

Figure 14 shows a configuration with 4 on-premises replicas in each of 2 sites and 12 cloud replicas distributed across 4 sites. This configuration simultaneously tolerates one compromise and one proactive recovery in the cloud domain, one compromise and one proactive

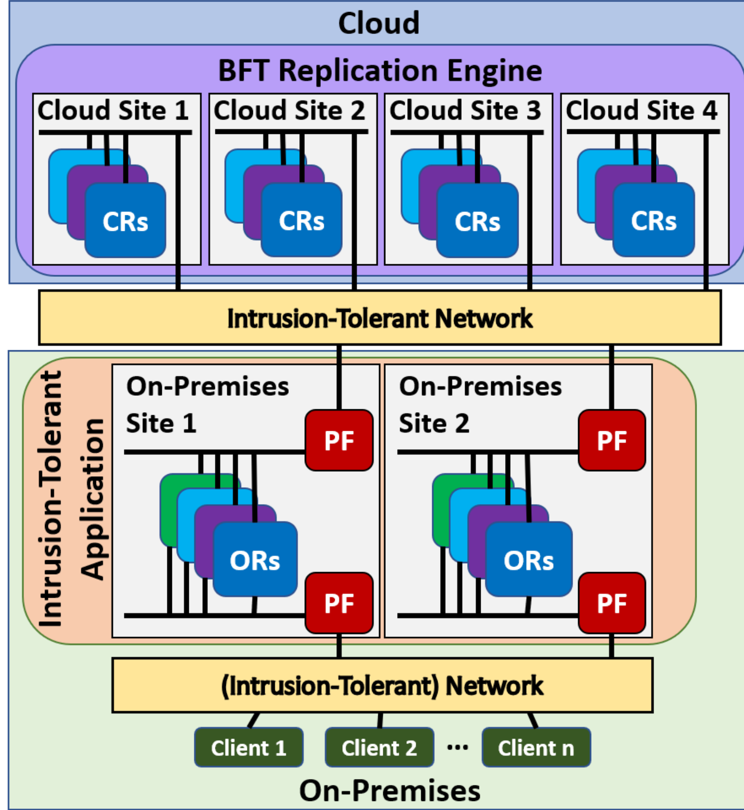


Figure 14: Decoupled Intrusion-Tolerant Architecture

recovery in each of the on-premises sites, one site disconnection in on-premises domain and one site disconnection in cloud domain ($f_o = f_c = d_o = d_c = k_o = k_c = 1$). To make site disconnections more difficult to execute successfully, we use an *intrusion-tolerant network* that uses an overlay approach to connect sites with redundancy [51, 15]. However, we make this optional for the on-premises network.

For system operators who currently use a primary and a backup site for fault tolerance, the configuration in Figure 14 offers the full resilience benefits of our architecture with minimal additional infrastructure. Operators only need to contract with a cloud provider and, assuming they have a primary and backup server in each site already, add two servers to each site. However, other options are possible. For example, if cloud sites have very high availability, we can consider a configuration with $d_c = 0$ that does not tolerate cloud site disconnections, but only requires one cloud site and may be useful for latency sensitive applications (see Section 5.7). Similarly, a configuration with $d_o = 0$ does not tolerate on-

premises site disconnections, but allows a system operator currently using a single site to gain intrusion tolerance and the ability to recover from management domain failures without constructing and managing any additional sites.

5.4.1 Calculating the Number of Required Cloud Replicas

To calculate the number of cloud replicas, we generalize the replica distribution formula from [15]. That work showed how to distribute replicas across sites to withstand exactly one site disconnection, while tolerating f intrusions and exactly one proactive recovery. We adapt this approach to tolerate any number of site disconnections, and any number of simultaneous proactive recoveries. We assume a BFT replication protocol that normally uses $3f + 2k + 1$ replicas to simultaneously withstand f intrusions and k proactive recoveries.

Similar to [15], we let the k parameter in the standard formula represent the total number of replicas that may be simultaneously *unavailable* (not only going through proactive recovery). For clarity, we use u to represent this total number of unavailable replicas, since we use k_c to represent the number of cloud replicas that may be going through proactive recovery. We let f_c represent the number of tolerated intrusions in the cloud, and let n_c represent the total number of cloud replicas. Therefore, in order for the BFT protocol to make progress, we require that $2f_c + u + 1$ out of $n_c = 3f_c + 2u + 1$ total replicas are correct, available, and connected.

If we assume replicas are distributed as evenly as possible across sites, then, to tolerate d_c site disconnections out of S_c total sites, in a system with n_c total replicas, we require:

$$u \geq d_c \left\lceil \frac{n_c}{S_c} \right\rceil + k_c \tag{1}$$

This guarantees that the tolerated number of unavailable replicas is at least the number of replicas in the d_c disconnected sites, plus the k_c replicas that may be down for proactive recovery.

To get a *lower bound* for the required value of u, we can drop the ceiling function and calculate:

$$u \geq d_c \left(\frac{n_c}{S_c} \right) + k_c \tag{2}$$

Substituting the formula for n_c into (2) gives us:

$$u \geq d_c \left(\frac{3f_c + 2u + 1}{S_c} \right) + k_c \tag{3}$$

Solving (3) for u, we get:

$$u \geq \frac{3d_c f_c + d_c + S_c k_c}{S_c - 2d_c} \tag{4}$$

Since we require u to be an integer (a whole number of replicas), we can apply the ceiling function and choose u as:

$$u = \left\lceil \frac{3d_c f_c + d_c + S_c k_c}{S_c - 2d_c} \right\rceil \tag{5}$$

To get a lower bound on the required number of replicas n_c, we can substitute the above lower bound for u into the formula $3f_c + 2u + 1$ to get:

$$n_c = 3f_c + 2 \left\lceil \frac{3d_c f_c + d_c + S_c k_c}{S_c - 2d_c} \right\rceil + 1 \tag{6}$$

In the case where the n_c resulting from equation (6) is evenly divisible by S_c, we can directly use this n_c as our number of cloud replicas, and distribute the replicas evenly across the S_c sites. This is guaranteed to satisfy the requirement in inequality (1). The reasoning for this is as follows: from inequalities (2)-(4) and equation (5), we have shown that setting u as in (5) satisfies inequality (2). When n_c is evenly divisible by S_c, the right-hand side of inequality (2) is exactly equal to the right-hand side of inequality (1). Thus, this choice of u also satisfies inequality (1) in this case.

However, when the n_c resulting from equation (6) is not evenly divisible by S_c, we are not guaranteed that inequality (1) is satisfied. In this case, we can calculate an upper bound on the required value of u, and then test each value of u between the lower bound and the upper bound to find one that satisfies inequality (1).

By the definition of the ceiling function, we know:

$$d_c \left\lceil \frac{n_c}{S_c} \right\rceil + k_c < d_c \left(\frac{n_c}{S_c} + 1 \right) + k_c \tag{7}$$

We want to find a value of u that is guaranteed to satisfy inequality (1). Based on inequality (7), we know the following choice of u is safe:

$$u = d_c \left(\frac{n_c}{S_c} + 1 \right) + k_c \tag{8}$$

Substituting the formula for n_c into (8) we get:

$$u = d_c \left(\frac{3f_c + 2u + 1}{S_c} + 1 \right) + k_c \tag{9}$$

Solving (9) for u, we get:

$$u = \frac{3d_c f_c + d_c + S_c d_c + S_c k_c}{S_c - 2d_c} \tag{10}$$

Since we require u to be an integer, we can apply the ceiling function:

$$u = \left\lceil \frac{3d_c f_c + d_c + S_c d_c + S_c k_c}{S_c - 2d_c} \right\rceil \tag{11}$$

Setting u as in (11) satisfies inequality (1). But, this is not necessarily the minimum value. To find the minimum integer value of u that satisfies (1), we consider every value of u between the values in (5) and (11):

$$\left\lceil \frac{3d_c f_c + d_c + S_c k_c}{S_c - 2d_c} \right\rceil \leq u \leq \left\lceil \frac{3d_c f_c + d_c + S_c d_c + S_c k_c}{S_c - 2d_c} \right\rceil \tag{12}$$

To find the number of required replicas n_c, we test each integer in this range as a possible value for u, and calculate n_c using the usual formula:

$$n_c = 3f_c + 2u + 1 \tag{13}$$

We start from the lower bound, and for each u and corresponding n_c, we check whether inequality (1) is satisfied. We choose the smallest u that satisfies this inequality, and use the corresponding n_c from equation (13) as the total number of replicas, distributing them as evenly as possible across the S_c sites.

Note that a clear implication of the lower bound (5) and upper bound (11) for u is that we require the quantity $S_c - 2d_c$ (the denominator in the formulas for u) to be strictly greater than 0. Thus, we require the total number of cloud sites:

$$S_c \geq 2d_c + 1 \tag{14}$$

5.5 Protocols for Decoupled Intrusion-Tolerant System

In order to support the system architecture described in Section 5.2, we must develop new protocols for handling client requests, and for performing state transfer and recovery.

5.5.1 Introducing New Client Requests

Figure 15 illustrates the steps involved in processing each client request. First, a client signs its request with its private key and sends it to the on-premises sites (step 1, Figure 15). If a client does not have the capability to do this on their own, a proxy can sign the request on the client's behalf [15, 46]. Once the request reaches an on-premises site, it is received by the client-side privacy firewall, which multicasts the request to all on-premises replicas in that site (step 2, Figure 15).

To enforce confidentiality of client requests, on-premises replicas encrypt the request using shared symmetric keys known to all on-premises replicas (but not known to the cloud replicas). In contrast to prior works that encrypt client requests (e.g. [75, 34], Chapter 4), the on-premises replicas encrypt not only the request body, but also the client headers so that the cloud replicas do not see client IDs and sequence numbers, and cannot easily learn client request patterns.

Next, on-premises replicas generate a threshold signature. Each replica encrypts the request, generates a partial signature share over it, and multicasts the signature share to the other replicas in its site. Upon collecting $f_o + 1$ partial signatures, the replica combines the partial signatures to generate a full threshold signature and sends the signed, encrypted

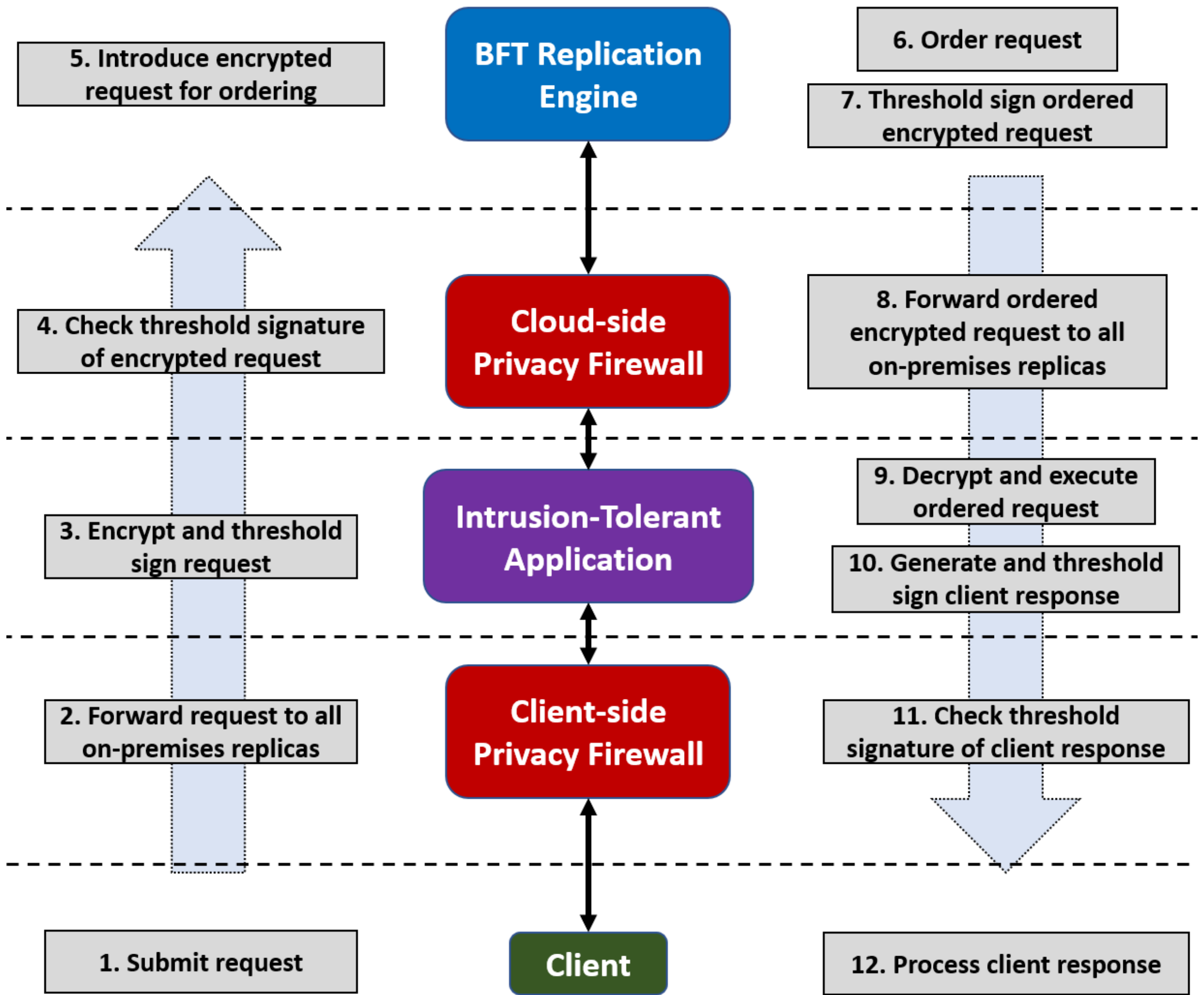


Figure 15: Client Request Flow in Decoupled Intrusion-Tolerant System

request to the cloud-side privacy firewall (step 3, Figure 15).

Note that to generate a full threshold signature, all on-premises replicas must generate partial signature shares over *identical encrypted content*. To ensure this, similar to the work in Chapter 4, each on-premises replica maintains two shared secret keys known to all on-premises replicas: one for symmetric encryption/decryption of client requests, and the other for generating initialization vectors via a pseudorandom function, similar to the approach in [34]. This allows all replicas to generate identical encrypted requests for a given client request. To avoid signing each request individually, it is possible to batch requests (see Section 5.5.3 for details.). However, for simplicity, we assume processing each request individually for our protocols in this chapter.

After verifying the threshold signature, which guarantees that at least one correct replica in the on-premises site endorsed the message content to be safe to leave the site (i.e., no leak of confidential data), the privacy firewall multicasts the signed encrypted request to the cloud replicas (step 4, Figure 15). Cloud replicas use the threshold signature to verify that the encrypted request is valid before introducing it for ordering by the BFT replication engine.

While the high-level process is relatively straightforward, fully encrypting and threshold signing client requests introduces new challenges: (1) we need to address a replay-attack vulnerability that encrypting client requests creates, and (2) we need a procedure for consistently updating encryption keys.

5.5.1.1 Preventing Replay Attacks

While encrypting the full client request (including headers) and threshold signing it enforces stronger confidentiality than previous approaches, this introduces a new vulnerability to *replay attacks*. A compromised on-premises replica can overload the BFT Replication Engine by storing threshold-signed encrypted requests and replaying them to the cloud replicas repeatedly. Since these *old* requests have valid threshold signatures, they will successfully pass through the privacy firewall and will be accepted as valid by the cloud replicas, causing them to waste processing resources and bandwidth ordering the duplicate requests.

Prior works have avoided this issue by using cleartext client IDs and sequence numbers in request headers to reject old/duplicate requests [75, 34] (also Chapter 4), but this exposes client information to the cloud replicas. A naive solution is for the cloud replicas to store a copy of every encrypted client request (e.g. in a hash table), and then use that to check for duplicates (when a new request arrives), which are then discarded. However, this is not practical as it requires unbounded memory (cloud replicas can never garbage collect old requests).

Validity Period. To address replays, we require on-premises replicas to append a *validity period* in cleartext to each encrypted request. The validity period is part of the content over which the threshold signature is generated (so a compromised replica cannot modify it). Cloud replicas store unique requests from the current validity period in a hash table, so they can discard requests from previous validity periods and duplicates from the current validity period.

To define a validity period for each encrypted request, simply allowing an on-premises replica to assign an expiration time (calculated using the current time at the on-premises replica) to the encrypted request will not work, since the calculated expiration time may differ for each on-premises replica for the same encrypted request. This is because we do not assume that clocks are synchronized, and requests can arrive at the on-premises replicas at slightly different times. Without identical expiration times, on-premises replicas cannot generate a threshold signature for the same encrypted request, as the expiration time must be part of the content over which the threshold signature is generated (otherwise, a compromised replica can modify it).

We can allow each on-premises replica to calculate the expiration time based on the timestamp inside the unencrypted client request, which will result in all on-premises replicas calculating identical expiration times. However, since the expiration time must be in cleartext for the cloud replicas, this will expose the client request timestamp to the cloud replicas, which violates our confidentiality guarantee.

Therefore, we develop a *novel* method to calculate identical validity periods across all on-premises replicas for the same encrypted request, while maintaining confidentiality of client request pattern. On-premises replicas determine the validity period based on the

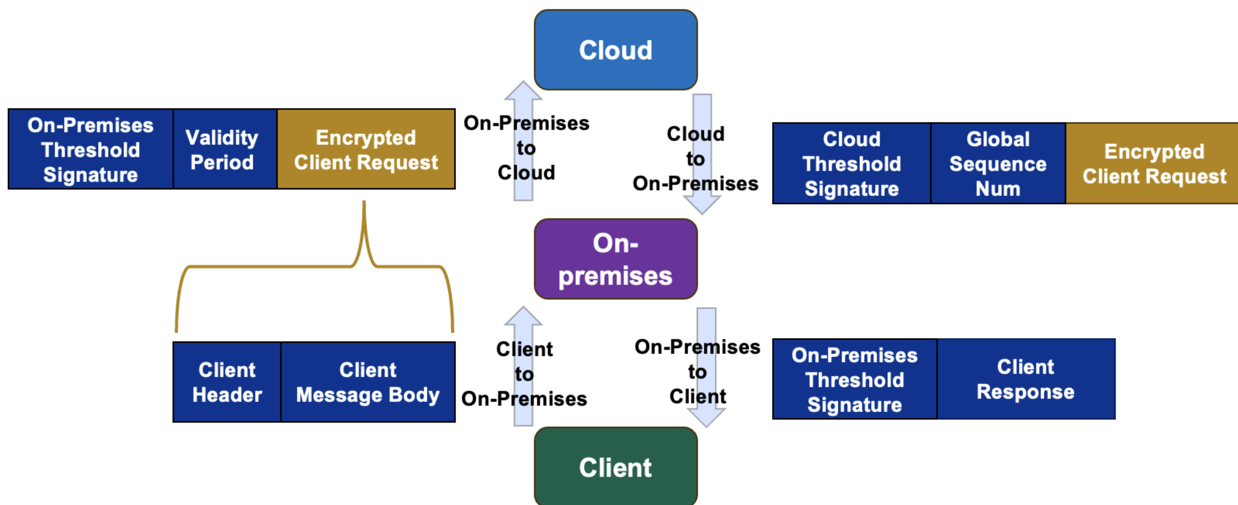


Figure 16: Message Formats in Decoupled Intrusion-Tolerant System

latest global sequence number ($lseq$) they have executed, and cloud replicas reject a request if the upper bound ($ubound$) of its associated validity period is less than the $lseq$ they have totally ordered. So that all on-premises replicas will typically have the same view of the validity period, we only update the validity period every vp_size sequence numbers. The lower bound ($lbound$) of the validity period is set as: $\lfloor \frac{lseq}{vp_size} \rfloor \times vp_size$, and the $ubound$ is set as: $lbound + (2 \times vp_size)$.

Ideally, the validity period size (vp_size) should be at least the maximum number of outstanding client requests. Smaller validity periods will not violate correctness, but can reduce performance, because on-premises replicas may assign validity periods that become stale by the time the request reaches the cloud replicas. With each client limited to one outstanding request, we can set vp_size to at least $M \times \lfloor \frac{n_o}{f_o+1} \rfloor \times S_o$ (where M is maximum number of clients, n_o is number of on-premises replicas per site, and S_o is number of on-premises sites). Note that there can still be brief periods where on-premises replicas disagree on the validity period (because they execute requests at slightly different times). However, this does not affect liveness, since a replica will retransmit its partial signature share for a client request (with an updated validity period) if the request is not executed before a timeout.

5.5.1.2 Updating Encryption Keys

Privacy firewalls prevent encryption key leakage, but periodic key refreshes are still needed to be able to recover from potential side-channel attacks or a malicious operator with physical access copying keys. To ensure that all on-premises replicas know which key to use to decrypt each request, we tie key changes to the validity period (since it is the only available cleartext information). When the end of the validity period approaches, each on-premises replica generates a new key proposal signed by a persistent hardware-based key (e.g. using the TPM) and submits it for ordering. The key for the next validity period is determined based on the ordered key proposals, similar to Chapter 4 (although they based the key-change interval on client sequence numbers that are not available in cleartext for us).

5.5.2 Ordering and Executing Client Requests

Upon receiving a new valid client request (i.e., one that is within the validity period, not a duplicate, and has a valid threshold signature), cloud replicas inject the request into the BFT replication engine (step 5, Figure 15). This executes the BFT agreement protocol to assign the request a global sequence number or *ordinal* (step 6, Figure 15). Cloud replicas threshold-sign the ordered encrypted request (step 7, Figure 15), and then multicast it to the cloud-side privacy firewalls, which forward it to the on-premises replicas (step 8, Figure 15).

Requests may occasionally arrive out-of-order due to network disruptions, so on-premises replicas use a sliding window buffer to maintain ordering. Upon receiving the next expected ordinal, the on-premises replica executes the corresponding request (step 9, Figure 15), generates a response, and cooperates to create a threshold signature (step 10, Figure 15). This response is sent to the client through the client-side privacy firewall (step 11, Figure 15). The client (or accompanying proxy) validates the correctness of the response by verifying the threshold signature (step 12, Figure 15).

The message formats for the client request flow in our Decoupled Intrusion-Tolerant System is shown in Figure 16.

5.5.3 Batching of Client Requests

For simplicity we assume each request is processed individually for our protocols in this chapter. However, we would still like to describe how we can achieve *batching* of requests in this section. To generate threshold signatures on new client requests efficiently, it is important to be able to *batch* requests, such that replicas do not need to sign every request individually. Unfortunately, client requests may not arrive in the same order at every on-premises replica, so replicas may not generate identical batches (and requiring replicas to agree on the batch contents is essentially equivalent to running the agreement protocol). Without identical batches, the partial signature shares generated over these batches will not combine correctly. Therefore, we allow replicas to contribute partial signatures to batches received from other replicas as described below.

Each on-premises replica batches received client requests with a limit on the maximum count and/or time, sorting batched requests by their client IDs. Next, the on-premises replica encrypts the batch, generates a partial signature over it, and sends the encrypted batch (with its partial signature) to all other on-premises replicas in its site. Upon receiving a batch of client requests, an on-premises replica decrypts it and verifies each of the client requests with the respective client's public key. Once the entire batch is verified, the on-premises replica generates a partial signature for the encrypted batch and sends it back. Upon collecting $f_o + 1$ partial signatures (including its own), an on-premises replica can generate a threshold signature. It sends the encrypted batch (with the threshold signature) to the cloud replicas through the cloud-side privacy firewall, which verifies the threshold signature before forwarding it.

To optimize the batching process, when an on-premises replica's batch of encrypted requests matches that of another on-premises replica's, it uses the accompanying partial signature from the other on-premises replica for its own batch of encrypted requests. Since we require the client requests inside a batch to be sorted by client IDs, and there is typically very little delay variation or chance of message loss within a site, we can expect that new batches from different on-premises replicas within a site to almost always match, and hence threshold signatures can be generated quickly.

Note that we do not need to change our validity period procedure for batching requests. Since we limit each client to one outstanding request at a time, in the worst case scenario, a malicious replica can space out the client requests to one per batch, but the validity period takes into account the total number of clients. Hence, the malicious replica will not be able to quickly fill up the validity period and slow down processing of new requests.

The cloud replicas treat this encrypted batch of requests same as a single encrypted request (check validity period and threshold signature, order the batch, threshold sign the ordered encrypted batch of requests, and finally send this back to the on-premises replicas). Upon receiving a ordered encrypted batch of requests, the on-premises replica checks the threshold signature, decrypts the batch, and then executes each request in the same order as they are in the batch. Since each batch has an ordinal number, batches are processed consecutively according to their ordinal numbers.

5.5.4 Checkpoints and Nearest-First Recovery

As discussed in Section 5.2, to enable recovery from network attacks and management domain failures, cloud replicas store encrypted state checkpoints and any encrypted requests that have been ordered since the latest checkpoint. On-premises replicas can request this encrypted state to recover from state loss or prolonged disconnections.

However, as also discussed in Section 5.2, *every* message leaving an on-premises site must be threshold signed. This requires new protocols for checkpointing and recovery across management domains. Two important challenges are: (1) Only threshold-signed recovery requests can be sent from on-premises replicas to the cloud replicas. This requires on-premises replicas to agree on which requests to send to the cloud. (2) Because client responses must also be threshold-signed, on-premises replicas must be able to recover these signatures in order to serve client retransmissions.

We address these issues with a new nearest-first recovery protocol that first attempts to perform recovery within a site, and only sends (threshold-signed) recovery requests outside the site if in-site recovery is unsuccessful. The protocol also enables on-premises replicas to collect the threshold-signed client response corresponding to each request it recovers. This

strategy is necessary under our system model, but also improves recovery latency and wide-area bandwidth usage by localizing state transfer as much as possible. Below we describe the checkpointing and recovery protocols.

5.5.4.1 Checkpoint Creation

Each on-premises replica periodically generates a checkpoint representing its current state (including the latest response for each client), encrypts it, and then cooperates with other on-premises replicas in its site to create a threshold signature. The replica stores the signed encrypted checkpoint and multicasts it to the cloud replicas, which can verify the threshold signature and then store the encrypted checkpoint. Any replica can safely remove ordered encrypted requests older than the currently stored checkpoint.

5.5.4.2 Nearest-First Recovery

Recovery begins when a replica detects that it is missing ordered requests, e.g., due to a site disconnection, crash, or proactive recovery.

Requesting Recovery. An on-premises replica triggers recovery when it receives an ordered request beyond the upper bound of its sliding window buffer from the cloud replicas. The *recovering replica* separately requests missing ordered encrypted requests and client responses. For each, it sends a request with a list of ordinals (global sequence numbers) that it is missing to the other on-premises replicas in its site.

Responding to a Recovery Request. Upon receiving a recovery request, an on-premises replica will respond with its stored encrypted checkpoint if any ordinal in the recovery request is older than the checkpoint. Otherwise, the replica sends all of the requested ordered encrypted requests or client responses it has to the recovering replica. It also sends the associated threshold signatures for client responses; if it does not yet have the threshold signature for a client response, it sends its partial signature share instead. To prevent malicious replicas from wasting resources, all replicas rate-limit their responses to repeated recovery requests from the same replica.

Applying Recovery Responses. Upon receiving client responses, a recovering replica

simply stores them. Upon receiving a checkpoint that is newer than its current state, the recovering replica verifies the threshold signature, and decrypts and applies the checkpoint to its local state (the latest client responses in the checkpoint are extracted and stored).

Upon receiving new ordered encrypted requests, the recovering replica decrypts and executes them (after verifying their threshold signatures) consecutively based on the ordinals. If it already collected a client response with threshold signature for the executed ordered request, then it simply stores this client response and moves on to the next ordinal. Otherwise, the recovering replica generates the client response, sends a partial signature to other on-premises replicas in its site, and waits to collect $f_o + 1$ partial signature shares (including its own). By applying a checkpoint and/or executing requests, the recovering replica eventually catches up to the latest state.

Recovering an On-Premises Site. If all the replicas in a site are missing the *same* ordered encrypted requests (which they will eventually find out), then they can generate a threshold signature for the recovery request and send it to the cloud replicas. To do this, every recovery request for ordered encrypted requests includes a validity period and a partial signature share over the request; the validity period is based on the ordinal of the latest ordered encrypted request received from the cloud replicas. On receiving such a request, a cloud replica multicasts the requested ordered encrypted requests or encrypted checkpoint to the on-premises replicas.

To mitigate replay attacks by a malicious replica (which may re-send an old threshold-signed recovery request), cloud replicas rate-limit their responses to recovery requests. New recovery requests are not subject to this rate limit. If a cloud replica receives a request that is in the current validity period, and is not a duplicate (checked with hashes of other requests in the current validity period), then it responds immediately.

Recovery Procedure in Cloud: Cloud replicas use a similar nearest first recovery strategy when they detect a gap in the global sequence numbers from the underlying BFT protocol. However, they do not store or request client responses.

5.6 Implementation

We have implemented Decoupled Spire, a SCADA system for the power grid, based on the open source Spire version 1.2 [69]. Spire 1.2 uses an integrated architecture in which all replicas fully participate in the BFT replication protocol, maintain application state, and execute requests [15].

5.6.1 Decoupled Spire Components

In Decoupled Spire, the on-premises sites host replicas of the SCADA master application. The clients are Programmable Logic Controllers (PLCs), Remote Terminal Units (RTUs), and Human Machine Interfaces (HMIs); we use the HMI and emulated PLCs/RTUs available in Spire 1.2. Like Spire 1.2, we use Prime [10, 56] as the BFT replication engine. We add a single-node *privacy firewall* to these components. Privacy firewalls run in each on-premises site and use the public service key for that site to verify threshold signatures on all outgoing messages. We use Spines [68] for our intrusion-tolerant networks: one Spines network connects all the cloud replicas to each other and the cloud-side privacy firewalls, and a second Spines network connects the clients to the client-side privacy firewalls. Inside each on-premises site, replicas communicate using UDP over a switched LAN (with application-level retransmissions).

5.6.2 Separating Agreement and Execution

In contrast to Spire 1.2, our SCADA master replicas do not participate in the Prime replication protocol. Instead, each SCADA master is linked with a simple intrusion-tolerance layer that *prepares* each client request for ordering, sends it to the cloud replicas, and then receives, buffers, and verifies signatures on incoming ordered updates. Preparing a client request for ordering involves encrypting it, appending a validity period, and generating a threshold signature on it. Our implementation of encryption is similar to that in Chapter 4, which is based on [34]. We use the client request and a pseudo-random function key (refreshed each validity period and shared by all on-premises replicas, as discussed in Section 5.5.1.2)

to generate a hash-based message authentication code (HMAC). This HMAC is used as the initialization vector (IV), along with the shared encryption key, to encrypt the *entire* client request using AES-256 in CBC mode. This encrypted request is accompanied by a clear-text header that includes the validity period and IV, and a threshold signature covering the header and the encrypted request.

5.7 Evaluation

The main benefit of Decoupled Spire is its clean separation of the cloud and on-premises domains, which simplifies management while supporting a strong threat model. Here, we quantify the performance overhead of this separation (the main tradeoff for system operators) by comparing Decoupled Spire with Spire 1.2 [15] and Confidential Spire (Chapter 4). We show that this tradeoff is acceptable for this latency-sensitive application.

All experiments are done using a local area network with emulated latencies between sites that reflect a realistic geographic distribution that spans 250 miles of the US East Coast (similar to [15]). This corresponds to emulated latencies of 2 to 5 ms between each pair of sites. We emulate ten power grid substations, where each introduces a new request every second for a total of 36,000 requests in a one hour period.

5.7.1 Normal Operation Performance ($f = 1$)

Table 4 shows client request latencies over a one-hour experiment for each configuration. We can see that Decoupled Spire ($f_o = 1, f_c = 1, d_o = 1, d_c = 1$), which tolerates one compromise, one proactive recovery and one site disconnection in both on-premises and cloud domains, has an average latency of 58.9ms, compared to about 50ms for Spire 1.2 ($f = 1$) and Confidential Spire ($f = 1$)[3], for an overhead of about 9ms (18%). The overhead comes from the additional wide-area communications on the critical path of request processing: in Decoupled Spire, on-premises replicas must send each client request to the cloud replicas

[3]In our experiments, the overhead of Confidential Spire compared to Spire 1.2 is smaller than the one reported in Chapter 4, likely due to hardware differences.

Table 4: Spire, Confidential Spire and Decoupled Spire normal operation performance for 36,000 updates over 1 hour

	Avg Latency	0.1^{st} percentile	1st percentile	99th percentile	99.9^{th} percentile
Decoupled Spire ($f_o = 1,\ f_c = 1,\ d_o = 0,\ d_c = 0$)	41.8 ms	27.9 ms	30.1 ms	53.1 ms	54.8 ms
Decoupled Spire ($f_o = 1,\ f_c = 1,\ d_o = 1,\ d_c = 0$)	41.9 ms	27.7 ms	30.1 ms	53.1 ms	54.9 ms
Decoupled Spire ($f_o = 1,\ f_c = 1,\ d_o = 0,\ d_c = 1$)	58.7 ms	46.6 ms	48.0 ms	69.5 ms	71.3 ms
Decoupled Spire ($f_o = 1,\ f_c = 1,\ d_o = 1,\ d_c = 1$)	58.9 ms	46.9 ms	48.1 ms	69.6 ms	71.3 ms
Confidential Spire (Chapter 4) ($f = 1$)	50.1 ms	38.4 ms	39.6 ms	60.9 ms	63.5 ms
Spire 2018 [15] ($f = 1$)	49.9 ms	38.2 ms	39.2 ms	60.5 ms	62.2 ms
Decoupled Spire ($f_o = 2,\ f_c = 1,\ d_o = 1,\ d_c = 1$)	58.9 ms	46.9 ms	48.2 ms	69.7 ms	71.6 ms
Decoupled Spire ($f_o = 1,\ f_c = 2,\ d_o = 1,\ d_c = 1$)	60.5 ms	48.5 ms	49.5 ms	71.7 ms	75.2 ms
Decoupled Spire ($f_o = 2,\ f_c = 2,\ d_o = 1,\ d_c = 1$)	62.0 ms	49.4 ms	50.6 ms	74.3 ms	78.2 ms
Confidential Spire (Chapter 4) ($f = 2$)	56.5 ms	42.2 ms	43.7 ms	69.8 ms	73.8 ms
Spire 2018 [15] ($f = 2$)	53.4 ms	39.6 ms	41.1 ms	64.1 ms	67.9 ms

before it is introduced for ordering, whereas, in Spire and Confidential Spire, the control center replicas directly inject requests via their local BFT replication instances. In our experimental setting, this extra wide-area delay adds a total of 4ms to the processing of each request. The remaining 5ms is due to the additional processing done by the on-premises and cloud replicas (more messages, encryption, decryption, duplicate check, and threshold signing and verification).

Decoupled Spire ($f_o = 1, f_c = 1, d_o = 0, d_c = 1$), which does not tolerate any on-premises site disconnection, has almost exactly the same overhead as Decoupled Spire ($f_o = 1, f_c = 1, d_o = 1, d_c = 1$), since it has the same additional wide-area communications and processing in the critical path.

Interestingly, Decoupled Spire ($f_o = 1, f_c = 1, d_o = 1, d_c = 0$), which does not tolerate any cloud site disconnection, achieves about 8ms (16%) *less* latency compared to Spire 1.2 ($f = 1$) and Confidential Spire ($f = 1$), and 17ms (29%) less latency compared to Decoupled

Spire ($f_o = 1, f_c = 1, d_o = 1, d_c = 1$). This is because the entire agreement protocol runs in a single cloud site: even with the additional wide-area delays for sending the request to the cloud and back, eliminating wide-area communication overhead in the agreement protocol results in lower total latency. For latency-sensitive applications, this may be attractive if service providers can guarantee near 100% uptime for their cloud sites and effectively bolster them against DoS attacks. Decoupled Spire ($f_o = 1, f_c = 1, d_o = 0, d_c = 0$), which does not tolerate any site disconnection, has similar performance, since it also uses just one cloud site.

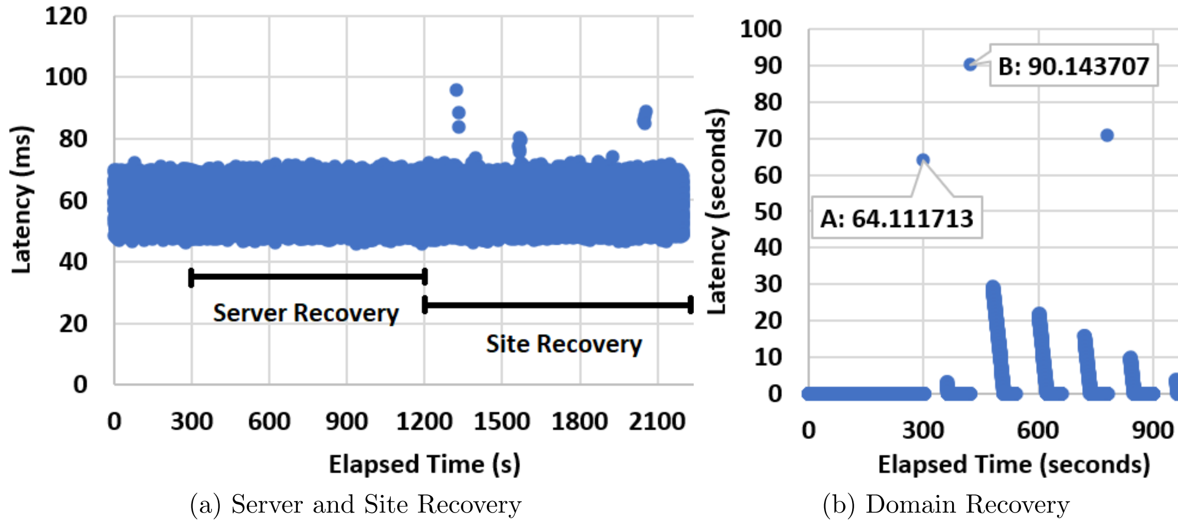
5.7.2 Increasing the Number of Tolerated Intrusions

In Decoupled Spire, increasing f_o from 1 to 2 (lower half of Table 4) has negligible effect on latency, since most of the additional communication happens inside the on-premises site over a LAN. This can be useful for system operators who want to increase the resiliency in their on-premises sites (which may be less protected) without needing any higher resiliency in the cloud (which may be better protected).

Table 4 also shows that Decoupled Spire with $f_o = f_c = 2$ increases latency by about 3.1ms compared to Decoupled Spire with $f_o = f_c = 1$. Interestingly, Spire 1.2 also adds about 3.3ms when increasing f from 1 to 2. This is because most of the increase comes from the increased wide-area communications in the BFT agreement protocol which is about the same for both systems (both increase the number of BFT replicas from 12 to 19). Confidential Spire shows a larger increase (6.4ms) when f is increased from 1 to 2, as the number of replicas in the BFT protocol increases from 14 to 21.

5.7.3 Performance during Failures and Recovery

We evaluated the performance of Decoupled Spire ($f_o = 1, f_c = 1, d_o = 1, d_c = 1$) while recovering after a failure or an attack. We emulated this in 1-hour long experiments by repeatedly killing and restarting the SCADA application of a single server (for server recovery), all servers in an on-premises site (for site recovery), or all servers in both on-premises sites (for domain recovery). For each experiment, we kill the SCADA application, wait 1 minute, restart the SCADA application, again wait 1 minute, then repeat. A restarted



(a) Server and Site Recovery

(b) Domain Recovery

Figure 17: Performance of Decoupled Spire During Attack Recovery

server must collect the latest state from other replicas, so it emulates the proactive recovery process after state has been corrupted by an attacker.

Our SCADA application requires client request latencies within a 100ms threshold under normal operations, and can tolerate up to 200ms for a few requests [1]. Figure 17a shows that our Decoupled Spire achieves just that since no request crosses 100ms during server or site recoveries. This is because during server recovery, the entire recovery process happens within the on-premises site, while the path for processing client requests through the other on-premises site remains unaffected. We see a few small spikes in latencies (but none over 100ms) during site recovery, since the recovering replicas need to pull the latest encrypted checkpoint and encrypted ordered requests from the cloud replicas which generates extra wide-area network traffic that affects request latencies.

During a domain recovery, we see large latency spikes while both on-premises sites are recovering, but the performance quickly returns to normal as soon as either on-premises site finishes recovering. Figure 17b shows a few requests with very high latencies (e.g. 64s for point A and 90s for point B). This is because those requests were submitted just before both the on-premises sites went down, and hence were ordered and held by the cloud replicas until the on-premises sites recovered.

Table 5: Spire, Confidential Spire and Decoupled Spire diversity analysis (number of diverse variants)

	Number of Diverse Variants		
	App	BFT	On-Premises (App/BFT)
Decoupled Spire ($f_o = 1,\ f_c = 1,\ d_o = 1,\ d_c = 1$)	4	12	4 / 0
Confidential Spire (Chapter 4) ($f = 1$)	8	14	8 / 8
Spire 2018 [15] ($f = 1$)	12	12	6 / 6
Decoupled Spire ($f_o = 2,\ f_c = 2,\ d_o = 1,\ d_c = 1$)	6	19	6 / 0
Confidential Spire (Chapter 4) ($f = 2$)	12	21	12 / 12
Spire 2018 [15] ($f = 2$)	19	19	10 / 10

5.7.4 Discussion on Throughput

In our current implementation, the throughput of the system is primarily limited by the BFT replication engine (in our case, Prime [10, 56]), so we do not focus on throughput in the evaluation. However, there are three additional factors in our architecture that can impact throughput: threshold signing for new client requests at on-premises replicas, threshold signing ordered encrypted updates at cloud replicas, and threshold signing client responses. Of these, the threshold signing of ordered encrypted updates is the only new addition compared to Confidential Spire (Chapter 4).

5.7.5 Diversity Analysis

To support failure independence assumptions, each instance of the application, and each instance of the BFT engine should be different from all other instances. Decoupled Spire significantly reduces the number of diverse variants required for the application. Because we tolerate f_o compromises per on-premises site, a system operator can set up diverse replicas in

one site, and then simply duplicate that same setup for their additional site(s). In contrast, Confidential Spire requires that all application replicas are diverse, and Spire runs an application instance at *every* replica, requiring a much larger number of diverse variants. While we require about the same number of variants for the BFT replication engine as previous systems, the service provider approach has an important benefit here: the service provider can use the same set of (diverse) replicas to serve many applications, which can help make it cost-effective to invest in expensive diversity techniques (e.g. N-version programming).

Table 5 summarizes the number of required diverse application and BFT engine variants in each architecture, as well as how many of those variants need to be deployed on-premises. For example, for the case of $f = 1$, we require only 4 diverse application variants compared to 8 for Confidential Spire or 12 for Spire. The total number of diverse components that need to be deployed on-premises is dramatically reduced, since both Spire and Confidential Spire require BFT replicas (with diverse variants) to be deployed on-premises.

6.0 Optimizing Deployment of Intrusion-Tolerance as a Service

6.1 Overview

In Chapter 5, we focused on enabling system operators to deploy intrusion-tolerant applications while fully outsourcing the responsibility for the BFT replication protocol to a cloud service. We envision the cloud service to be made up of two separate stakeholders: an *infrastructure provider*, and an *Intrusion-Tolerance as a Service (ITaaS) provider*. The infrastructure provider offers cloud resources as a service, and there are several infrastructure providers that exist today, e.g. Amazon Web Services (AWS) offers Elastic Compute Cloud (AWS EC2) [63] and Equinix [36] offers colocation in their data centers.

We envision the ITaaS provider to be a new entity that builds our Intrusion-Tolerance as a Service, including the BFT Replication Engine, on top of the cloud resources provided by the infrastructure provider. Currently, infrastructure providers offer a range of geographically distributed cloud resources, e.g. virtual machines (VMs), dedicated servers, and data center colocation spaces, with different levels of access/control for their users. Therefore, based on the specific cloud resources that the ITaaS provider builds their technology on and the level of trust/control the ITaaS provider has with the infrastructure provider, the ITaaS provider will choose a service model to offer their respective customers, the system operators.

In this chapter, we introduce three service models that the ITaaS provider can offer the system operators, where each service model is based on the type of cloud resources being used and the level of trust between the ITaaS provider and infrastructure provider. However, this raises the question of how to make each of these service models cost-effective for the ITaaS providers so they can successfully provide "intrusion-tolerance as a service".

Based on our discussion in Section 2.5, an ITaaS provider's focus will be to minimize cost while maintaining the required service level agreements (SLAs), which can be achieved by designing a scalable service. Hence, to make it more feasible to offer "intrusion-tolerance as a service", the ITaaS provider would want to maximize the number of system operators' intrusion-tolerant applications that they can support by efficiently distributing their replicas

across cloud resources while minimizing the number of cloud resources needed, without violating the SLAs for any application.

Specifically, our research question is: *How can we efficiently optimize the placement of replicas from diverse applications across a range of cloud resources, while guaranteeing safety and liveness, and supporting proactive recovery?*

In this chapter, we investigate new challenges of optimizing the deployment of cloud replicas supporting different applications from separate system operators in the same physical machines for scalability while still ensuring safety and liveness. For each service model, we develop heuristic optimization algorithms and Mixed-Integer Linear Programming (MILP) formulations for optimal solutions. Then we evaluate these algorithms in terms of feasibility, efficiency and cost. While these metrics vary significantly depending on the algorithm and Service Model employed, optimal solutions and select heuristic algorithms consistently exhibit high effectiveness in minimizing cloud resource usage and overall costs, particularly in scenarios involving a large number of applications, all while meeting the necessary application requirements.

The main contributions of this chapter are:

- We show how an entity can offer *Intrusion-Tolerance as a Service (ITaaS)* by building on top of cloud resources provided by an infrastructure provider. We define three service models based on the type of cloud resources being used, and the level of trust between the ITaaS provider and the infrastructure provider.
- We design and implement a framework for optimizing the distribution of replicas of different applications across shared cloud resources, while guaranteeing safety, liveness, and supporting proactive recovery. We develop heuristic algorithms, and Mixed-Integer Linear Programming (MILP) formulations for this framework.
- We evaluate our optimization framework in terms of feasibility, efficiency, performance and cost analysis. We show that optimal solutions and select heuristic algorithms consistently exhibit high effectiveness in minimizing cloud resource usage and overall costs, particularly in scenarios involving a large number of applications, all while meeting the necessary application requirements.

6.2 Challenges of Optimizing Intrusion-Tolerance as a Service

In this section we discuss the challenges in optimizing Intrusion-Tolerance as a Service (ITaaS). While traditional cluster scheduling algorithms focus on distributing tasks belonging to jobs across a cluster of machines, our concern lies in the placement of replicas for an intrusion-tolerant application. The primary distinction arises from the nature of these replicas, which run continuously (except when temporarily halted for proactive recovery), unlike jobs or tasks that eventually terminate. Thus, algorithms designed for scheduling jobs/tasks in distributed clusters are not well suited for our problem. However, there are scheduling algorithms for optimizing the placement for continuously running processes, such as web servers [58], but they do not address the new challenges essential for our placement algorithms (see details in Section 6.2.2).

6.2.1 Limitation on Sharing Physical Machines

The first challenge for our replica placement problem is that replicas that run on the same physical machine are subject to shared vulnerabilities. If the number of replicas of a given application running on a single machine exceeds the tolerated fault threshold f for that application, then a compromise of that machine can violate the system's safety guarantees (and a crash would violate liveness guarantees). This challenge is not unique to our optimization problem, since there are *fault-tolerant scheduling* algorithms [76, 47] which specifically distribute replicated services over separate physical machines. However, this shared vulnerabilities constraint along with our new challenges (as detailed in Section 6.2.2), require us to develop new strategies for optimizing intrusion-tolerance as a service.

We address this shared vulnerabilities challenge by limiting at most one replica per application on each physical machine. This way, even if the physical machine is compromised, the attacker can compromise *at most one* replica per application (which does not exceed the threat model for each application). Given this constraint, we assume that the ITaaS provider can safely deploy multiple replicas, each isolated in its own Virtual Machine (VM), from different applications on a single physical machine.

6.2.2 New Challenges

Our problem of deploying cloud replicas to support intrusion tolerance as a service brings new considerations that differentiate it from traditional cluster scheduling problems.

6.2.2.1 Calculation of Number of BFT Replicas

To address the specific requirements of our network-attack-resilient BFT system in the cloud, we introduced a novel formula for calculating the number of replicas, as detailed in Section 5.4.1. This formula is essential for accommodating not only system compromises but also proactive recoveries and site disconnections. Therefore, any scheduling algorithm for deployment of cloud replicas of a network-attack-resilient BFT system must incorporate our formula, which takes into account variable numbers for each of the three items: f compromises, k proactive recoveries, and d site disconnections.

Traditional fault-tolerant scheduling approaches typically tolerates only *non-Byzantine* faults [76, 47]. Byzantine fault tolerance necessitates a different approach. This stems from the distinct nature of Byzantine faults, where a malicious component may respond with incorrect or malicious actions, in addition to failing to respond or responding late. Some existing scheduling algorithms may address fault tolerance in Byzantine concensus-based systems [12], but none incorporate the specific considerations for network-attack-resilient BFT systems, making our formula unique in this regard.

6.2.2.2 Latency Constraint for BFT Replication Protocol

To ensure the viability of intrusion-tolerant applications, especially those critical ones like power grids, pipelines, and healthcare systems, it's crucial to consider their strict latency requirements. These applications must function seamlessly even under the threat of cyber-attacks [33, 52, 2]. Given the necessity of Byzantine Fault Tolerant (BFT) replication for maintaining operation integrity amidst compromise [21], it's imperative to assess the BFT replication protocol's performance within these latency constraints.

The performance of the BFT replication protocol is heavily influenced by wide-area-

network latency between geographically distributed sites. Therefore, for each intrusion-tolerant application, we need to pick sites (for distributing the cloud replicas) that does not exceed the *maximum expected latency* while running the BFT replication protocol for that application. This BFT replication protocol latency constraint is new to our problem of optimizing intrusion tolerance as a service, and is not addressed by traditional cluster scheduling algorithms. Therefore, it presents a new challenge that we need to address while developing scheduling algorithms.

6.2.2.3 Scheduling Proactive Recovery Across Applications

One more new challenge in our optimization problem is that we need to support proactive recovery of the intrusion-tolerant applications while still maintaining the required performance and SLAs for each application. This also is not a concern for existing cluster scheduling algorithms. Proactive recovery is essential for long system lifetimes, as it enables replicas to be taken down periodically and refreshed to a known good state; however, it requires an out-of-band mechanism for triggering recovery, since a compromised replica will not voluntarily recover itself. The authors in [55] considered deployments where the entire system is under one management domain and the operator has full access to the hardware, so they performed proactive recovery by automatically cycling the power using netbooter devices. The same authors in [55] also consider a virtualized version, but the tradeoff is that this requires trusting the hypervisor.

To address this new challenge, we introduce three service models (discussed in the next section) that use different types of cloud resources, and require different levels of trust between the ITaaS provider and the infrastructure provider. Service Model 1 assumes virtual machines (VMs) are provided by the infrastructure provider, and the ITaaS provider trusts the infrastructure provider to properly isolate their VMs from other VMs, which can be from other customers, running on the same physical machine, and to properly shutdown and restart the ITaaS provider's VMs for proactive recovery.

Service Model 2 assumes dedicated servers are provided by the infrastructure provider,

and even though the ITaaS provider has control over what VMs are run in each dedicated server, they still need to trust the infrastructure provider's hypervisor to properly shutdown and restart each VM for proactive recovery (similar to the work in [55]).

Service Model 3 assumes data center colocation spaces are provided by the infrastructure provider, and the ITaaS provider is in charge of setting up and maintaining their own servers in those spaces. Hence, this requires the least amount of trust between the ITaaS provider and the infrastructure provider. We also assume that the hypervisor can be malicious, so proactive recovery in this Service Model requires the ITaaS provider to completely shutdown and restart each physical server (e.g., by using netbooters similar to the work in [55]), including the hypervisor and all the VMs running on it. However, since each of these VMs belong to a separate application, the ITaaS provider must deploy VMs with the same *proactive recovery schedule* in the same physical machine.

The proactive recovery schedule is a predetermined timetable for an application, outlining when each replica undergoes proactive recovery at specified times. This schedule is determined by various characteristics of the application, specifically, the number of tolerated compromises, proactive recoveries, site disconnections, and the interval for proactive recovery.

6.3 Service Models

To address the challenges of optimizing the placements of replicas on servers across multiple sites, we introduce three different service models for our Intrusion-Tolerance as a Service (ITaaS), based on the type of cloud resources used, and the level of trust between the ITaaS provider and the infrastructure provider. The main difference among the service models involves the tradeoffs between the degree of trust the ITaaS provider needs to have in the infrastructure provider, the amount of physical infrastructure the ITaaS provider needs to manage, and the cost for the ITaaS provider. It is useful to consider all three of these service models because different ITaaS providers may prefer various types of cloud resources provided by the infrastructure provider, depending on factors such as availability, cost, level

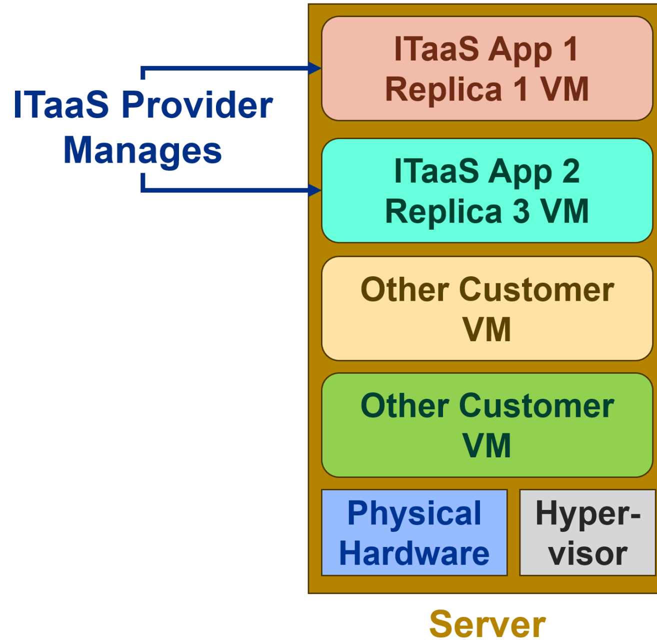


Figure 18: Service Model 1: Virtual-Machine-based ITaaS

of trust, and manageability of physical resources.

6.3.1 Service Model 1 (SM1): Virtual-Machine-based ITaaS

In Service Model 1, the ITaaS provider reserves virtual machines (VMs) from an infrastructure provider (Figure 18). Each replica of the intrusion-tolerant application that the ITaaS provider deploys is instantiated in a separate VM, with the infrastructure provider overseeing its assignment to a physical machine based on infrastructure provider's resource availability and optimization criteria. The ITaaS provider will need to trust the infrastructure provider to properly isolate their VMs from other VMs, which can be from other customers, running on the same physical machine. However, to address the constraint in 6.2.1, we need to make sure that no more than one replica of the same application is assigned to the same physical machine.

Some infrastructure providers, such as AWS and Azure, allow assigning VMs to a *placement group*. A placement group allows users to control the placement of instances (virtual machines) within a data center to meet specific requirements, such as proximity to each other for low-latency communication or spreading instances across different hardware to improve

fault tolerance [8]. Our ITaaS provider can choose the latter placement strategy for their placement group in order to make sure replicas of the same application do not share the same physical machine.

We assume no limit on the number of VMs that can be deployed in this service model. This is because we assume the number of VMs our ITaaS provider needs is much less than the total capacity of the infrastructure provider, since we consider a infrastructure provider like AWS with the capacity to serve a very large number of customers.

To perform proactive recovery, the ITaaS provider will need to issue commands to shutdown and restart a VM via the infrastructure provider's platform. This process can be scheduled to occur automatically at an specified interval in the infrastructure provider's platform. This approach for proactive recovery requires the ITaaS provider to trust the infrastructure provider's platform to properly shutdown and restart the VMs.

Mapping to Cloud Offerings We can map this service model to several infrastructure providers, such as Amazon Web Services (AWS), Azure, and Google Cloud Services (GCP). For AWS in particular, the mapping involves utilizing AWS Elastic Compute Cloud (EC2)'s Instances [7, 6]. These instances represent virtual machines (VMs) from various customers coexisting on shared physical hardware. In this setup, the infrastructure provider handles the assignment of instances to physical hardware, a process not directly controlled by the ITaaS provider. To ensure compliance with the model's requirements, including preventing the coexistence of multiple replicas from the same application on the same physical machine, AWS EC2 provides Placement Groups [8]. ITaaS provider will assign the instances of each application in an AWS region (i.e, a geographical area where Amazon has established data centers to host its cloud computing services) in its own Placement Group with *Spread* placement strategy, thus trusting AWS to place these Instances in different physical machines. To schedule proactive recovery, the ITaaS provider can utilize Amazon Lambda and Amazon Eventbridge services to stop and start Instances at regular intervals (as detailed in [4]).

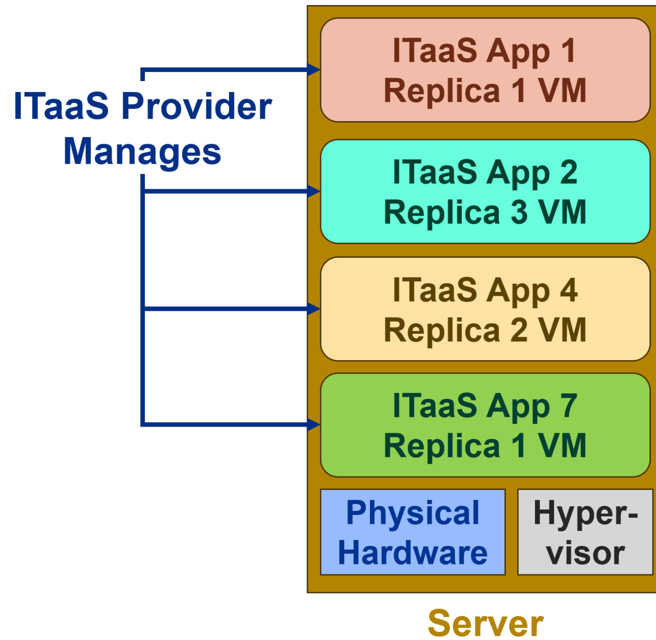


Figure 19: Service Model 2: Dedicated-Server-based ITaaS

6.3.2 Service Model 2 (SM2): Dedicated-Server-based ITaaS

In Service Model 2, the ITaaS provider reserves dedicated servers from an infrastructure provider (Figure 19). Each replica of the intrusion-tolerant application that the ITaaS provider deploys is instantiated in a separate virtual machine (VM). Based on the ITaaS provider's available dedicated servers and their optimization criteria, the ITaaS provider deploys the VM to one of their reserved dedicated servers. In this service model, we assume the ITaaS provider only reserves a limited number of dedicated servers across multiple geographically distributed locations. Since the ITaaS provider is in charge of assignment, they will ensure that replicas of the same application are placed in separate dedicated servers in order to address the constraint in 6.2.1.

In this approach, the ITaaS provider will need to trust the hypervisor running on the dedicated server to operate correctly and to properly manage all the VMs deployed in that dedicated server. To perform proactive recovery, the ITaaS provider will need to issue commands to shutdown and restart a VM via the hypervisor running on the respective dedicated server (similar to the approach in [55]). Alternatively, if the ITaaS provider does not have access to the hypervisor directly (since some infrastructure providers limit access to hyper-

visors), the ITaaS provider will need to issue commands to shutdown and restart a VM via the infrastructure provider's platform. This process can be scheduled to occur at an interval in the infrastructure provider's platform. However, this will require the ITaaS provider to trust the infrastructure provider to properly shutdown and restart the VMs.

Maapping to Cloud Offerings Similar to Service Model 1, we can also map this service model to several infrastructure providers. For AWS in particular, the mapping involves utilizing AWS Elastic Compute Cloud (EC2)'s Dedicated Hosts [5]. Dedicated Hosts represent bare metal servers equipped with a lightweight Nitro hypervisor [62], allowing workloads direct access to CPU and RAM resources. Since AWS limits direct access to the hypervisor, the ITaaS provider will need to trust and use AWS's platform to assign and deploy the VMs (as Instances similar to that of Service Model 1) to the correct Dedicated Hosts, ensuring no two Instances of the same application are placed in the same Dedicated Host. To schedule proactive recovery, the ITaaS can utilize Amazon Lambda and Amazon Eventbridge services to stop and start Instances at regular intervals (as detailed in [4]). Notably, this does not affect other Instances running on the same Dedicated Host.

6.3.3 Service Model 3 (SM3): Colocation-based ITaaS

In Service Model 3, the ITaaS provider reserves colocation rack spaces in data centers from an infrastructure provider (Figure 20). The ITaaS provider is in charge of setting up their own servers on the colocated rack spaces. The ITaaS provider only needs to trust the infrastructure provider for the physical safety of the servers, and also continuous power and cooling for them. We assume the ITaaS provider has full control of their servers, including the hypervisor, and we also assume the ITaaS provider has a limited number of such colocated servers across multiple geographically distributed data centers.

Each replica of the intrusion-tolerant application that the ITaaS provider deploys is instantiated in a separate virtual machine (VM). Based on the ITaaS provider's available colocated servers and their optimization criteria, the ITaaS provider deploys the VM to one of their colocated servers. Since the ITaaS provider is in charge of assignment, they will

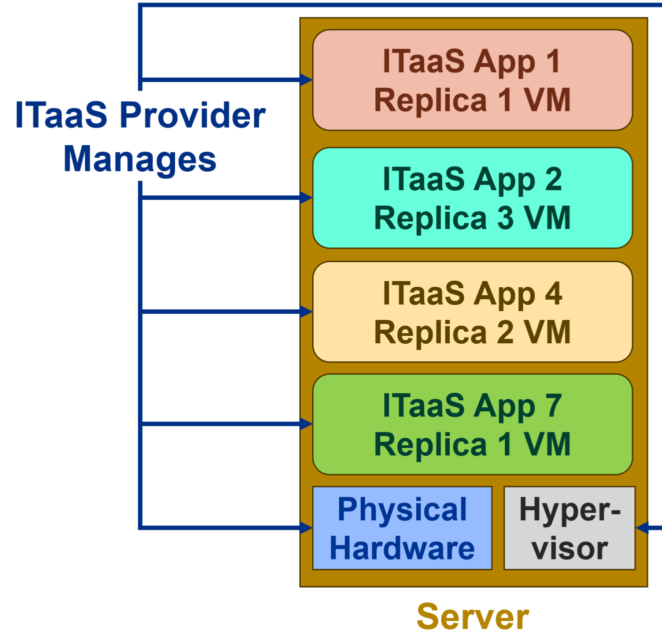


Figure 20: Service Model 3: Colocation-based ITaaS

ensure that replicas of the same application are placed in separate colocated servers in order to address the constraint in 6.2.1.

In Service Model 3, we assume no trust in the underlying hypervisor in each colocated server. Therefore, to perform proactive recovery, the ITaaS provider will utilize netbooter devices (similar to the approach in [55]) to cycle the power of the colocated servers at regular intervals, which will restart all the VMs and the hypervisor in each colocated server. As a consequence, while deploying replicas as VMs in the colocated servers, the ITaaS provider must ensure that VMs with the same application characteristics (i.e., number of tolerated intrusions, proactive recoveries, site disconnections, and proactive recovery interval, since all of these parameters together define the proactive recovery schedule for the application) are assigned in the same physical machines (see details in Section 6.2.2.3).

Mapping to Cloud Offerings There are numerous infrastructure providers that offer colocation services in their data centers, such as Equinix [36], Digital Realty [32], and Coresite [26]. They all provide space, power and cooling as services in their data centers, which are geographically distributed across multiple regions, while the ITaaS provider will need to setup and manage their own servers. As mentioned above, the ITaaS provider oversees

the assignment of VMs to the colocated servers (ensuring replicas of the same application are deployed in separate colocated servers, and replicas in the same colocated server have the same proactive recovery schedule), and will setup a netbooter device for each colocated server, to perform proactive recovery at regular intervals.

6.4 Strategies for Optimizing Replica Placement of Intrusion-Tolerant Applications across Cloud Resources

This section delves into techniques for effectively distributing replicas of Byzantine fault-tolerant applications across a range of cloud resources. Through the use of Mixed-Integer Linear Programming (MILP), these strategies aim to optimize placements while considering crucial constraints like resource availability, latency thresholds, and fault tolerance requirements. Alongside MILP, heuristic optimization techniques offer alternative approaches to placement, additionally taking into account heuristics to increase efficiency, such as choosing sites closer to an application, choosing sites that has least BFT replication protocol latency for an application, and maximizing the number of sites for an application to minimize number of replicas. Together, these algorithms ensure both efficient resource utilization and strict adherence to application specifications across the three service models.

6.4.1 Optimal Solutions using Mixed-Integer Linear Programming

In the following formulations, given a list of available resources and a list of Byzantine fault-tolerant applications, the output is a list of placements. These placements indicate the assignment of replicas to sites for SM1 or servers within the sites for SM2 and SM3. Available resources are characterized by site names, the number of servers (applicable only for SM2 and SM3), and the geographical coordinates of each site. Byzantine fault-tolerant applications are characterized by the number of tolerated Byzantine faults (f), tolerated site disconnections (d), simultaneous proactive recoveries (k), proactive recovery interval (p), maximum latency constraint (l_{max}), and geographical coordinates.

6.4.1.1 Service Model 1 (VM-based ITaaS) MILP formulation

For SM1, we formulate a set of variables and constraints to meet the requirements of the system (eq. 1.1 - 1.20 as detailed below). Next, we define an objective function (eq. 1.21) aimed at maximizing the total number of applications that can be assigned while adhering to these constraints. In this formulation, each application can have multiple *configurations* based on the number of sites being used by that application. Hence we specify an application's configuration number as the number of sites used for that application. For example, an application with configuration number 3 will distribute its replicas across 3 geographically distributed sites. We employ a Mixed-Integer Linear Programming (MILP) formulation and utilize a solver to optimize the allocation of replicas based on the given set of applications and available resources.

Initially, we execute the solver on this MILP formulation with the provided set of applications and resource availability. The output provides the maximum number of applications that could be assigned without violating any constraints. This maximum number is then incorporated as an additional constraint in the MILP formulation (eq. 1.22).

With the updated formulation, we redefine the objective function (eq. 1.23) to minimize the total number of replicas in the system while maintaining compliance with all constraints. Subsequently, we rerun the solver on this modified MILP formulation with the same set of applications and available resources.

Finally, the output of the solver provides the optimal solution in terms of placements, indicating which replica should be assigned to which site. The ITaaS provider will use this information to reserve virtual machines (one for each replica) in the respective sites using the infrastructure provider's platform. In this service model (SM1), the infrastructure provider decides on which physical machine the VM is assigned, so the ITaaS provider only decides on which site the VM is reserved (see details in Section 6.3.1). This solution ensures efficient resource utilization, fault tolerance, and adherence to application requirements within the VM-based environment.

Variables:

$$A : \{1, ..., total_apps\} \quad \text{i.e. set of all applications} \tag{1.1}$$

$$S : \{1, ..., total_sites\} \quad \text{i.e. set of all sites} \tag{1.2}$$

$$\begin{aligned} a_{ic} &: 1 \text{ if application } i \text{ with configuration } c \text{ is active,} \\ &\quad 0 \text{ otherwise, where } i \in A \text{ and } c \in S \end{aligned} \tag{1.3}$$

$$x_{ij} : \text{number of replicas of application } i \text{ in site } j\text{, where } i \in A \text{ and } j \in S \tag{1.4}$$

$$s_{ij} : 1 \text{ if application } i \text{ has replicas in site } j\text{, 0 otherwise, where } i \in A \text{ and } j \in S \tag{1.5}$$

$$\begin{aligned} z_{ijk} &: 1 \text{ if application } i \text{ has replicas in both site } j \text{ and site } k, \\ &\quad 0 \text{ otherwise, where } i \in A \text{ and } j \in S,\ k \in S \end{aligned} \tag{1.6}$$

Constraints:

The following constraint requires that at most one configuration is 1 (active) for each application:

$$\sum_{c \in S} a_{ic} \leq 1 \quad \forall i \in A \tag{1.7}$$

The following constraints set all configurations that are not possible for each application to 0, where $min_sites(i)$ is the minimum number of sites possible for application i (see details in Section 6.4.2.1), $max_sites(i)$ is the maximum number of sites possible for application i (details in Section 6.4.2.1), and $total_sites$ is the total number of sites:

$$a_{ic} = 0 \quad \forall i \in A, \forall c \in \{0, ..., min_sites(i) - 1\} \tag{1.8}$$

$$a_{ic} = 0 \quad \forall i \in A, \forall c \in \{max_sites(i) + 1, ..., total_sites\} \tag{1.9}$$

The following constraints set the required number of sites for each configuration for each application (if a_{ic} is 1, then the total number of sites for application i must be c, otherwise we require the number of sites to be between 0 and $total_sites$):

$$\begin{aligned} &\sum_{j \in S} s_{ij} \leq (a_{ic} * c) + ((1 - a_{ic}) * total_sites) \\ &\forall i \in A, \forall c \in \{min_sites(i), ..., max_sites(i)\} \end{aligned} \tag{1.10}$$

$$\sum_{j \in S} s_{ij} \geq (a_{ic} * c) \quad \forall i \in A, \forall c \in \{min_sites(i), ..., max_sites(i)\} \tag{1.11}$$

The following constraints match s (site variable) with x (placement variable), where $total_servers$ is the total number of servers in a site. s_{ij} is 1 when x_{ij} is at least 1, s_{ij} is 0 otherwise, for application i and site j:

$$x_{ij} \geq s_{ij} \quad \forall i \in A, \forall j \in S \tag{1.12}$$

$$x_{ij} \leq (s_{ij} * total_servers) \quad \forall i \in A, \forall j \in S \tag{1.13}$$

Before we can define the latency constraint, we need to setup the z variable. The following constraints set z_{ijk} to 1 if both s_{ij} and s_{ik} is 1, z_{ijk} is 0 otherwise, for application i, and sites j and k:

$$z_{ijk} \leq s_{ij} \quad \forall i \in A, \forall j \in S, \forall k \in S \tag{1.14}$$

$$z_{ijk} \leq s_{ik} \quad \forall i \in A, \forall j \in S, \forall k \in S \tag{1.15}$$

$$z_{ijk} \geq s_{ij} + s_{ik} - 1 \quad \forall i \in A, \forall j \in S, \forall k \in S \tag{1.16}$$

After properly setting up our s and z variables, we are finally ready to define the latency constraint. Using Practical Byzantine Fault Tolerance [21] as our BFT replication protocol, we limit the total time to order the request, which consists of the following message transitions: request from client to leader site, pre-prepare from leader site to another site, prepare from one site to another site, commit from one site to another site, and finally response from leader site to client. The following is our latency constraint, where $lat_app_site(i, j)$ is the one-way latency between application i and site j, $lat_site_site(j, k)$ is the one-way latency between sites j and k, $app_lat(i)$ is the maximum expected latency for application i, and max_lat is the largest possible maximum expected latency of *any* application (when a_{ic} is not active, we keep the latency constraint higher than any latency possible in the system, hence max_lat is used):

$$\begin{aligned} (s_{ij} * lat_app_site(i, j) * 2) + (z_{ijk} * lat_site_site(j, k)) \\ +(z_{ilm} * lat_site_site(l, m) * 2) \\ \leq (app_lat(i) * a_{ic}) + (max_lat * (1 - a_{ic})) \\ \forall i \in A, \forall j \in S, \forall k \in S, \forall l \in S, \forall m \in S, \\ \forall c \in \{min_sites(i), ..., max_sites(i)\} \end{aligned} \tag{1.17}$$

Each configuration for an application must have at least the minimum number of replicas, where $min_reps(i,c)$ is the minimum number of replicas required for application i with configuration c (details in Section 6.4.2.1). The following constraints set this requirement:

$$\sum_{j\in S} x_{ij} \geq min_reps(i,c) * a_{ic} \quad \forall i \in A, \forall c \in \{min_sites(i), ..., max_sites(i)\} \tag{1.18}$$

$$\sum_{j\in S} x_{ij} \leq (min_reps(i,c) * a_{ic}) + ((1 - a_{ic}) * total_servers * total_sites) \\ \forall i \in A, \forall c \in \{min_sites(i), ..., max_sites(i)\} \tag{1.19}$$

Finally, no site should have more than the required number of replicas, where $max_reps_per_site(i,c)$ is the maximum number of replicas per site for application i with configuration c (details in Section 6.4.2.1). The following constraint sets this requirement:

$$x_{ij} \leq (max_reps_per_site(i,c) * a_{ic}) + (total_servers * (1 - a_{ic})) \\ \forall i \in A, \forall j \in S, \forall c \in \{min_sites(i), ..., max_sites(i)\} \tag{1.20}$$

Objective Function

The primary objective is to maximize the total number of applications:

$$maximize(\sum_{i\in A, c\in S} a_{ic}) \tag{1.21}$$

After the objective above is optimized, we set the optimal number of applications ($num_apps_optimal$) as an additional constraint:

$$\sum_{i\in A, c\in S} a_{ic} = num_apps_optimal \tag{1.22}$$

The secondary objective is to minimize the total number of replicas:

$$minimize(\sum_{i\in A, j\in S} x_{ij}) \tag{1.23}$$

MILP Formulation: For a given set of applications and available resources, we develop the MILP formulation for Virtual-Machine-based ITaaS in Python (using the Python-MIP library [57]) using the variables (eq. 1.1 - 1.6) and constraints (eq. 1.7 - 1.20). We optimize this MILP formulation in two steps: we optimize the primary objective function (eq. 1.21), which maximizes the total number of applications; next, we set this maximum total number of applications as an additional constraint (eq. 1.22), and then optimize the secondary objective function (eq. 1.23), which minimizes the total number of replicas.

6.4.1.2 Service Model 2 (Dedicated-Server-based ITaaS) MILP formulation

The MILP formulation for Service Model 2 is similar to that of Service Model 1. So, we use the same variables, constraints, and objective function (eq. 1.1 - 1.23) here. For Service Model 2, in addition to these, we introduce one more constraint (eq. 2.1 as detailed below) to accommodate the fixed number of servers in each site and impose a cap on the number of replicas each server can host.

We follow the same procedure as Service Model 1 to run the optimization, i.e. optimize for maximizing the number of applications, then add the optimal number of applications as an additional constraint, and then optimize again, but now for minimizing the number of replicas.

Finally, the output of the solver provides the optimal solution in terms of placements, indicating which replica should be assigned to which site. To assign the replicas to specific servers within a site, we use a greedy approach to assign the applications with the largest number of replicas first. Specifically, we rank the applications in that site from largest number of replicas per site to smallest number of replicas per site, and then, following this order of applications, we assign the replicas to the servers in round-robin fashion starting from the first available server.

Finally, we return the assignment of replicas to specific servers as the optimal solution. The ITaaS provider can use this information to instantiate replicas as virtual machines in their specific dedicated servers, which they reserved from the infrastructure provider (as discussed in our discussion of Service Model 2 in Section 6.3.2). This solution ensures efficient resource utilization, fault tolerance, and adherence to application requirements within the Dedicated-Server-based environment.

Additional Constraint: At most total number of servers in each site, and at most 4 replicas per server (in our experiments in Chapter 5, we successfully ran 4 replicas per server and hence we use that number for our formulation here, but this can be changed based on the compute power of the reserved servers):

$$\sum_{i \in A} x_{ij} \leq (total_servers * 4) \quad \forall j \in S \tag{2.1}$$

MILP Formulation: For a given set of applications and available resources, we develop the MILP formulation for Dedicated-Server-based ITaaS in Python (using the Python-MIP library [57]) using the variables (eq. 1.1 - 1.6) and constraints (eq. 1.7 - 1.20, and eq. 2.1). We optimize this MILP formulation in two steps: we optimize the primary objective function (eq. 1.21), which maximizes the total number of applications; next, we set this maximum total number of applications as an additional constraint (eq. 1.22), and then optimize the secondary objective function (eq. 1.23), which minimizes the total number of servers.

6.4.1.3 Service Model 3 (Colocation-based ITaaS) MILP formulation

For SM3, we formulate a set of variables and constraints to meet the requirements of the system (eq. 3.1 - 3.31 as detailed below). The primary difference in the formulation of SM3 as compared to SM1 and S2 is that, unlike in SM1 and SM2, where we assign application replicas to sites, in SM3 we assign applications to *recovery groups*, where a recovery group contains just the applications with the same proactive recovery schedule, and then assign the number of servers in each site to recovery groups. The maximum number of recovery groups possible is same as the number of applications, since every recovery group must contain at least one application.

In this formulation, similar to our SM1 and SM2 formulations, each application can also have multiple *configurations* based on the number of sites being used by that application. Hence we specify an application's configuration number as the number of sites used for that application. For example, an application with configuration number 3 will distribute its replicas across 3 geographically distributed sites.

We define an objective function (eq. 3.32) aimed at maximizing the total number of applications that can be assigned while minimizing the total number of servers used. We employ a Mixed-Integer Linear Programming (MILP) formulation and utilize a solver to optimize the allocation of replicas based on the given set of applications and available resources.

Finally, the output of the solver provides the optimal solution in terms of placements, indicating which replica should be assigned to which site. To assign the replicas to specific servers within a site, we extract this information from the assignments of applications to

proactive recovery groups (which is also provided by the output of the solver). Additionally, the output of the solver also gives us the specific sites and servers that are assigned to each proactive recovery group. Therefore, we can determine the precise placements of replicas on specific servers and return this as the optimal solution. The ITaaS provider can use this information to instantiate replicas as virtual machines in their specific colocated servers (as detailed in our discussion of Service Model 3 in Section 6.3.3). This solution ensures efficient resource utilization, fault tolerance, and adherence to application requirements within the Colocation-based environment.

Variables:

$$A : \{1, ..., total_apps\} \quad \text{i.e. set of all applications} \tag{3.1}$$

$$S : \{1, ..., total_sites\} \quad \text{i.e. set of all sites} \tag{3.2}$$

$$R : \{1, ..., total_apps\} \quad \text{i.e. set of all recovery groups} \tag{3.3}$$

$$\begin{aligned} a_{icr} : 1 \text{ if application } i \text{ with configuration } c \text{ is assigned to recovery group } r, \\ 0 \text{ otherwise, where } i \in A,\ c \in S, \text{ and } r \in R \end{aligned} \tag{3.4}$$

$$\begin{aligned} v_{ir} : 1 \text{ if application } i \text{ is assigned to recovery group } r, \\ 0 \text{ otherwise, where } i \in A, \text{ and } r \in R \end{aligned} \tag{3.5}$$

$$\begin{aligned} x_{rj} : \text{number of servers assigned to recovery group } r \text{ in site } j, \\ \text{where } r \in R \text{ and } j \in S \end{aligned} \tag{3.6}$$

$$\begin{aligned} s_{rj} : 1 \text{ if recovery group } r \text{ has servers in site } j, \\ 0 \text{ otherwise, where } r \in R \text{ and } j \in S \end{aligned} \tag{3.7}$$

$$\begin{aligned} z_{rjk} : 1 \text{ if recovery group } r \text{ has servers in both site } j \text{ and site } k, \\ 0 \text{ otherwise, where } r \in R \text{ and } j \in S,\ k \in S \end{aligned} \tag{3.8}$$

$$q_r : \text{max expected latency for recovery group } r, \text{ where } r \in R \tag{3.9}$$

$$w : \text{ratio of number of servers used to total servers} \tag{3.10}$$

Constraints:

The following constraint requires that at most one configuration and one recovery group is 1 for each application:

$$\sum_{c \in S, r \in R} a_{icr} \leq 1 \quad \forall i \in A \tag{3.11}$$

The following constraint requires that a recovery group is limited to at most 4 applications (in our experiments in Chapter 5, we successfully ran 4 replicas per server and hence we use that number for our formulation here, but this can be changed based on the compute power of the colocated servers):

$$\sum_{i \in A, c \in S} a_{icr} \leq 4 \quad \forall r \in R \tag{3.12}$$

The following constraint requires that applications with different proactive recovery schedule must not be in the same recovery group, where $pr_schedule(i, c)$ is the proactive recovery schedule for application i with configuration c:

$$\begin{aligned} a_{icr} + a_{jdr} \leq 1 \quad \forall r \in R \\ \forall i, j \in A, \forall c, d \in S : pr_schedule(i, c) \neq pr_schedule(j, d) \end{aligned} \tag{3.13}$$

The following constraints set v_{ir} to 1 if an application i for any configuration is assigned to recovery group r:

$$\sum_{c \in S} a_{icr} \geq v_{ir} \quad \forall i \in A, \forall r \in R \tag{3.14}$$

$$\sum_{c \in S} a_{icr} \leq v_{ir} \quad \forall i \in A, \forall r \in R \tag{3.15}$$

The following constraints set all configurations that are not possible for each application to 0 across all recovery groups, where $min_sites(i)$ is the minimum number of sites possible for application i (details in Section 6.4.2.1), $max_sites(i)$ is the maximum number of sites possible for application i (details in Section 6.4.2.1), and $total_sites$ is the total number of sites:

$$a_{icr} = 0 \quad \forall i \in A, \forall r \in R, \forall c \in \{0, ..., min_sites(i) - 1\} \tag{3.16}$$

$$a_{icr} = 0 \quad \forall i \in A, \forall r \in R, \forall c \in \{max_sites(i) + 1, ..., total_sites\} \tag{3.17}$$

The following constraints set the required number of sites for each configuration for each application and recovery group (if a_{icr} is 1, then the total number of sites for application i

in recovery group r must be c, otherwise we require the number of sites to be between 0 and $total_sites$):

$$\sum_{j \in S} s_{rj} \leq (a_{icr} * c) + ((1 - a_{icr}) * total_sites)$$
$$\forall i \in A, \forall r \in R, \forall c \in \{min_sites(i), ..., max_sites(i)\} \tag{3.18}$$

$$\sum_{j \in S} s_{rj} \geq (a_{icr} * c) \quad \forall i \in A, \forall r \in R, \forall c \in \{min_sites(i), ..., max_sites(i)\} \tag{3.19}$$

The following constraints match s (site variable) with x (placement variable), where $total_servers$ is the total number of servers in a site. s_{rj} is 1 when x_{rj} is at least 1, s_{rj} is 0 otherwise, for recovery group r and site j:

$$x_{rj} \geq s_{rj} \quad \forall r \in R, \forall j \in S \tag{3.20}$$

$$x_{rj} \leq (s_{rj} * total_servers) \quad \forall r \in R, \forall j \in S \tag{3.21}$$

Before we can define the latency constraint, we need to setup the z variable. The following constraints set z_{rjk} to 1 if both s_{rj} and s_{rk} is 1, z_{rjk} is 0 otherwise, for recovery group r, and sites j and k:

$$z_{rjk} \leq s_{rj} \quad \forall r \in R, \forall j \in S, \forall k \in S \tag{3.22}$$

$$z_{rjk} \leq s_{rk} \quad \forall r \in R, \forall j \in S, \forall k \in S \tag{3.23}$$

$$z_{rjk} \geq s_{rj} + s_{rk} - 1 \quad \forall r \in R, \forall j \in S, \forall k \in S \tag{3.24}$$

After properly setting up our s and z variables, we are finally ready to define the latency constraints. Using Practical Byzantine Fault Tolerance [21] as our BFT replication protocol, we limit the total time to order the request, which consists of the following message transitions: request from client to leader site, pre-prepare from leader site to another site, prepare from one site to another site, commit from one site to another site, and finally response from leader site to client. The following is our latency constraints, where $lat_app_site(i, j)$ is the one-way latency between application i and site j, $lat_site_site(j, k)$ is the one-way latency between sites j and k, $app_lat(i)$ is the maximum expected latency for application i, and max_lat is the largest possible maximum expected latency of *any* application (when a_{icr} is

not active, we keep the latency constraint higher than any latency possible in the system, hence max_lat is used):

$$q_r \geq (z_{rjk} * lat_site_site(j,k)) + (z_{rlm} * lat_site_site(l,m) * 2) \quad \forall r \in R, \forall j \in S, \forall k \in S, \forall l \in S, \forall m \in S \tag{3.25}$$

$$q_r + (s_{rj} * lat_app_site(i,j) * 2) \leq (app_lat(i) * v_{ir}) + (max_lat * (1 - v_{ir})) \quad \forall i \in A, \forall r \in R, \forall j \in S \tag{3.26}$$

The following constraints require that each configuration for an application must have at least the minimum number of replicas, where $min_reps(i,c)$ is the minimum number of replicas required for application i with configuration c (details in Section 6.4.2.1):

$$\sum_{j \in S} x_{rj} \geq min_reps(i,c) * a_{icr} \quad \forall i \in A, \forall r \in R, \forall c \in \{min_sites(i), ..., max_sites(i)\} \tag{3.27}$$

$$\sum_{j \in S} x_{rj} \leq (min_reps(i,c) * a_{icr}) + ((1 - a_{icr}) * total_servers * total_sites) \quad \forall i \in A, \forall r \in R, \forall c \in \{min_sites(i), ..., max_sites(i)\} \tag{3.28}$$

The following constraint requires that no site should have more than the required number of replicas, where $max_reps_per_site(i,c)$ is the maximum number of replicas per site for application i with configuration c (details in Section 6.4.2.1):

$$x_{rj} \leq (max_reps_per_site(i,c) * a_{icr}) + (total_servers * (1 - a_{icr})) \quad \forall i \in A, \forall r \in R, \forall j \in S, \forall c \in \{min_sites(i), ..., max_sites(i)\} \tag{3.29}$$

The following constraint requires that for each site the number of servers assigned to proactive recovery groups does not exceed the total number of servers in that site:

$$\sum_{r \in R} x_{rj} \leq total_servers \quad \forall j \in S \tag{3.30}$$

Finally, the following constraint sets w to be greater than or equal to the ratio of number of servers used to total servers across all the sites:

$$w \geq \frac{\sum_{r \in R, j \in S} x_{rj}}{total_servers * total_sites} \tag{3.31}$$

Objective Function

The objective function is to maximize the number of applications while minimizing the number of replicas:

$$maximize((\sum_{i \in A, c \in S, r \in R} a_{icr}) - w) \tag{3.32}$$

MILP Formulation: For a given set of applications and available resources, we develop the MILP formulation for Colocation-based ITaaS in Python (using the Python-MIP library [57]) using the variables (eq. 3.1 - 3.10) and constraints (eq. 3.11 - 3.31). We optimize this MILP formulation in a single step: we optimize the objective function (eq. 3.32), which simultaneously maximizes the total number of applications and minimizes the total number of servers.

6.4.2 Heuristic Optimization Algorithms

Although our MILP formulations give optimal solutions, there is an issue with this approach: the runtime of solvers on our MILP formulations does not scale well with larger number of applications or resources. In our experiments (see Section 6.6.3), using an experimental setup with 8 vCPUs and 32GB Memory, the MILP solvers for SM1 and SM2 took about 1 minute for 30 applications and 9 sites (resources), while the MILP solver for SM3 took about 30 minutes for 7 applications and 9 sites (resources). We expect that our MILP formulations for SM1 and SM2 can feasibly scale to either 100 applications or 100 sites, while our MILP formulation for SM3 can feasibly scale to either 30 applications or 30 sites. Beyond these numbers, the solvers can take significantly long time to execute (e.g., months) without access to significantly more computational power.

Hence, we develop heuristic algorithms which run significantly faster than our MILP formulations, and also scale well with larger number of applications or resources. Our heuristic

Algorithm 1 *heuristic_optimizer*: Heuristic algorithm for optimizing placement of Byzantine Fault Tolerance (BFT) applications in cloud resources

---

Require: Applications list A, Resources List R, Method m, Check Latency c

1: Initialize P as an empty list to store placements for all apps
2: **for** each application app in A **do**
3: Initialize p as an empty list to store placements for current app
4: $r = available_resources(R, P)$ {Remove resources already used by placements in current P from R }
5: $Z = pick_sites(app, r, m, c)$ {Select candidate sites for deploying the application (see pseudocode in Algorithm 2)}
6: **if** Z is not empty **then**
7: $n = calculate_num_replicas(app, Z)$ {Determine the number of replicas needed for fault tolerance (see details in Section 6.4.2.1)}
8: **for** each site z in Z **do**
9: $q = calculate_num_replicas_for_site(n, z, Z)$ {Evenly distribute replicas across the Z sites}
10: $p.append(place(app, q, z))$ {Generate placement of replicas for the application on the selected site}
11: **end for**
12: **end if**
13: $P.append(p)$ {Store the placement}
14: **end for**
15: **return** P {Return the list of placements}

Algorithm 2 pick_sites: Algorithm for selecting sites for deploying replicas of a Byzantine Fault Tolerance (BFT) application

---

Require: Application *app*, Available Resources r, Method m, Check Latency c

1: Initialize Z as an empty list to store selected sites
2: $S_{\min} = 2 \times app.d + 1$ {Calculate minimum number of sites required for the application}
3: $S_{\max} = 3 \times app.f \times 2 \times (app.d + app.k) + 1$ {Calculate maximum number of sites possible for the application}
4: $S = \min(S_{\max}, |r|)$ {Choose the size of available resources if needed}
5: **while** $S \geq S_{\min}$ **do**
6:   **if** m is Round-Robin **then**
7:     $Z = round_robin(app, r, S)$ {Select S sites using Round-Robin method}
8:   **else if** m is Closest **then**
9:     $Z = closest(app, r, S)$ {Select S sites which are geographically closest to the application location}
10:   **else if** m is Best-Latency **then**
11:     $Z = best_latency(app, r, S)$ {Select a combination of S sites that provide the optimal latency for executing the BFT protocol for the application}
12:   **end if**
13:   $L = calculate_latency(app, Z)$ {Calculate latency for executing the BFT protocol for the application with the selected sites Z}
14:   **if** $c == False$ **or** $L \leq app.l_{max}$ **then**
15:     {check whether L exceeds the application max latency constraint}
16:     **return** Z {Return the list of selected sites}
17:   **else**
18:     $S = S - 1$ {Decrease the number of sites by 1 and try again}
19:   **end if**
20: **end while**
21: **return** $\emptyset$ {Return empty list if unable to satisfy latency constraint}

algorithms assign each application one at a time, instead of trying to find the best assignments for all applications simultaneously. We believe this is ideal in real world scenarios: we are likely to have several BFT applications already deployed, and redistributing their replicas when a new BFT application joins may not be feasible.

In our heuristic algorithms, given a list of available resources and a list of Byzantine fault-tolerant applications, the output is a list of placements. These placements indicate the assignment of replicas to sites for SM1 or servers within the sites for SM2 and SM3. Available resources are characterized by site names, the number of servers (applicable only for SM2 and SM3), and the geographical coordinates of each site. Byzantine fault-tolerant applications are characterized by the number of tolerated Byzantine faults (f), tolerated site disconnections (d), simultaneous proactive recoveries (k), proactive recovery interval (p), maximum latency constraint (l_{max}), and geographical coordinates.

The pseudocode for our heuristic algorithms is provided in Algorithm 1 and Algorithm 2. Note that for Service Model 3, this pseudocode will additionally need to account for *proactive recovery groups*. Instead of assigning application replicas to sites/servers, we must assign applications to proactive recovery groups, and then assign servers to proactive recovery groups. This is necessary to ensure that applications with the same *proactive recovery schedule* are assigned to the same servers, as discussed in Section 6.2.2.3.

To address the diverse requirements and constraints, we employ multiple heuristic algorithms, each utilizing distinct strategies for selecting sites to assign replicas. For each vairant of the heuristic algorithm, we, initially, decide whether to opt for the minimum or maximum number of cloud sites for each application (Section 6.4.2.1). Subsequently, we employ one of three methods: Round-Robin, Closest, or Best-Latency (Section 6.4.2.2), to select the specific sites for replica distribution. Additionally, the heuristic algorithm may choose to adhere to the Maximum Expected Latency constraint (Section 6.4.2.3). As a result of these choices, we develop eight heuristic algorithms for Service Model 1 and Service Model 2, and four heuristic algorithms for Service Model 3. All the heuristic algorithms are listed in Table 6.

Table 6: Cloud Optimization Algorithms

	Full Name	Short Name
Heuristic	Service Model 1 Minimum Round-Robin Sites	sm1_round_robin_min_sites
	Service Model 1 Minimum Closest Sites	sm1_closest_min_sites
	Service Model 1 Minimum Best-Latency Sites	sm1_best_lat_min_sites
	Service Model 1 Maximum Round-Robin Sites	sm1_round_robin_max_sites
	Service Model 1 Maximum Closest Sites	sm1_closest_max_sites
	Service Model 1 Maximum Best-Latency Sites	sm1_best_lat_max_sites
	Service Model 1 Maximum Closest Sites Meeting Latency Constraints	sm1_closest_max_sites_meet_lat
	Service Model 1 Maximum Best-Latency Sites Meeting Latency Constraints	sm1_best_lat_max_sites_meet_lat
	Service Model 2 Minimum Round-Robin Sites	sm2_round_robin_min_sites
	Service Model 2 Minimum Closest Sites	sm2_closest_min_sites
	Service Model 2 Minimum Best-Latency Sites	sm2_best_lat_min_sites
	Service Model 2 Maximum Round-Robin Sites	sm2_round_robin_max_sites
	Service Model 2 Maximum Closest Sites	sm2_closest_max_sites
	Service Model 2 Maximum Best-Latency Sites	sm2_best_lat_max_sites
	Service Model 2 Maximum Closest Sites Meeting Latency Constraints	sm2_closest_max_sites_meet_lat
	Service Model 2 Maximum Best-Latency Sites Meeting Latency Constraints	sm2_best_lat_max_sites_meet_lat
	Service Model 3 Minimum Closest Sites Meeting Latency Constraints	sm3_closest_min_sites
	Service Model 3 Minimum Best-Latency Sites Meeting Latency Constraints	sm3_best_lat_min_sites
	Service Model 3 Maximum Closest Sites Meeting Latency Constraints	sm3_closest_max_sites
	Service Model 3 Maximum Best-Latency Sites Meeting Latency Constraints	sm3_best_lat_max_sites
MILP	Service Model 1 Mixed-Integer Linear Programming Formulation	sm1_milp_max_sites
	Service Model 2 Mixed-Integer Linear Programming Formulation	sm2_milp_max_sites
	Service Model 3 Mixed-Integer Linear Programming Formulation	sm3_milp_max_sites

6.4.2.1 Calculation of Number of Sites and Replicas

To calculate the minimum number of cloud sites required for each application, we use the formula in Section 5.4, specifically $S_c \geq 2 * d_c + 1$, where d_c is the application's number of tolerated cloud site disconnections, and S_c is the application's number of cloud sites. To maximize the number of cloud sites, the application must have at least one replica in each site. Hence, the maximum number of cloud sites will be the same as the number of cloud replicas. So, we can use $S_c = 3f * 2 * (d_c + k_c) + 1$, where k_c is the application's number of simultaneous proactive recoveries, to calculate the maximum number of sites. If this number is larger than the total number of available cloud sites to the ITaaS provider, then we instead set S_c to the total number of available cloud sites (Note that we still need to ensure $S_c \geq 2 * d_c + 1$ for a valid deployment).

Once we calculate the number of sites, S_c, for an application, we can calculate the required number of cloud replicas for that application using the formula in Section 5.4, specifically $n_c = 3f_c + 2\left\lceil \frac{3f_c d_c + d_c + S_c k_c}{S_c - 2d_c} \right\rceil + 1$, where n_c is the application's required number of cloud replicas, f_c is the application's number of tolerated cloud compromises, and the other parameters are defined same as above. We distribute these replicas as evenly as possible across the S_c sites.

6.4.2.2 Strategies for Selecting Sites

For each application, once we calculate the number of sites, S_c, and the required number of replicas for each site (as detailed in Section 6.4.2.1), we pick the specific sites out of all the available sites based on one of three strategies: Round-Robin sites, Closest sites, or Best-Latency sites.

To implement the Round-Robin sites strategy, we initially handle the case of deploying the replicas for the first application by starting from the first available site in the resource list. For subsequent applications, we begin by assigning the required number of replicas at the available site next to the last-used site for the previous application. We then proceed to

assign the required number of replicas at the next available site, cycling through the list of resources in a round-robin fashion until all replicas for the current application are allocated.

For Closest sites strategy, we simply pick the sites that are closest to the application's location, and assign the required number of replicas in each of those sites.

Finally, to implement the Best-Latency sites strategy, for each possible combination of sites (from available resources) for the given application, we calculate the *maximum expected latency* (see details in Section 6.4.2.3). Subsequently, we select the combination that minimizes the overall maximum expected latency. Once the optimal combination is identified, we proceed to assign the required number of replicas to each of the selected sites.

6.4.2.3 Consideration of Maximum Latency Constraint

Given a set of sites for an application, we calculate the *maximum expected latency* as follows: we calculate the expected latency to order a request, using Practical Byzantine Fault Tolerance [21] as our BFT replication protocol, and the latency between two sites is estimated based on the distance between the two sites. Ordering a request consists of the following message transitions: request from application to leader site, pre-prepare from leader site to another site, prepare from one site to another site, commit from one site to another site, and finally response from leader site to client. Since for a given set of sites, expected latency can differ based on the sites selected for each message transition, we calculate all possible expected latencies for message transitions for the given set of sites, and pick the *maximum expected latency.*

For some of the heuristic algorithms, when choosing the maximum number of sites for an application (based on either Closest or Best-Latency strategy as discussed in Section 6.4.2.2), if the maximum expected latency for ordering a request exceeds the maximum latency constraint of the application, then we decrease the number of sites by one and try again. Once we have the maximum number of sites that does not exceed the maximum latency constraint of the application, we assign the required number of replicas in each of those sites.

6.5 Experimental Setup and Implementation

In order to evaluate the efficacy and scalability of our Intrusion Tolerance as a Service (ITaaS), we designed a series of experiments aimed at comprehensively assessing its performance across different service models (SM1, SM2, and SM3). Figure 21 gives an overview of our experiment setup.

6.5.1 Synthetic Application Data and Available Resources

A crucial aspect of our experiment design involves the generation of synthetic application data. This is summarized in Table 7. We systematically varied the sizes of application sets, ranging from small-scale configurations comprising 3 applications to larger-scale scenarios with up to 100 applications. For each application within these sets, we probabilistically assign values to the following parameters: the number of tolerated intrusions (f), tolerated site disconnections (d), simultaneous proactive recoveries (k), frequency of proactive recovery (r), and latency constraints. We pick the options for these parameters and their probabilities from arbitrary but reasonable choices, where most customers may use minimum tolerance levels for lower cost and/or lower latency, while more critical applications may want more tolerance levels. Also, application locations are distributed across various regions in the USA, considering geographic distribution and urban centers. This approach ensures the creation of diverse and representative application datasets, reflecting real-world scenarios.

In parallel, we defined a fixed set of resources to serve as the infrastructure for our experiments. This resource set comprises 9 sites, each equipped with 30 servers, strategically distributed across different geographic locations including Columbus, Ohio; Eastern Oregon; San Francisco, California; and Virginia. These locations are a subset of Amazon Web Services (AWS)'s data center locations. By establishing a consistent and realistic resource environment, we aim to facilitate meaningful comparisons and evaluations across different algorithmic approaches and service models.

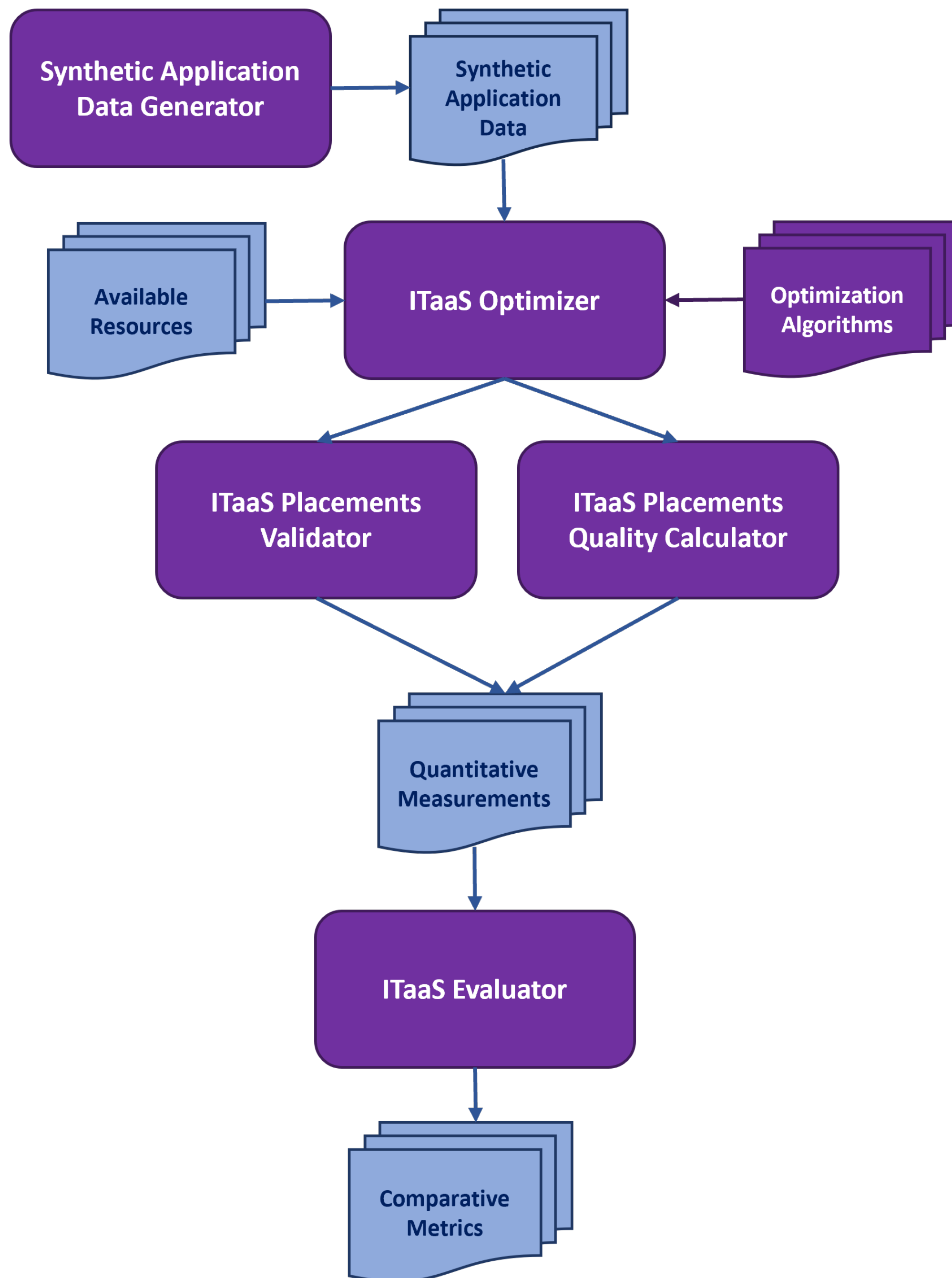


Figure 21: ITaaS Experimental Setup

Table 7: Experimental Setup Details for Generating Synthetic Application Data

Generating Synthetic Applications Data	
Application Parameters	Options for application set sizes: 3, 5, 7, 9, 11, 13, 15, 20, 30, 50, 100
	Number of samples for each application set: 10
Fault Tolerance Characteristics	Options for f: 1 (50% probability), 2 (33.33% probability), 3 (16.67% probability)
	Options for d: 1 (50% probability), 2 (33.33% probability), 3 (16.67% probability)
	Options for k: 1 (50% probability), 2 (33.33% probability), 3 (16.67% probability)
Proactive Recovery	Options for r: 1 (33.33% probability), 3 (26.67% probability), 6 (20% probability), 12 (6.67% probability), 24 (6.67% probability), 48 (6.67% probability)
Latency Constraint	Options for latency constraint: 100, 200, 500, 1000 (all equally likely)
Application Location	20 locations picked across USA, considering geographic distribution and urban centers (all equally likely)

6.5.2 ITaaS Optimizer, Placements Validator, and Quality Calculator

We implemented all the heuristic algorithms and the MILP formulations for each of the service model (6.4) using Python 3 programming language. For the MILP formulations we used the Python-MIP library [57], and used the COIN-OR Linear Programming (CLP) solver that comes by default with the Python-MIP library.

The execution of our experiments is facilitated by our ITaaS Optimizer which is designed to systematically run all selected algorithms for each service model on every application set and the set of available resources. For each execution instance, we pass the placement results to the ITaaS Validator and ITaaS Quality Calculator.

The ITaaS Validator checks if the resultant placements meet all the requirements for the

set of applications, resources and the given service model. For Service Model 1, the Validator checks that: every site exists, the minimum number of sites for each application is satisfied, the total number of replicas for each application is greater than minimum, every site has at least the minimum number of replicas per site for each application, and the latency constraint for each application is satisfied (based on PBFT).

For Service Model 2, the Validator checks that: every site exists, the minimum number of sites for each application is satisfied, the total number of replicas for each application is greater than minimum, every site has at least the minimum number of replicas per site for each application, the latency constraint for each application is satisfied (based on PBFT), each specific machine exists, no machine is overloaded (i.e. more than 4 replicas), and no machine has more than one replica of the same application.

For Service Model 3, the Validator checks that: every site exists, the minimum number of sites for each application is satisfied, the total number of replicas for each application is greater than minimum, every site has at least the minimum number of replicas per site for each application, the latency constraint for each application is satisfied (based on PBFT), each specific machine exists, no machine is overloaded (i.e. more than 4 replicas), no machine has more than one replica of the same application, recovery group contains only applications with same proactive recovery schedule, and no recovery group is overloaded (i.e. more than 4 applications).

The ITaaS Quality Calculator takes the resultant placements and calculates the quality in terms of a set of quantitative measurements, including number of applications assigned, total replicas used, and physical servers utilized. This set of quantitative measurements, the execution time, and validity of placements are captured and stored in a structured format in a CSV file, to enable comprehensive analysis and comparison across different algorithmic approaches and experimental scenarios.

6.5.3 ITaaS Evaluator

Subsequently, we employ an evaluation tool to analyze and compare the performance of heuristic algorithms against the optimal MILP solutions across all service models. Our

evaluation focuses on key metrics such as the number of violations, applications assigned, replicas used, and execution time, providing valuable insights into the efficacy, scalability, and trade-offs associated with each algorithmic approach and service model.

6.6 Evaluation

In this section, we systematically assess several key aspects of our proposed Intrusion Tolerance as a Service (ITaaS) optimization framework, spanning overall feasibility, efficiency for large-scale deployments, processing time of optimization algorithms, and cost analysis of large-scale deployments. These evaluations are pivotal in determining the viability and effectiveness of our optimization framework in real-world scenarios.

Assessing the overall feasibility of our algorithms is paramount as it ensures that the placements generated meet the specified requirements consistently. Efficiency evaluations help us understand how well each optimization algorithm scales with increasing numbers of applications and sites, providing insights into its effectiveness. Moreover, processing time assessments shed light on the computational demands and timeliness of our algorithms, revealing their practicality for real-world use cases. Finally, cost analyses offer a comprehensive understanding of the financial implications of deploying ITaaS across different service models, assisting stakeholders in determining optimal pricing strategies to ensure profitability. Overall, these evaluations collectively contribute to validating our ITaaS optimization framework, ensuring its readiness for real-world environments.

6.6.1 Overall Feasibility

In assessing overall feasibility, our evaluation focuses on the algorithms' consistency in generating valid placements across diverse service models. This is crucial as it ensures that the placements align with specified requirements. Feasibility is evaluated through *verification scores*, wherein placements of application sets onto cloud resources receive a verification score of 1.0 if all requirements for all applications are met (including the option of not placing

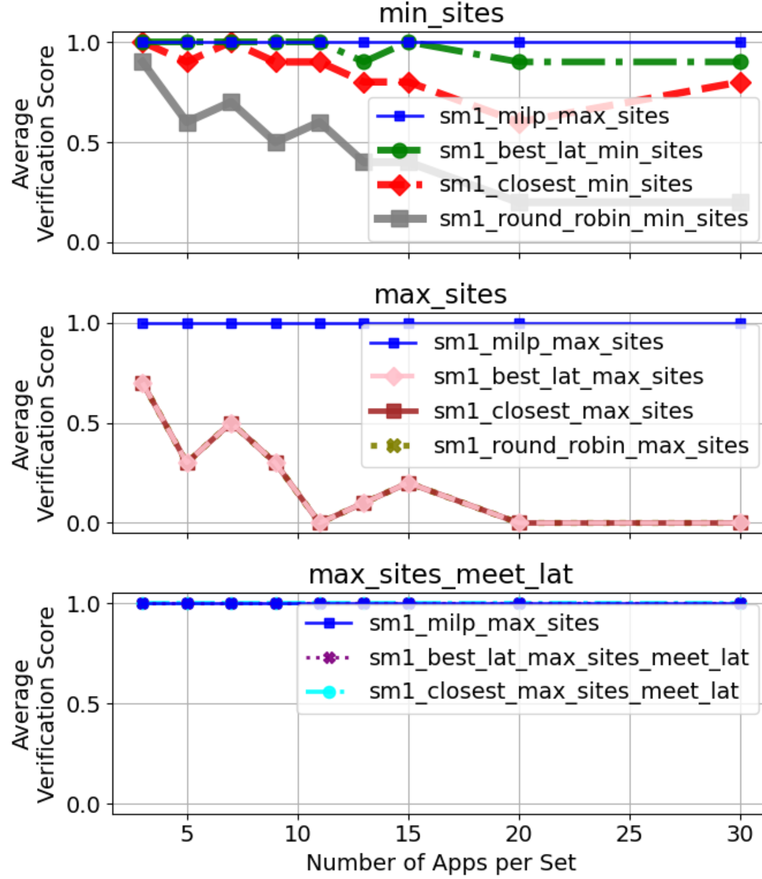


Figure 22: Service Model 1 Average Verification Score

an application). Conversely, if any application's requirements are unmet, the placements for the entire application set is assigned a verification score of 0.0. These metrics are collected by running experiments on the various applications set sizes and recording the outcomes of placements generated by the different algorithms.

Across all three service models (SM1, SM2, and SM3), our evaluation demonstrates the feasibility of our proposed Intrusion Tolerance as a Service (ITaaS) (Figure 22, Figure 23, and Figure 24). For each service model, we have an MILP (Mixed-Integer Linear Programming) formulation, as well as at least one efficient heuristic algorithm that always give valid placements and hence have perfect 1.0 verification score.

Since the MILP formulations are designed to only give placements that meet the constraints (6.2) and requirements for each application, they always result in a verification score of 1.0. Moreover, several heuristic algorithms also achieve perfect verification scores, fur-

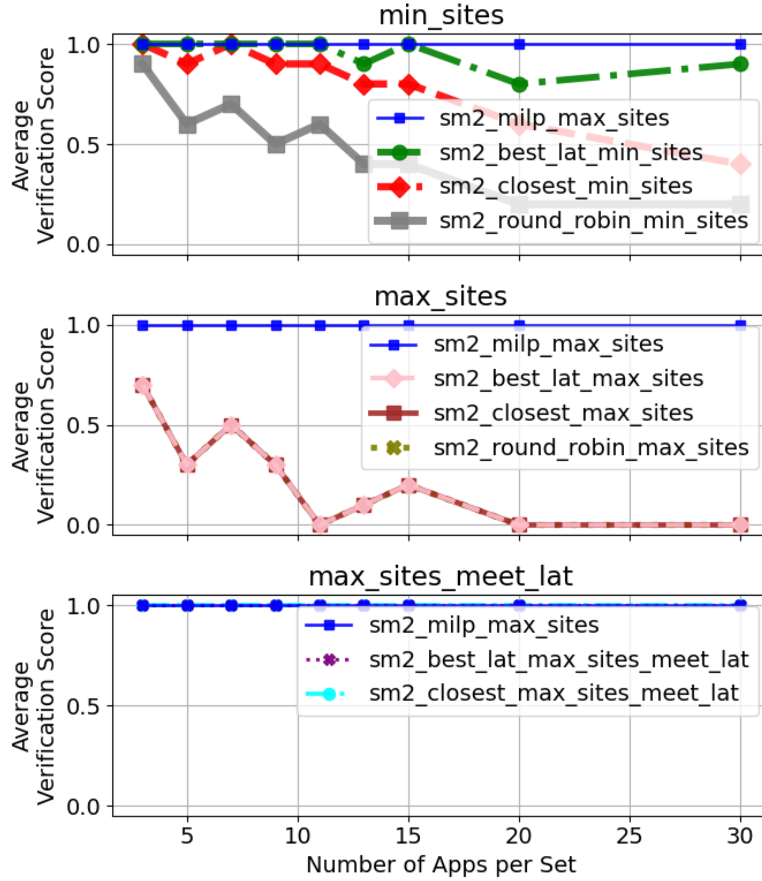


Figure 23: Service Model 2 Average Verification Score

ther validating the feasibility of our service models, since it shows that we can find feasible placements using less expensive algorithms (that can more easily scale to large numbers of applications). Across all our heuristic algorithms that do not achieve perfect verification scores, the primary reason for failing is exceeding max expected latency of the applications.

For both Service Model 1 and 2, as shown in Figure 22 and Figure 23, *max_sites_meet_lat* algorithms achieve similar performance to the optimal solution by prioritizing latency constraints. The *meet_lat* variants are designed to always give valid placements, same as the optimal. In contrast, the *max_sites* algorithms for both Service Model 1 and Service Model 2 exceed the maximum latency most of the time due to the increased likelihood of surpassing latency thresholds with more sites. Similarly, *sm1_round_robin_min_sites* and *sm2_round_robin_min_sites* also perform poorly as they neglect application latency considerations.

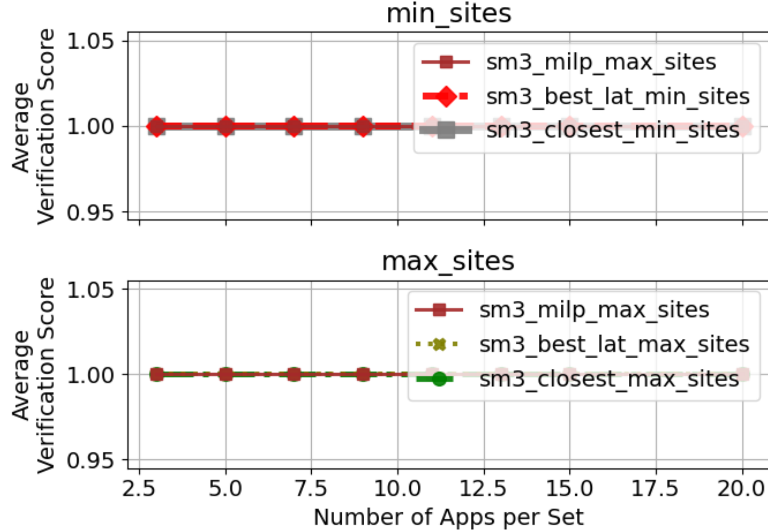


Figure 24: Service Model 3 Average Verification Score

Most of the validation failures occur as we increase the number of applications that the algorithms have to assign in each instance, since this increases the likelihood of exceeding at least one application's max expected latency. Despite not specifically considering application latency constraints, *sm1_best_lat_min_sites* (Figure 22) and *sm2_best_lat_min_sites* (Figure 23) perform well (close to the optimal) by comparing all combinations of the minimum number of sites and selecting the combination that has the least latency. Similarly, *sm1_closest_min_sites* also performs relatively well by selecting the closest minimum sites to the application (this is computationally faster than trying all combinations of sites). Although *sm2_closest_min_sites* follows the same strategy, the additional constraint of fixed set of resources in Service Model 2 leads to a poor verification score as we increase the number of applications.

For Service Model 3, all algorithms successfully meet application latency and proactive recovery requirements, avoiding violations similar to the optimal solution, as shown in Figure 24. This is because all of them are designed to satisfy these requirements.

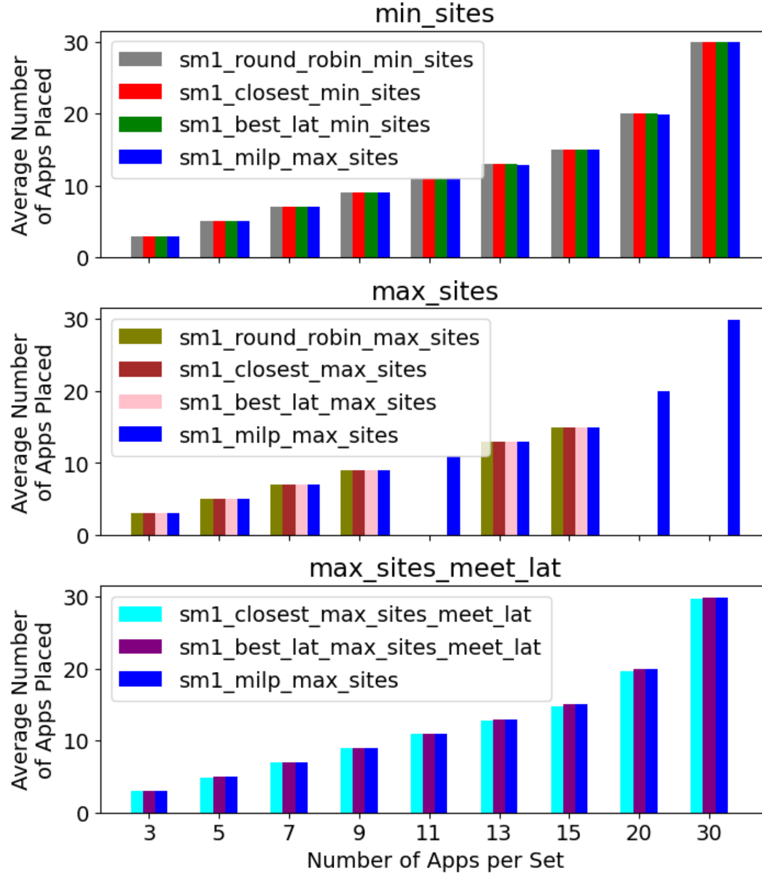


Figure 25: Service Model 1 Average Number of Apps

6.6.2 Efficiency for Large-Scale Deployments

Efficiency for large-scale deployments is another critical aspect evaluated in our experiments. The aim is to maximize the number of applications served while minimizing costs. We collect metrics, specifically the number of applications assigned and the total number of replicas (VMs in Service Model 1) or machines (hosts in Service Model 2, power/colocation space in Service Model 3) required, to assess efficiency. These metrics provide insights into how well the algorithms scale with increasing application set sizes across the different service models.

Note that we collect the number of applications assigned metric only from *valid* placements (i.e., with verification score of 1.0), since it is possible to assign a high number of applications by violating one or more of application requirements. For the number of repli-

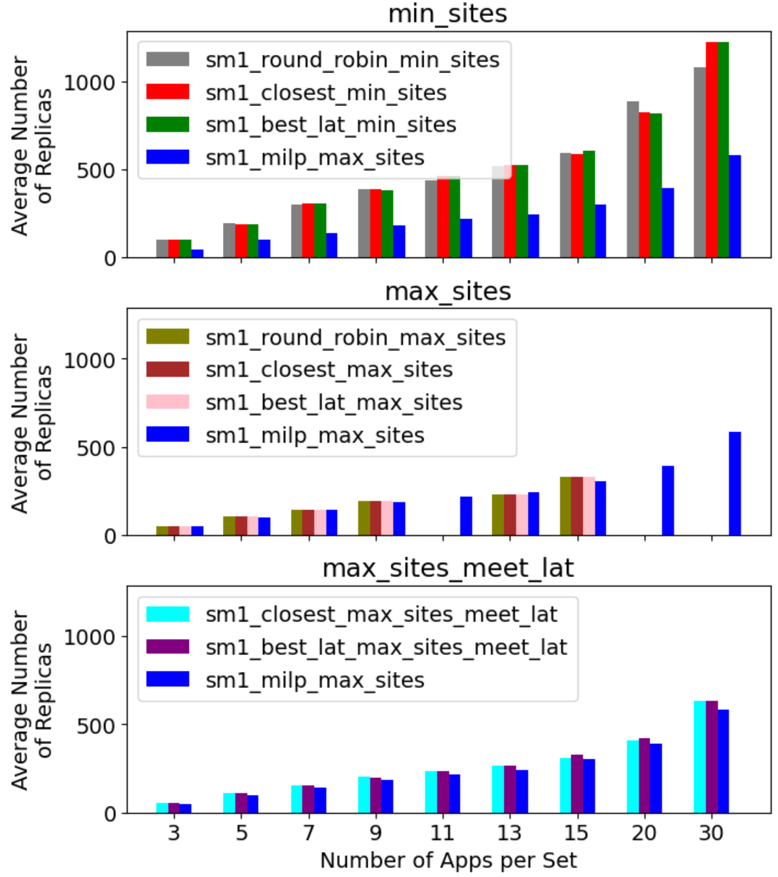


Figure 26: Service Model 1 Average Number of Replicas

cas/machines assigned metric, we only collect from the instances that result in both valid placements and with the *highest* number of applications assigned, since it is also possible to minimize the number of replicas/machines by minimizing the number of applications that are placed.

The MILPs are designed to find the optimal solutions, but heuristics can come close while being feasible to run over larger numbers of applications. In Service Model 1 (SM1), as shown in Figure 25, *min_sites* algorithms demonstrate comparable performance to the optimal solution in terms of the number of applications assigned, attributed to the absence of an upper cap on servers per site. The very slight dip in the *sm1_milp_max_sites* compared to the *min_sites* algorithms is because it avoids assigning applications that exceed the maximum expected latency, unlike the *min_sites* algorithms. Conversely, *max_sites* algorithms struggle to assign many application sets without violations, particularly with larger

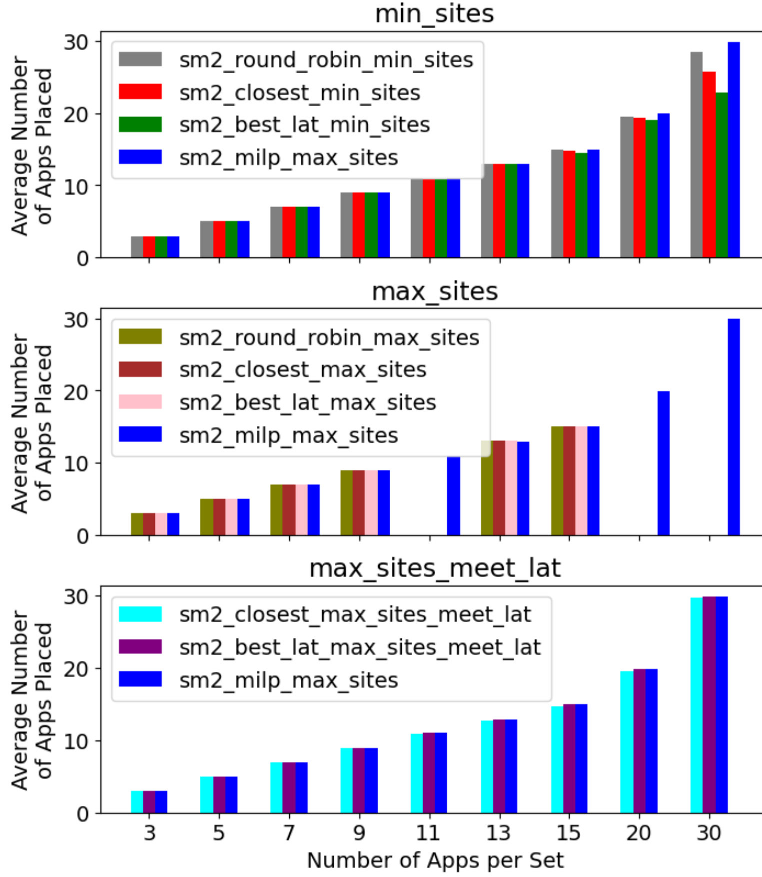


Figure 27: Service Model 2 Average Number of Apps

sets. However, *max_sites_meet_lat* algorithms closely mirror the optimal solution's performance, with minor deviations for *sm1_closest_max_sites_meet_lat* in instances where full application sets exceed latency constraints.

In Service Model 2 (SM2), as shown in Figure 27, trends similar to SM1 are observed, with *max_sites* algorithms closely matching the optimal solution's performance when assigning applications without violations. However, *min_sites* algorithms performs slightly worse, especially with larger sets, as they require a large number of replicas and resources are capped. Interestingly, *sm2_round_robin_min_sites* demonstrates the best performance among the *min_sites* algorithms due to its even distribution across sites. Once again, *max_sites_meet_lat* algorithms exhibit strong performance, with the majority closely aligning with the optimal solution's results.

In Service Model 3 (SM3), as shown in Figure 29, trends similar to SM2 are observed.

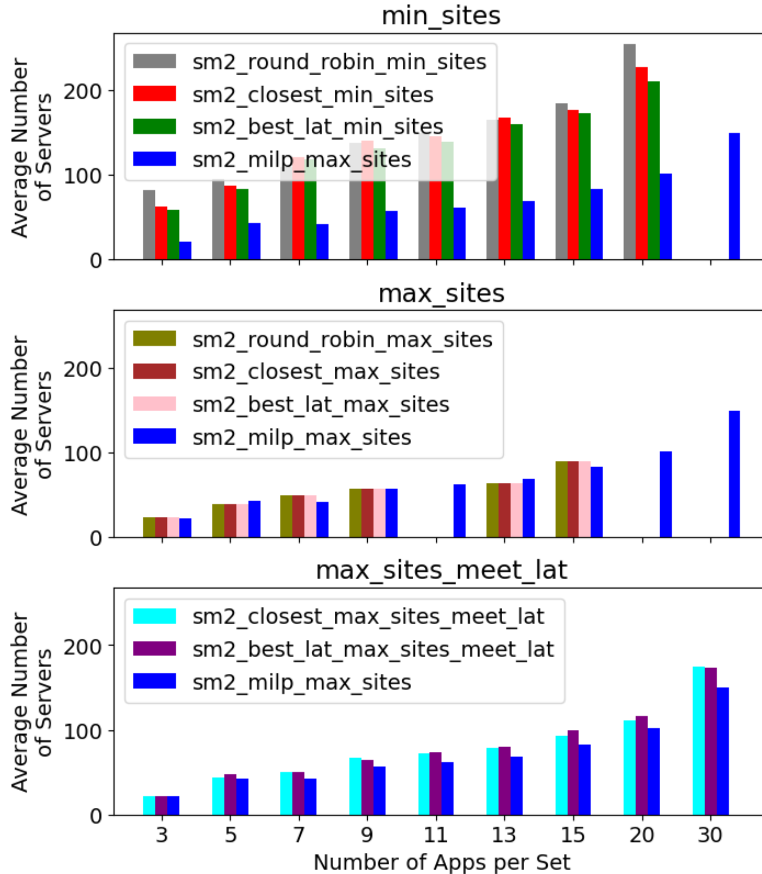


Figure 28: Service Model 2 Average Number of Machines

The performance of the *min_sites* algorithms is notably inferior to the optimal solution. This is primarily due to the fact that, with fewer sites, the algorithms allocate more replicas for applications, depleting available resources rapidly. Additionally, the proactive recovery constraint mandates that applications with identical configurations share the same physical resources, further exacerbating resource scarcity.

In both Service Model 1 and 2, as shown in Figure 26 and Figure 28 respectively, the number of replicas and machines, respectively, required per site inversely correlates with the number of sites, with *min_sites* algorithms necessitating significantly more replicas/machines. Interestingly, *sm2_round_robin_min_sites* requires the most number of machines among the *min_sites* algorithms in Service Model 2 due to its even distribution across sites. Some of the bars in *max_sites* for both Service Model 1 and 2 are zero since they either fail verification or assign less number of applications than optimal. Notably, *max_sites_meet_lat* algorithms

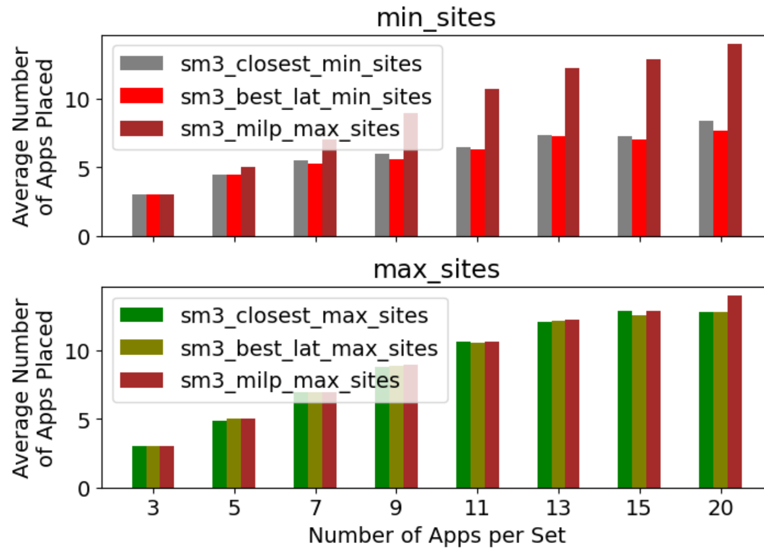


Figure 29: Service Model 3 Average Number of Apps

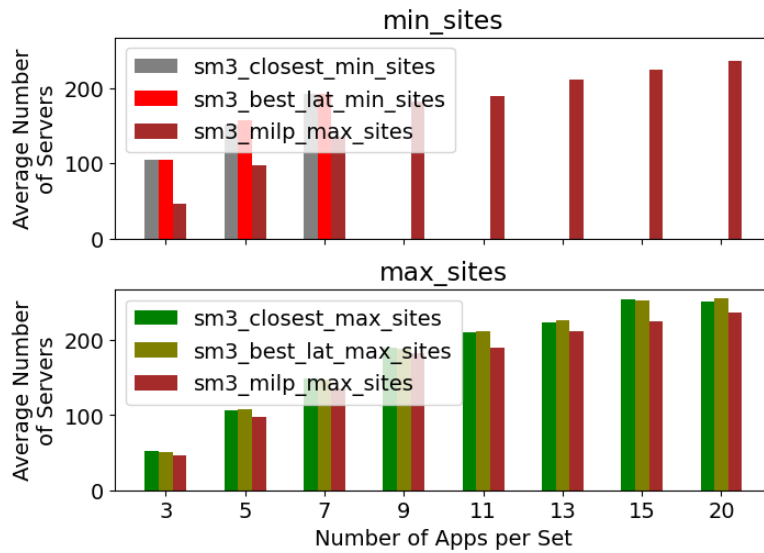


Figure 30: Service Model 3 Average Number of Machines

in both Service Model 1 and 2 exhibit strong performance compared to the optimal solution, While the optimal solutions perform the best, the *best_lat* variants does slightly better than *closest* variants on larger sets since it is able to find larger set of sites for some applications without exceeding the max latency, which consecutively reduces the number of replicas (in case of Service Model 2, the number of machines is reduced since replicas are assigned to machines).

In Service Model 3, as shown in Figure 30, similar trends are observed in regards to number of machines. However, several bars are empty in *min_sites* since they are not able to assign as many applications as the optimal. This is because along with the applications requiring more replicas with less number of sites, the available resources quickly dries up as the proactive recovery constraint only allows applications with the same configuration to share the same physical resources. Notably, the *max_sites* algorithms perform well and close to the optimal, since maximizing number of sites also minimizes the number of replicas, and in turn reduces the number of machines.

6.6.3 Processing Time of Optimization Algorithms

Our evaluation also considers the processing time of optimization algorithms to understand their computational demands. Shorter execution times may be desirable as they indicate faster decision-making and scalability due to lower computational demand. Execution times are collected for all instances of all algorithms by measuring the time taken for each algorithm to generate placements for a given applications set and available resources.

In Service Model 1, 2 and 3, as shown in Figure 31, Figure 32 and Figure 33 respectively, heuristic algorithms typically boast shorter execution times compared to the optimal solution, although slight variations exist due to differing computational demands. Notably, *best_lat* variants for each of the Service Model exhibits the longest execution times among heuristic algorithms. This is primarily attributed to the extensive computation required to calculate latency across various site combinations. The *best_lat_min_sites* variants with fewer sites face a larger number of potential combinations, leading to prolonged processing

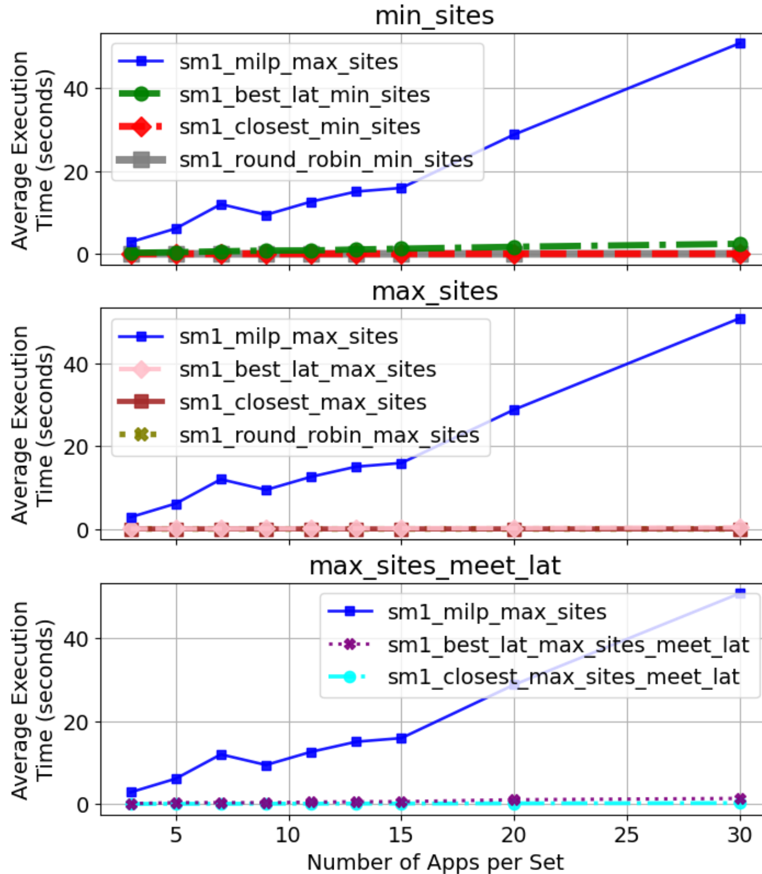


Figure 31: Service Model 1 Average Execution Time

times. Conversely, *closest* heuristic algorithms employing the closest sites approach benefit from simpler computations, resulting in significantly faster performance. This streamlined approach allows these algorithms to swiftly identify the nearest sites to the application, contributing to their efficient execution.

In Service Model 3, the optimal solution flattens out its execution time with more than 7 applications in each set (Figure 33). This is because the execution hits our imposed cap on runtime. Since the MILP formulation for Service Model 3 is very large, it takes very long time to execute in our experimental setup (8 vCPUs and 32 GB Memory), especially with larger numbers of applications. Hence, we impose a cap of 30 min for each run of the MILP solver. Consequently, while the SM3 optimal solution generally requires more processing time than those in SM1 and SM2, the execution times of heuristic algorithms in SM3 appear relatively insignificant in comparison.

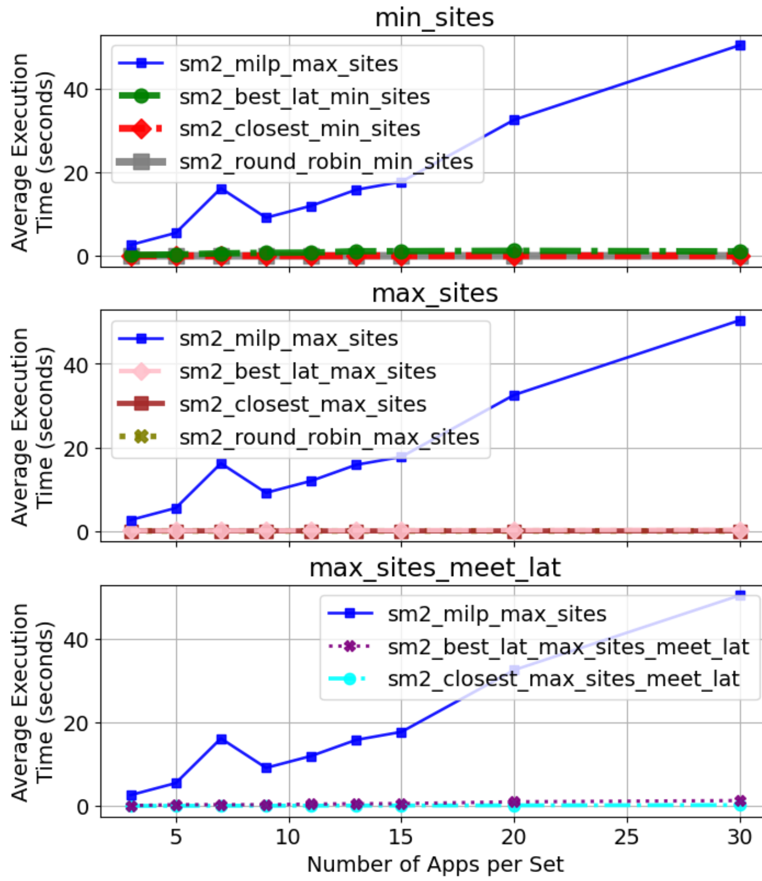


Figure 32: Service Model 2 Average Execution Time

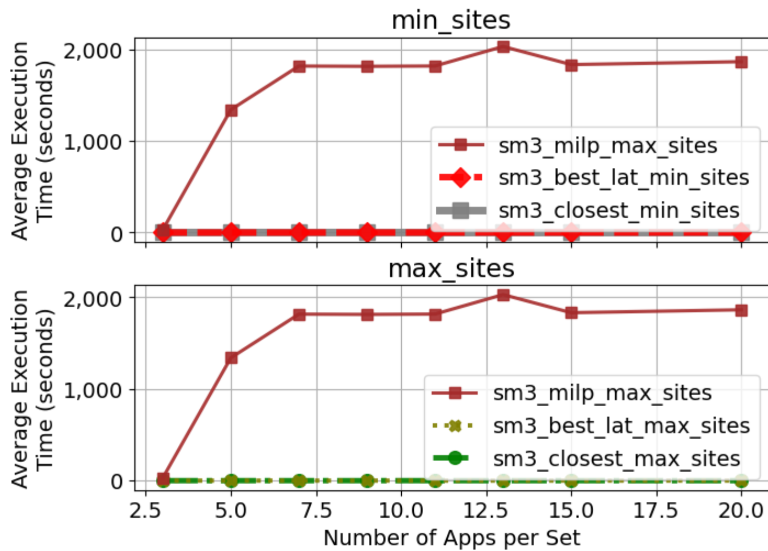


Figure 33: Service Model 3 Average Execution Time

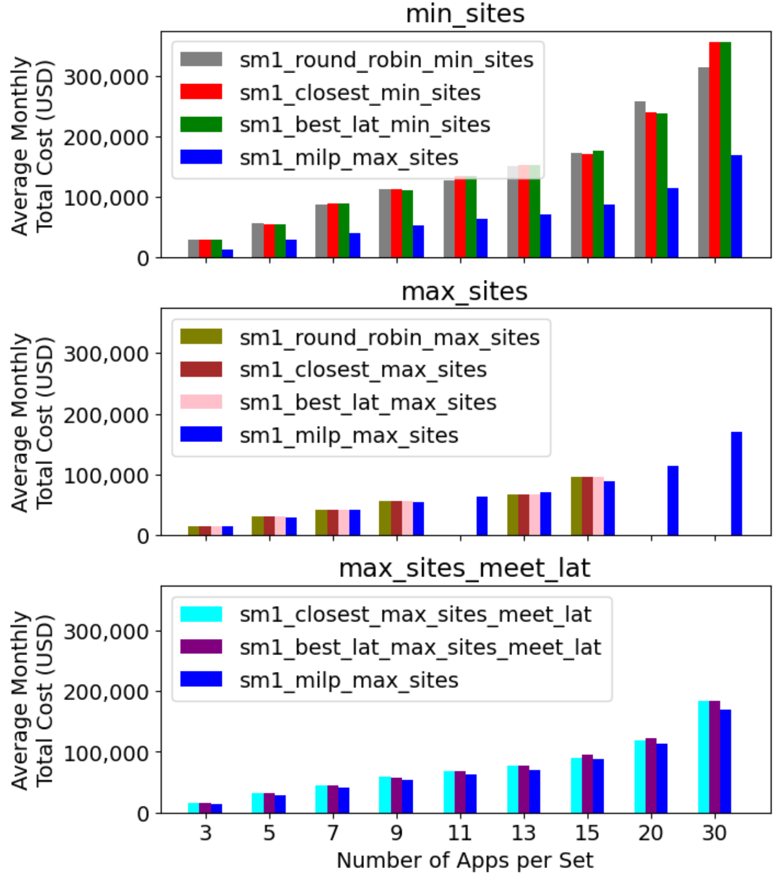


Figure 34: Service Model 1 Average Monthly Total Cost

6.6.4 Cost Analysis of Large-Scale Deployments

In this section, we conduct a comprehensive cost analysis to assess the financial implications of implementing our proposed Intrusion Tolerance as a Service (ITaaS) across three different Service Models (SMs). For each SM, we evaluate the cost implications using various algorithms and deployment scenarios using total cost and cost per application metrics.

6.6.4.1 Service Model 1: AWS EC2 Instance Deployment

For Service Model 1, we utilized the AWS Price Calculator to estimate the on-demand cost for an Instance of typical specifications (c4.2xlarge with 8 vCPUs and 15GiB Memory [61]), which amounted to $290.54 per month. Although AWS offers savings plans, we opted for on-demand pricing to establish an upper bound for our cost analysis.

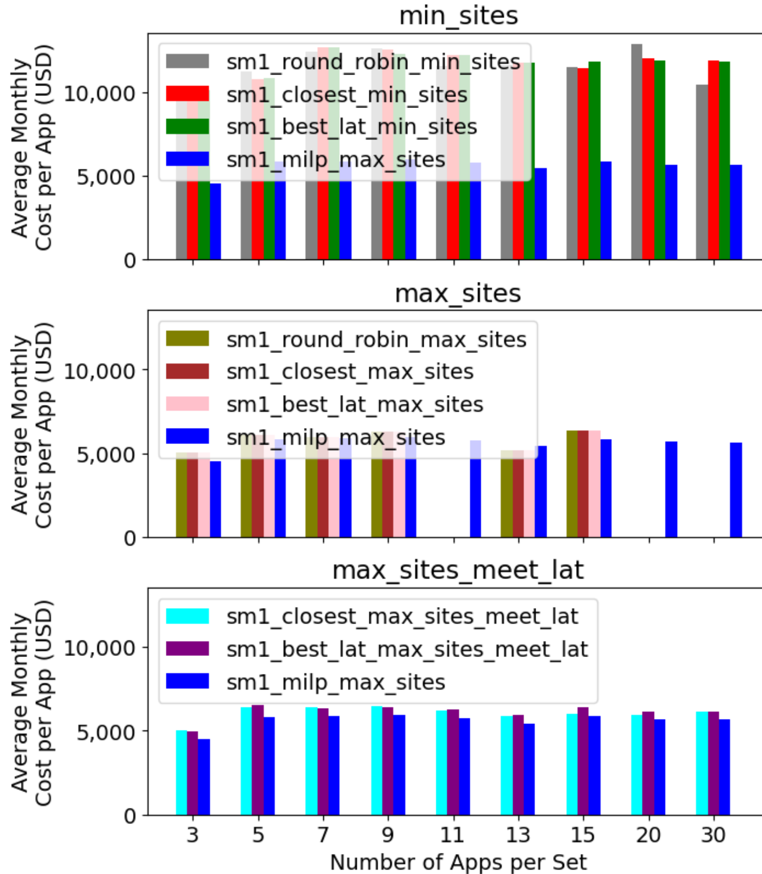


Figure 35: Service Model 1 Average Monthly Cost per App

We employed this pricing data to calculate the average total cost for different application set sizes. Our analysis, as depicted in Figure 34, reveals interesting trends. Overall, the optimal solution performs the best (lowest cost) across all application set sizes. Notably, algorithms such as *min_sites* incur nearly double the cost of the optimal scenario due to their requirement for additional replicas. Conversely, *max_sites_meet_lat* algorithms closely approximate the optimal cost, demonstrating their efficiency. Furthermore, Figure 35 illustrates that *max_sites_meet_lat* algorithms maintain a consistent cost of approximately $6,500 per application, irrespective of the application set size, highlighting their linear cost scalability.

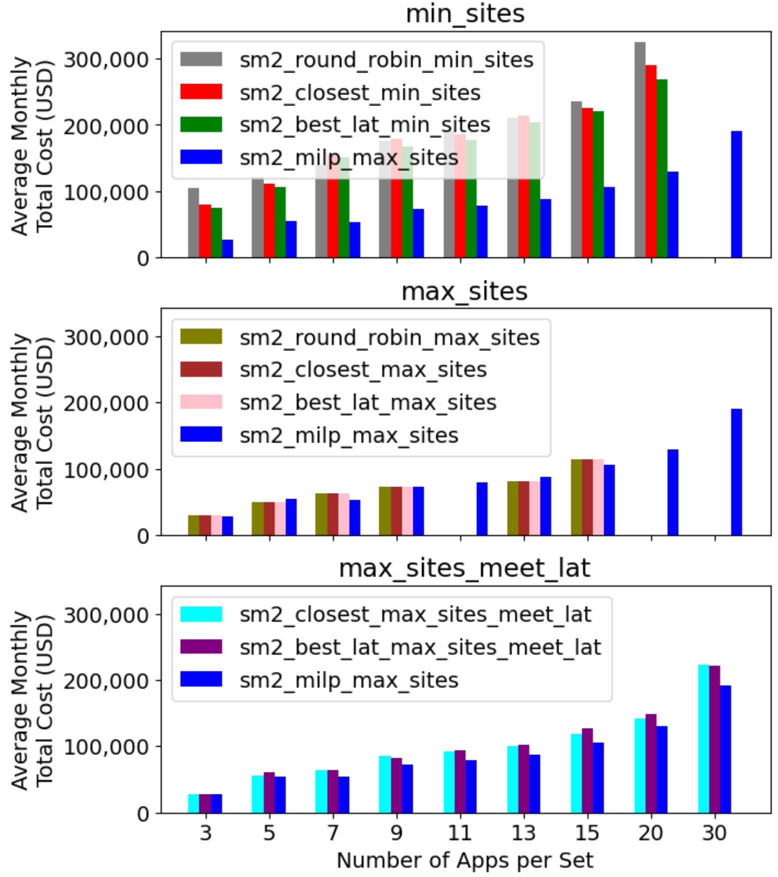


Figure 36: Service Model 2 Average Monthly Total Cost

6.6.4.2 Service Model 2: AWS EC2 Dedicated Host Deployment

In Service Model 2, we assessed the cost implications of deploying on Dedicated Hosts, employing a typical instance (c4.metal with 36 vCPUs and 60 GiB Memory [61]) priced at $1,277.50 per month. Despite potential savings plans, we again utilized on-demand pricing for our analysis.

Our calculations, illustrated in Figure 36, indicate that the optimal solution performs the best (lowest cost) across all application set sizes. The *min_sites* algorithms, particularly *round_robin_min_sites*, incur higher costs compared to other algorithms due to their even distribution of replicas across sites, necessitating more Dedicated Hosts. Conversely, *max_sites_meet_lat* algorithms exhibit cost efficiency, closely approaching the optimal scenario.

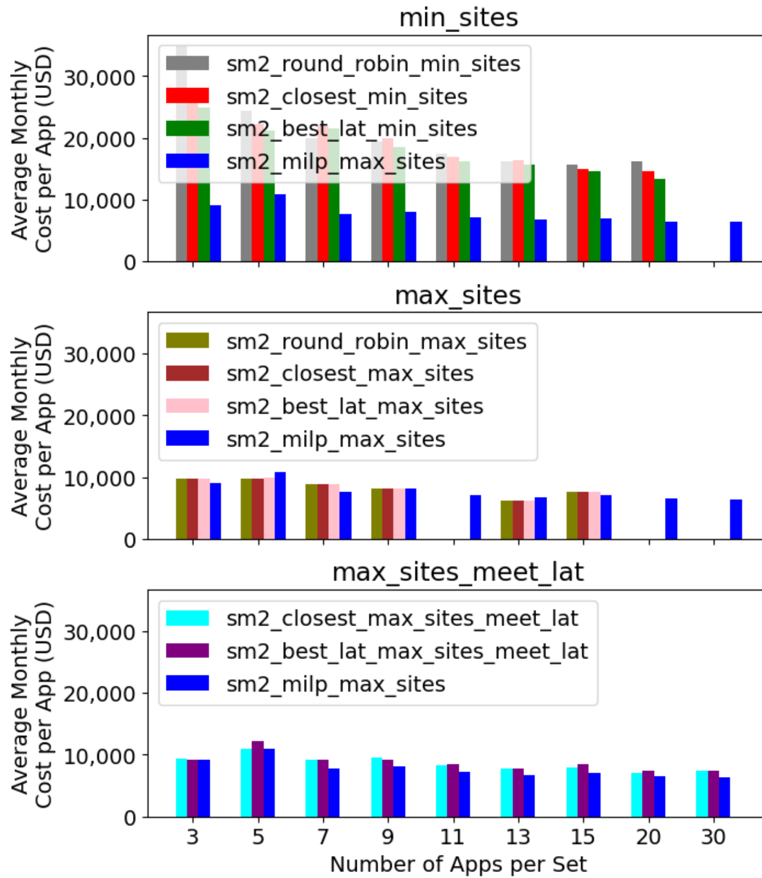


Figure 37: Service Model 2 Average Monthly Cost per App

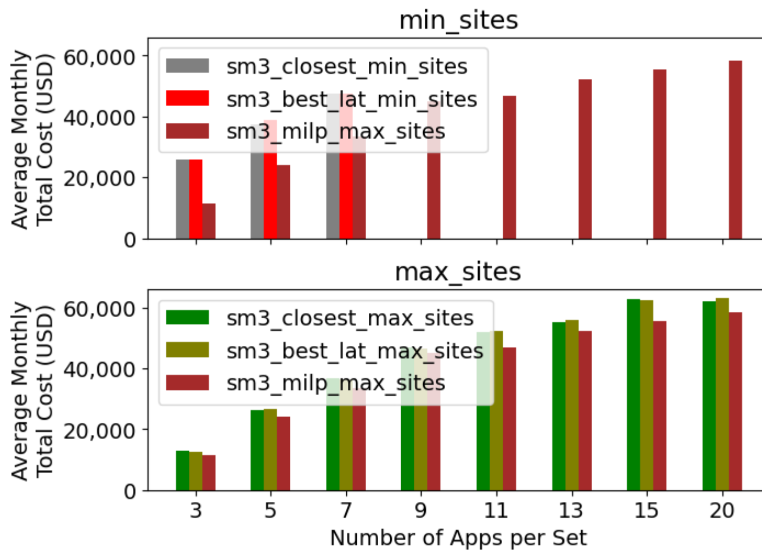


Figure 38: Service Model 3 Average Monthly Total Cost

Figure 39: Service Model 3 Average Monthly Cost per App

Figure 37 further demonstrates that *max_sites_meet_lat* algorithms entail a cost of approximately $10,000 per application for application set sizes of 3, gradually decreasing with larger set sizes, reaching around $7,500 for 30 applications.

6.6.4.3 Service Model 3: Colocation Deployment

For Service Model 3, we assume the service provider will colocate servers in existing data centers by renting space. The cost to colocate is typically measured in terms of power, priced at approximately $163.44 per kW/month, as detailed in [22]. We estimate that a typical server will consume about 1 kW/month, albeit this is an upper bound assumption. Additionally, we consider the initial purchase price of a server to be approximately $5,000, amortized over 5 years.

Thus, the monthly cost of a server, factoring in power consumption and amortization, amounts to approximately $246.77 per month. Although this number is significantly less than that of Service Model 2, it's essential to note that this approach will require technicians/engineers to set up and maintain these servers, potentially increasing the total cost substantially. However, for simplicity, we focus solely on the server cost in this analysis.

Our analysis, depicted in Figure 38, indicates that the optimal solution performs the best (lowest cost) across all application set sizes. The *max_sites* algorithms exhibit cost efficiency, costing approximately $60,000 to support 20 applications. Figure 39 illustrates that these algorithms incur about $4,750 per application for application set sizes of 3, gradually decreasing to around $3,000 for 20 applications.

7.0 Conclusion

7.1 Closing Statement

We designed the first BFT system that leverages offsite data centers to achieve resilience against simultaneous network attacks and system compromises without exposing confidential state or algorithms to data center servers. We extended the system to provide confidentiality guarantees in case an on-premises server is compromised. Our system's performance overhead of providing confidentiality is acceptable, and it can meet the latency requirements of power grid SCADA.

Next, we defined a hybrid management model for intrusion-tolerant systems that allows system operators to leverage intrusion-tolerant ordering and encrypted storage services from a cloud service provider. We designed the first system architecture that enables system operators to deploy intrusion-tolerant applications while offloading the BFT replication protocol to the cloud, preserving confidentiality, and providing resilience to a broad threat model. Our architecture can recover from management domain failures that affect all replicas hosted by the system operator (on-premises). We implemented and evaluated the architecture in an industrial control application, showing that it increases latency by about 9ms (18%) compared to a fully system-operator-managed BFT system but meets application performance requirements.

Furthermore, we make the deployment of our solution easier and cost-effective for an Intrusion-Tolerance as a Service (ITaaS) provider by designing a framework for optimizing the distribution of replicas of different applications across shared cloud resources. We develop heuristic optimization algorithms and Mixed-Integer Linear Programming (MILP) formulations for three separate service models. Evaluating these algorithms reveals their effectiveness in maximizing the number of applications deployed, while minimizing cloud resource usage and overall costs, all while meeting application requirements and constraints unique to ITaaS.

7.2 Lessons Learned

As we conclude this dissertation, we reflect on key lessons learned that will guide future endeavors:

Integrating BFT replication into practical systems is challenging: Despite the theoretical advancements in Byzantine Fault Tolerant (BFT) replication, integrating it into real-world systems presents various challenges. These challenges include complexity in implementation, deployment and management, and compatibility with existing infrastructure and critical applications.

Cloud infrastructure offers a promising solution for managing and deploying intrusion-tolerant systems: Leveraging cloud infrastructure can provide benefits such as scalability, resilience, and cost-effectiveness. Cloud service providers invest in building highly resilient data center infrastructure, which can mitigate many of the challenges associated with deploying and managing intrusion-tolerant systems on-premises.

Decoupled system management can enhance accessibility and scalability: The decoupling of system management between system operators and cloud service providers can improve accessibility and scalability of intrusion-tolerant systems. By separating the responsibilities of application management and intrusion-tolerance management, system operators can leverage the expertise of specialized intrusion-tolerance experts while retaining control over their applications.

Concerns about data sensitivity and specialized hardware hinder full adoption of cloud-hosted solutions: Some applications, particularly those in critical infrastructure sectors like power grids, have stringent requirements for data confidentiality and may rely on specialized hardware or communication protocols that are not easily accommodated in cloud environments. Addressing these concerns is crucial for broader adoption of cloud-based intrusion-tolerant systems.

Optimizing resource distribution in cloud environments is crucial for cost-effectiveness and performance: Efficient resource allocation is essential for maximizing the benefits of cloud-hosted intrusion-tolerant systems. Optimization algorithms can help distribute resources effectively, ensuring that applications meet performance requirements

while minimizing costs.

Critical infrastructure resilience requires a proactive cybersecurity culture with intrusion tolerance: Organizations should adopt a proactive approach to cybersecurity, prioritizing prevention and mitigation strategies alongside intrusion detection and response mechanisms. Incorporating Byzantine Fault Tolerance (BFT) into their cybersecurity framework can significantly enhance resilience against malicious attacks and safeguard critical assets.

7.3 Future Work

This dissertation lays a foundation for further exploration in various directions. Here, we outline a few potential areas for extension:

Enhance the scalability and efficiency of Decoupled Intrusion-Tolerant Systems for large-scale deployments: Further research can address scalability challenges and optimize resource utilization for large-scale deployments. This could involve developing/integrating efficient distributed consensus algorithms, and parallel processing techniques to support large-scale deployments

Investigate further methods to simplify the integration of cloud-based intrusion tolerant systems into diverse application domains: Research could focus on developing abstraction layers, frameworks, or tools that streamline the integration of cloud-based intrusion-tolerant systems into different types of applications. This could involve designing standardized interfaces, providing comprehensive documentation and tutorials, and developing automated deployment scripts to simplify the adoption process.

Provide training and support mechanisms to assist system operators in deploying and managing intrusion-tolerant systems effectively: Developing comprehensive training programs, certification courses, and knowledge-sharing platforms can help system operators acquire the necessary skills and expertise to deploy and manage intrusion-tolerant systems. This could involve collaborations with industry associations, academic institutions, and professional training organizations to develop standardized curricula and

best practices.

Refine optimization algorithms for resource distribution to improve cost-effectiveness and performance in complex environments: Continuing research into optimization algorithms for resource distribution can help enhance the efficiency and cost-effectiveness of cloud-hosted intrusion-tolerant systems. This could involve refining heuristic approaches, and developing machine learning algorithms for predictive resource allocation.

Explore quantum-resistant encryption for enhanced security in intrusion-tolerant systems: Investigate the potential use of quantum-resistant encryption algorithms for securing communication channels and data storage in intrusion-tolerant systems. Research into post-quantum cryptography techniques can help future-proof intrusion-tolerant systems against emerging cryptographic threats.

Bibliography

[1] Ieee standard communication delivery time performance requirements for electric power substation automation. *IEEE Std 1646-2004*, pages 1–36, 2005.

[2] The growing threat of ransomware attacks on hospitals. https://www.aamc.org/news-insights/growing-threat-ransomware-attacks-hospitals, July 2021. Retrieved April 20, 2022.

[3] Imtiaz Ahmad, Mohammad Gh AlFailakawi, Asayel AlMutawa, and Latifa Alsalman. Container scheduling techniques: A survey and assessment. *Journal of King Saud University-Computer and Information Sciences*, 34(7):3934–3947, 2022.

[4] Amazon Web Services. How do i use lambda to stop and start amazon ec2 instances at regular intervals? https://repost.aws/knowledge-center/start-stop-lambda-eventbridge, 2023. Accessed: 2024-03-23.

[5] Amazon Web Services. EC2 User Guide: Dedicated Hosts Overview. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/dedicated-hosts-overview.html, 2024. Accessed: March 23, 2024.

[6] Amazon Web Services. EC2 User Guide: Dedicated Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/dedicated-instance.html, 2024. Accessed: March 23, 2024.

[7] Amazon Web Services. EC2 User Guide: Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Instances.html, 2024. Accessed: March 23, 2024.

[8] Amazon Web Services. EC2 User Guide: Placement Groups. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html, 2024. Accessed: March 23, 2024.

[9] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *IEEE Int. Conf. Dependable Systems and Networks (DSN)*, pages 197–206, June 2008.

[10] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable and Secure Computing*, 8(4):564–577, July 2011.

[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Filipe Araujo, Serhiy Boychenko, Raul Barbosa, and António Casimiro. Replica placement to mitigate attacks on clouds. *Journal of Internet Services and Applications*, 5:1–13, 2014.

[13] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.

[14] Amy Babay, John Schultz, Thomas Tantillo, and Yair Amir. Toward an intrusion-tolerant power grid: Challenges and opportunities. In *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1321–1326, Vienna, Austria, July 2018.

[15] Amy Babay, Thomas Tantillo, Trevor Aron, Marco Platania, and Yair Amir. Network-attack-resilient intrusion-tolerant SCADA for the power grid. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 255–266, Luxembourg City, Luxembourg, June 2018.

[16] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[17] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *Acm transactions on storage (tos)*, 9(4):1–33, 2013.

[18] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. DepSpace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, page 163–176, New York, NY, USA, 2008. Association for Computing Machinery.

[19] Alysson Neves Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Ferreira Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. Scfs: A shared cloud-backed file system. In *USENIX Annual Technical Conference*, pages 169–180. Citeseer, 2014.

[20] BFT-SMaRt. Byzantine fault-tolerant (bft) state machine replication (smart) v1.2. https://github.com/bft-smart/library, 2024. Accessed: 2024-04-15.

[21] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[22] CBRE. North american data center pricing nears record highs, driven by strong demand and limited availability. CBRE Press Release, March 2024.

[23] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *SIGOPS Oper. Syst. Rev.*, 41(6):189–204, October 2007.

[24] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 277–290, New York, NY, USA, 2009. Association for Computing Machinery.

[25] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *USENIX Symp. Networked Syst. Design and Implem. (NSDI)*, pages 153–168, 2009.

[26] Coresite. Coresite Colocation Services. https://www.coresite.com/colocation, 2024. Accessed: March 23, 2024.

[27] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *IEEE Int. Symp. Reliable Distributed Systems (SRDS)*, pages 174–183, Oct 2004.

[28] Miguel Correia, Nuno Neves, and Paulo Verissimo. BFT-TO: Intrusion tolerance with less replicas. *The Computer Journal*, 56(6):693–715, June 2013.

[29] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[30] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

[31] J Deshpande, A Locke, and M Madden. Smart choices for the smart grid. *Alcatel-Lucent Technolgy White Paper*, 2011.

[32] Digital Realty. Digital Realty Colocation Services. https://www.digitalrealty.com/platform-digital/colocation, 2024. Accessed: March 23, 2024.

[33] Dragos Inc. Crashoverride analysis of the threat to electric grid operations. https://dragos.com/blog/crashoverride/CrashOverride-01.pdf. Retrieved April 20, 2022.

[34] S. Duan and H. Zhang. Practical state machine replication with confidentiality. In *IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 187–196, 2016.

[35] Michael Eischer and Tobias Distler. Resilient cloud-based replication with low latency. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 14–28, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Equinix. Equinix Colocation Services. https://www.equinix.com/products/data-center-services/colocation, 2024. Accessed: March 23, 2024.

[37] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Os diversity for intrusion tolerance: Myth or reality? In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN)*, pages 383–394. IEEE, 2011.

[38] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44(6):735–770, 2014.

[39] Essam H Houssein, Ahmed G Gad, Yaser M Wazery, and Ponnuthurai Nagaratnam Suganthan. Task scheduling in cloud computing based on meta-heuristics: review, taxonomy, open challenges, and future trends. *Swarm and Evolutionary Computation*, 62:100841, 2021.

[40] IEEE. Ieee standard communication delivery time performance requirements for electric power substation automation. *IEEE Std 1646-2004*, pages 1–24, 2005.

[41] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 237–250, Hollywood, CA, October 2012. USENIX Association.

[42] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

[43] John C Knight and Nancy G Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on software engineering*, (1):96–109, 1986.

[44] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[45] Bijun Li and Rüdiger Kapitza. Bft-dep: automatic deployment of byzantine fault-tolerant services in paas cloud. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 109–114. Springer, 2016.

[46] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza. Troxy: Transparent access to byzantine fault-tolerant systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 59–70, 2018.

[47] Fang Liu, Kejie Hu, Jing He, Wei Hu, Heyuan Li, Min Peng, and Yanxiang He. A fault-tolerant scheduling algorithm that minimizes the number of replicas in heterogeneous service-oriented cloud computing systems. *The Journal of Supercomputing*, pages 1–17, 2024.

[48] David R Matos, Miguel L Pardal, Georg Carle, and Miguel Correia. Rockfs: Cloud-backed file system resilience to client-side attacks. In *Proceedings of the 19th International Middleware Conference*, pages 107–119, 2018.

[49] Z. Milosevic, M. Biely, and A. Schiper. Bounded delay in byzantine-tolerant state machine replication. In *IEEE Int. Symp. Reliable Distributed Systems (SRDS)*, pages 61–70, Sept 2013.

[50] Rodrigo Nogueira, Filipe Araújo, and Raul Barbosa. Cloudbft: elastic byzantine fault tolerance. In *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pages 180–189. IEEE, 2014.

[51] D. Obenshain, T. Tantillo, A. Babay, J. Schultz, A. Newell, M. E. Hoque, Y. Amir, and C. Nita-Rotaru. Practical intrusion-tolerant networks. In *IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, pages 45–56, June 2016.

[52] Office of Cybersecurity, Energy Security, and Emergency Response. Colonial pipeline cyber incident. https://www.energy.gov/ceser/colonial-pipeline-cyber-incident. Retrieved April 20, 2022.

[53] R. Padilha and F. Pedone. Belisarius: BFT storage with confidentiality. In *IEEE 10th International Symposium on Network Computing and Applications*, pages 9–16, 2011.

[54] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[55] Marco Platania, Daniel Obenshain, Thomas Tantillo, Ricky Sharma, and Yair Amir. Towards a practical survivable intrusion tolerant replication system. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 242–252. IEEE, 2014.

[56] Prime: Byzantine replication under attack. www.dsn.jhu.edu/prime. Retrieved 2020-12-09.

[57] Python-MIP. Introduction — python-mip documentation. https://python-mip.readthedocs.io/en/latest/intro.html, 2024. Accessed: March 24, 2024.

[58] Lili Qiu, Venkata N Padmanabhan, and Geoffrey M Voelker. On the placement of web server replicas. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1587–1596. IEEE, 2001.

[59] Tom Roeder and Fred B. Schneider. Proactive obfuscation. *ACM Trans. Comput. Syst.*, 28(2):4:1–4:54, July 2010.

[60] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. Toward confidential cloud computing. *Communications of the ACM*, 64(6):54–61, 2021.

[61] Amazon Web Services. Compute – amazon ec2 instance types – aws. https://aws.amazon.com/ec2/instance-types/, 2024. Accessed: March 24, 2024.

[62] Amazon Web Services. The security design of the aws nitro system. urlhttps://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html, February 2024. Accessed: March 23, 2024.

[63] Amazon Web Services. What is amazon ec2? - amazon elastic compute cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html, 2024. Accessed: 2024-04-17.

[64] Yuxi Shen, Haopeng Chen, Lingxuan Shen, Cheng Mei, and Xing Pu. Cost-optimized resource provision for cloud applications. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, pages 1060–1067, 2014.

[65] Paulo Sousa, Alysson Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Trans. Parallel Distrib. Syst.*, 21(4):452–465, 2010.

[66] Paulo Sousa, Nuno Ferreira Neves, and Paulo Verissimo. Hidden problems of asynchronous proactive recovery. In *Proceedings of the Workshop on Hot Topics in System Dependability*, 2007.

[67] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

[68] The Spines Messaging System. www.spines.org. Retrieved 2020-12-09.

[69] Spire: Intrusion-tolerant SCADA for the power grid. https://github.com/spire-resilient-systems/spire. Retrieved 2024-04-19.

[70] Thomas Tantillo. *Intrusion-Tolerant SCADA for the Power Grid*. PhD thesis, Johns Hopkins University, 2018.

[71] Robin Vassantlal. Confidential BFT state machine replication. Master's thesis, Universidade de Lisboa, 2019.

[72] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *IEEE Int. Symp. Reliable Distributed Systems (SRDS)*, pages 135–144, Sept 2009.

[73] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, Jan 2013.

[74] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *Proceedings of the sixth conference on Computer systems*, pages 123–138, 2011.

[75] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 253–267, New York, NY, USA, 2003. Association for Computing Machinery.

[76] Laiping Zhao, Yizhi Ren, Yang Xiang, and Kouichi Sakurai. Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 434–441. IEEE, 2010.

[77] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Apss: Proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.*, 8(3):259–286, aug 2005.