AUTOMATED PERFORMANCE ATTACK DISCOVERY IN DISTRIBUTED SYSTEM IMPLEMENTATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Hyojeong Lee

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2014

Purdue University

West Lafayette, Indiana

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

DHT	Distributed Hash Table
ROR	Recursive Overlay Routing
RTT	Round Trip Time
BFT	Byzantine Fault Tolerant system
PBFT	Practical Byzantine Fault Tolerant system
TSC	Time Stamp Counter
VM	Virtual Machine
CSMA	Carrier Sense Multiple Access
BW	Bandwidth
KVM	Kernel Virtual Machine
KSM	Kernel Shared Memory
NIC	Network Interface Controller
PFN	Page Frame Number

ABSTRACT

Lee, Hyojeong Ph.D., Purdue University, December 2014. Automated Performance Attack Discovery in Distributed System Implementations. Major Professors: Charles E. Killian and Cristina Nita-Rotaru.

Security and performance are critical goals for distributed systems. The increased complexity in design, incomplete expertise of developers, and limited functionality of existing testing tools often result in implementations with vulnerabilities and make the debugging process difficult and costly. The deployed vulnerabilities are often exploited by adversaries preventing the system from achieving its design goals. We refer to attacks that slow down the performance of a system as performance attacks. In the past, finding performance attacks has been a painstaking manual process that involved an expert of the target implementation. Given the cost associated with each vulnerability that occurs in the production, there is a need for tools to automatically check that the implementation of a protocol achieves its performance goals with respect to malicious components in the system. In this dissertation, we find performance attacks automatically from implementations of distributed systems. We do not try to show that an implementation is free from all attacks. Our goal is to find attacks and report them to the user in a timely manner. We first investigate how to find attacks automatically from implementations under a simulated environment. A simulated approach, however, has a fundamental limitation in terms of applicable target systems, as certain assumptions are made about languages, operating systems or libraries used. Therefore, we next investigate challenges and requirements to automatically find attacks in implementations of distributed systems under an emulated environment where no limiting assumptions are made.

1 INTRODUCTION

Distributed systems have become the basis of many important real-world services and applications. Many of those applications and services require high availability and high performance at the same time. To meet such requirements under complicated environments, many distributed systems are designed to provide fault-tolerance, security and performance goals.

The increased design complexity, incomplete expertise of developers, and limited functionality of existing testing tools often result in vulnerabilities that prevent implementations of such systems from achieving their design goals in practice, making the systems unusable or generating losses. For example, Amazon is estimated to earn $9,823 every five seconds [1], in other words, five seconds of system downtime can cause a loss of $9,823 for Amazon. Many of these vulnerabilities manifest after the code has already been deployed, making the process of finding vulnerabilities difficult and costly. Moreover, vulnerabilities are often exploited by adversaries and make the system susceptible to attacks. Given the cost associated with each vulnerability that occurs in the wild, there is a need for *tools to check that the implementation of a protocol achieves its performance goals with consideration of malicious components in the system.*

To ease finding vulnerabilities before deployment, automatic debugging techniques such as model checking and symbolic execution have been proposed and their effectiveness have been shown in finding bugs. However, automatic debugging techniques are insufficient because they do not consider the possibility of malicious participants in distributed systems. Automatic debugging techniques only test scenarios where all participants follow the protocol while compromised nodes can behave arbitrarily and conduct attacks. Moreover, there are environments where not every node can necessarily be trusted. Therefore, it is necessary to have a way to automatically test distributed systems with malicious participants.

To test if a system can tolerate attacks, techniques such as abstract model checking have been extended to include attack models. By modeling the system design and attacks, one can check the correctness of the design of a target system against attacks. Checking just the design, however, is also deficient for many reasons. First, implementations include many optimizations and adaptive mechanisms that are not formally specified and proved. Also, many of the actual deployed systems consist of several subprotocols that have unexpected interactions and sometimes opposite goals. Therefore, testing all the interactions of subprotocols is important but challenging. Moreover, implementations do not always fully follow the design for various reasons.

While there are many types of attacks, one class of attacks that are especially difficult to find are attacks that slow down performance of a system, preventing it from achieving its potential performance given the network's condition, referred to as performance attacks. Performance attacks are difficult to find because they are difficult to describe as performance is not a binary property. They are however important, because a system ceases to be useful when its performance drops below a certain point, and performance degradation is costly for both the user and the service provider even when the performance is at a point that the system is still useful.

Finding performance attacks in distributed system implementations primarily has been a manual process. In this thesis, we aim to find performance attacks automatically from implementations of distributed systems. We do not try to show an implementation is free from any attack. Our goal is to find attacks from implementations and report to the user in a timely manner. We first investigate how to find attacks automatically from implementations under a simulated environment, where we have more control in the messages and execution of the target application. Next we investigate challenges and requirements to automatically find attacks in implementations of distributed systems under an emulated environment, where no assumptions are made about the languages, operating systems or libraries used.

1.1 Background and motivation

Most distributed systems are designed to meet application-prescribed metrics that ensure availability and high-performance. However, attacks can significantly degrade performance, limiting the practical utility of these systems in adversarial environments. In particular, compromised participants can manipulate protocol semantics through attacks that target the messages exchanged with honest nodes. Many distributed systems are designed and tested without consideration of such malicious participants. Because of the importance of performance attacks, some systems are specifically designed to tolerate attacks caused by malicious participants behaving arbitrarily. Even so, implementations of those systems are often susceptible to those attacks due to the complexity of implementations.

Traditionally, finding performance attacks in distributed system implementations has been a tedious manual task that requires an expert who possesses vast knowledge about the system and the environment. For example, PBFT [2], one of systems designed and implemented to tolerate the type of attacks we target, is susceptible to a type of attack where a compromised node can degrade the performance of the system significantly by delaying a type of message. However, it took several years to discover this vulnerability.

The problem of automatically finding performance attacks has received very little attention in the past. It is due to the difficulty of expressing performance as an invariant in a system, and the state-space explosion that occurs as attackers are modeled more realistically. The only works we are aware of are the works of Stanojevic et al. [3] and Kothari et al. [4]. Stanojevic et al. [3] find attacks by lying in headers of packets in two-party protocols, and explore all state. This method shows that it is possible to find attacks by exploring state-space; however, it does not scale for distributed systems. MAX [4] requires the user to supply information about the vulnerability of the system, which is a suspected line of code, indicating that it should not be executed many times. This method scales better, however, it focuses only on attacks that exploit control flow. More importantly, it requires specific information about the vulnerability which is often difficult to know.

Therefore, we argue that there is a need for support to find attacks automatically from implementations of distributed systems. In this thesis, we focus on finding performance attacks in distributed system implementations automatically, without requiring the user to provide specific information about vulnerabilities of the system. We focus on performance attacks, since today no distributed system can be be practical without maintaining some level of performance in stable networks.

1.1.1 Finding attacks in distributed systems implementations in a simulated environment

In a typical manual attack finding process, one should first make up attack strategies then implement codes for malicious participants to execute and evaluate such strategies. As we want to automate this process, we need a way to execute malicious participants without manually writing malicious codes. To imitate a reasonable attacker, a malicious participant should behave similar to a benign participant; however, it is not easy to automatically generate malicious codes that deviate from benign behavior in subtle ways.

Building robust, high-performance, large-scale distributed systems is a challenging task. Given the complexity and the importance of distributed systems, many methods have been developed to ease designing, testing, and debugging such systems. Specifically, languages and toolkits have been developed that support a structured message serialization and allow a user to write distributed systems that can run on both a simulated environment and deployment environment without modification [5, 6].

We take advantage of such techniques and assume an environment where events are simulated, and message serialization is structured. A simulated environment helps us to manipulate the timings and executions of events, while structured message serialization allows us to manipulate the message contents without modifying target implementations. Therefore, it is easier to generate malicious behaviors without manually implementing malicious software. We investigate how to automatically find attacks in distributed system implementations under a simulated environment. Then, we design and implement Gatling to show it is possible to find attacks automatically.

1.1.2 Finding attacks in distributed systems implementations in an emulated environment

A simulated approach requires the target application to be written in the language that is supported by the simulator. However, many distributed systems are already implemented in various languages, and most languages do not allow a user to run their implementations directly on a simulated environment. To use a simulation-based solution, the user would need to rewrite their software, which is often unfeasible or very expensive.

Furthermore, many distributed system implementations often depend on some specific environment, such as an OS or a library. These dependencies also often play important roles in performance or vulnerability. Therefore, it is important to test implementations under an environment that is as close to the deployment environment as possible. However, in a simulation- based approach, the environment is replaced with the simulator and it is impossible to include the environment in a simulation-based approach.

The requirement of a specific language, and lack of ability to use an environment that is similar to the deployment environment limits the practicality of simulated approaches. Therefore, we need a way to automatically find performance attacks in distributed system implementations, under realistic environments, without posing restrictions on languages or methods the application is implemented in. Also, to maximize its practicality, we want to avoid modification of the implementation or the operating system.

To test applications that are written in various languages under an environment that is similar to the deployment environment, we choose an emulated environment instead of a simulated environment. In an emulated environment, we do not have control over timing of events nor message serialization. Therefore, we need a new way to imitate an attacker. We investigate the challenges in finding attacks for a less constrained, emulated environment. Then, we design and implement a platform, Turret, which addresses these challenges.

1.2 Thesis contributions

This thesis focuses on finding attacks automatically in distributed system implementations without requiring the user to provide information about vulnerabilities in the implementation.

We summarize our key contributions:

- We investigate how to find performance attacks automatically from distributed system implementations and design Gatling, a platform to find attacks in an environment where timing events are simulated, and message serialization is centralized.

- We provide a concrete implementation of our design, Gatling, on top of the Mace toolkit. We present an attack discovery algorithm that uses a greedy search approach based on an impact score that measures attack effectiveness.

- We apply Gatling to nine different distributed system implementations and found a total of 48 attacks many of which were not previously discovered.

- We investigate how to find performance attacks in unmodified implementations of distributed system implementations and design Turret, a platform for performance attack discovery in unmodified distributed system implementations running in a realistic environment. Turret leverages virtualization to run arbitrary operating systems and applications, and uses network emulation to connect these virtualized hosts in a realistic network. We also present new attack finding algorithms that have low time overhead, which is critical for platforms that run systems in realistic conditions where the systems run in real time and not simulated time.

- We present an implementation of Turret that uses the KVM virtualization infrastructure and the NS3 network emulator. Our attack finding algorithms require execution branching to compare the attack impact for attack and non-attack executions that are branched from an identical checkpoint. To support execution branching of the entire distributed system, we design and implement checkpointing and restoring the state of its execution. In addition, we create a page sharing aware snapshot management functionality to optimize the snapshot performance.

- We show the effectiveness of Turret by applying it to implementations of five different intrusion tolerant systems. We found a total of 30 attacks, and many of them were not previously documented. We also describe how we used Turret in a graduate distributed systems class.

1.3 Thesis roadmap

The rest of this thesis is organized as follows. A description of our system and attacker model is in Chapter 2. We present our simulation based approach to find attacks automatically in distributed system implementations in Chapter 3. Next we present Turret, an emulation based approach that overcomes limitations of the simulation-based approach can not avoid in Chapter 4. We describe related work in Chapter 5, then conclude in Chapter 6.

2 SYSTEM AND ATTACK MODEL

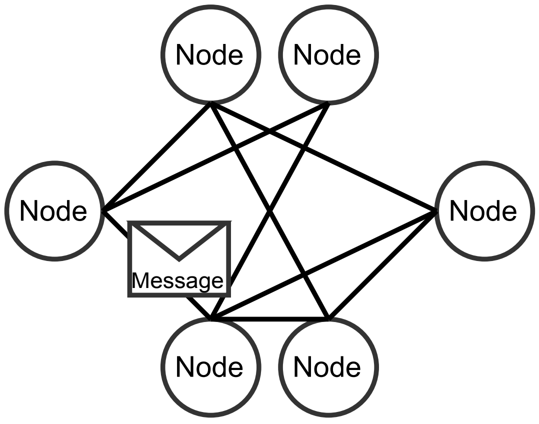
In this chapter, we describe the system model that we consider, specifically, how nodes communicate to invoke behavior of the other, and then describe our attack model. We focus on performance attacks in distributed systems caused by compromised participants. We first define performance attacks and describe actions that malicious participants can cause.

2.1 System model

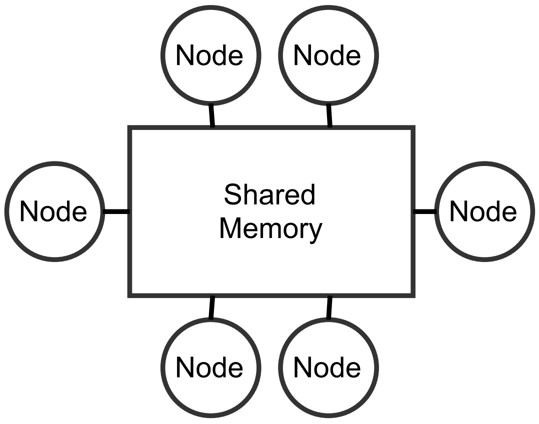
Many distributed systems [5, 7–15] are designed following event-based state machines that communicate through messages and messages trigger events in processes involved. Several other systems use RPCs [6, 16], continuations [17], or data flow [18], which are *compatible* with this approach, where nodes communicate via messages, as opposed to sharing memory as shown in Fig. 2.1. Thus, we focus on distributed systems implementations that are conceptually message passing event-driven state machines, and we will refer to this as the *message-event model*.

Messages can be delivered in one or multiple packets, depending on the system and/or the size of the message. One can define a network event based on delivery of a packet or delivery of a message. As we focus on message-event model, we consider a network event as an event to deliver a message rather than a packet, if a message is contained in several packets.

We also focus on systems that have measurable performance metrics, such as throughput, latency, etc. Many distributed systems already make their performance metrics clearly measurable.



(a) Message passing



(b) Shared memory

Figure 2.1.: Shared memory model vs. message passing model

2.2 Attack model

In this section, we describe the type of attacks we consider, then describe malicious actions that malicious participants are capable of.

2.2.1 Performance attack

We focus on attacks against system performance where compromised participants running malicious implementations try to degrade the overall performance of the system through their actions. Evaluating such attacks require that the system has a performance metric that gives an indication of the progress the system made in completing its goals. Most distributed systems already have some measurable performance. How much of performance degradation is meaningful depends on the metric, the system, and the application. For example, a latency of 200 ms can be tolerable in a peer to peer file download system, but video conference applications require a latency less than 150ms. While it is not possible to generalize how much of performance degradation can be regarded as an attack, it is often clear for the user how much of performance degradation is acceptable or not. Therefore, we ask the user to provide some Δ, which means the relative amount of performance degradation from the benign performance that he or she would consider problematic. The goal of performance attacks we consider is to degrade performance of the system performance by the Δ that the user has provided.

Specifically, we define:

Performance attack A set of actions that deviate from the protocol, taken by a group of malicious nodes and resulting in performance that is worse by some Δ than in benign scenarios.

2.2.2 Malicious actions

We consider insider attacks that have a global impact on system performance and which are conducted through message manipulation. We want to automate insider attackers without asking the user to implement malicious software. Below we describe capabilities of the attacker, that are the primitives to build up an attack.

A malicious node can globally influence the performance of the system by misleading other honest participants through exchanged messages. We classify all malicious actions on messages into two categories, message delivery actions and message lying actions. Specific malicious actions we consider are the following:

Message delivery actions. Performing message delivery actions does not require knowledge of the messaging protocol, because the actions are being applied to where and when the message is delivered, rather than modifying the message contents. We define the following types of malicious message delivery actions.

- *Dropping*: A malicious node drops a message instead of sending it to its intended destination.
- *Delaying*: A malicious node does not immediately send a message and injects a delay.
- *Diverting*: A malicious node does not send a message to its destination as intended by the protocol, and instead enqueues the message for delivery to a node other than the original destination.
- *Duplicating*: A malicious node sends a message twice instead of sending only one copy and applies delay to the second copy.

Message lying actions. We define message lying actions as actions where malicious participants modify the content of the message they are sending to another participant. An effective lying action involves intelligently modifying fields of messages to contain contents likely to cause different behaviors, which is more sophisticated than random bit-flipping. Consider, for example, a simple attack which is to modify a boolean value in a message. While random bit-flipping can easily change the right bit, knowing which bit is the right bit to flip requires either costly trial-and-error or sophisticated static analysis of the code. These changes are dependent on the messaging protocol. As the number of possible values that the message field could contain may be extremely large, we define a few general strategies for field types that stress the system in different ways based on previously found attack practices. We lie about values based on the type of the message field, the original value of the benign message, or both.

3 GATLING: FINDING ATTACKS IN A SIMULATED ENVIRONMENT

Finding performance attacks in implementations of distributed systems manually is a tedious and difficult task. Recent success of automatic debugging techniques helps to reduce bugs in distributed system implementations, however, these techniques do not consider malicious scenarios. In this chapter, we investigate how to find performance attacks in distributed system implementations automatically with assists of languages and tools that are designed to ease development of distributed systems that allow us to test the implementation on a simulated environment.

3.1 Introduction

Current best practices in building robust, high-performance, large-scale distributed systems involve strict coding standards, careful code review, extensive test plans, and a combination of automated and manual testing aids. Large scale testing is primarily done manually using simulation platforms such as NS3 [19], GTNets [20], and P2PSim [21], or emulation environments such as Emulab [22], ModelNet [23], and DETER [24]. There are also wide-area testbeds such as PlanetLab [25], RON [26], and GENI [27]. Each of these has their advantages and disadvantages, but generally only allow user-driven testing at a larger scale. Programming platforms such as Wids, P2, Mace, Tame, and Splay have also been developed to ease the development of code and for the re-use of previously implemented components [5, 6, 14, 17, 18]. These platforms also often include tight integration with simulation or testing tools. Available testing tools are primarily focused on finding bugs in systems implementations. These include execution logging and replay tools such as the Friday [28], X-ray [29] or Wids Checker tools [30], test case generation tools such as Klee [31], and a variety of simulation-based systematic testing tools.

Given the difficulty of finding bugs manually, several techniques have been proposed and applied to find them automatically. Model checking using static analysis [32, 33] has been used to verify that the design of a system is correct. Given the model of the system, this approach proves that some invariants hold for every reachable state in the system. Model checking has limited applicability for complex designs due to the intractable number of states that must be checked. Additionally, checking the design is not a guarantee that the actual code is free from bugs because models do not capture all the intricacies of real implementations, and additional bugs can be introduced during implementation.

Finding bugs in distributed system implementations has been done with the use of symbolic execution, fault injection, and model checking. Symbolic execution [31] has been used to generate test cases that are capable of covering many control flow branches. This technique suffers from a state-space explosion when applied to more complex implementations. Fault injection [34] has been used to discover unknown bugs by testing fault handling code that would not normally be tested. Fault injection is often limited in scope because it is difficult to apply to implementations in a systematic manner. Finally, model checking using a systematic state-space exploration [32, 35–39] has been used on system implementations to find bugs that violate specified invariants. To mitigate the effect of state-space explosion, this approach uses an iterative search, bounding some aspect of the execution. Heuristics based on systematic state-space exploration techniques do not prove bug absence, but rather help pinpoint where bugs do exist.

More recently, debugging techniques have been applied to find attacks automatically . Many works have been focused on finding or preventing vulnerabilities that either cause the victim to crash or allow the attacker to gain escalated privileges [40–44]. Dynamic taint analysis [42–44] has been used to protect implementations from well-defined attacks, such as buffer overflow attacks. Taint analysis is limited in that it is a *detection* mechanism, not a search mechanism. Fault injection with an iterative parameter space search [45] has also been used to find vulnerabilities in distributed systems. However, this approach requires a costly parameter optimization limiting the size of the system it can be used to analyze.

Most distributed systems are designed to meet application-prescribed metrics that ensure availability and high-performance. However, attacks can significantly degrade performance, limiting the practical utility of these systems in adversarial environments. In particular, compromised participants can manipulate protocol semantics through attacks that target the messages exchanged with honest nodes. To date, finding attacks against performance has been primarily a manual task due to both the difficulty of expressing performance as an invariant in the system and the state-space explosion that occurs as attackers are more realistically modeled. The only works we are aware of that focused on automatically finding performance attacks are the woks in [3] which considers lying in headers of packets in two-party protocols, and the work in [4] which assumes the user supplies a suspect line of code, indicating that it should not be executed many times. The method in [3] explores all states and does not scale for a distributed system. The technique used in [4] provides better scalability by combining static with runtime testing, but focuses only on attacks that exploit control flow and where attackers know the state of the benign nodes.

In this chapter, we focus on how to automatically detect performance attacks on *implementations of large-scale message passing distributed systems*. We consider insider attacks that have a global impact on system performance and which are conducted through message manipulation. We focus on these attacks given they have received limited attention, they can cause significant disruption on the system, and they are applicable to many distributed systems. Our goal is to provide a list of effective attacks to the user in a timely manner while requiring the user to provide only one or multiple metrics measuring system progress. In this work, we focus on distributed system implementations that are written in languages that are designed to help implement and test distributed systems. Our contributions are:

- We propose Gatling, a framework that combines a model checker and simulator environment with a fault injector to find performance attacks in event-based message passing distributed systems. We identify basic malicious message delivery and message lying actions that insider attackers can use to create attacks. We design an attack discovery algorithm that uses a greedy search approach based on an impact score that measures attack effectiveness. Gatling works for a large class of distributed systems and does not require the user to write a malicious implementation. While Gatling does not fix attacks nor prove their absence, it provides the user with protocol-level semantic meaning about the discovered attacks.

- We provide a concrete implementation of Gatling for the Mace toolkit [5]. Mace provides a compiler and runtime environment for building high performance, modular distributed systems. Our changes include: (1) adding an interposition layer between Mace services and the networking services to implement malicious message delivery actions, (2) modifying the Mace compiler to include a message serialization code injector to implement message lying actions, and (3) modifying the simulator to implement our attack discovery algorithm. The user provides an implementation of a distributed system in Mace and specifies an impact score in a simulation driver that allows the system to run in the simulator.

- We demonstrate with a case study how to use Gatling to find attacks on a real system implementation, the BulletPrime peer-to-peer file distribution system. Our goal is not to criticize BulletPrime's design, but to explore its behavior in an adversarial environment. While some of the attacks we found on BulletPrime were expected, such as delaying or dropping data messages, others were more surprising. Specifically, BulletPrime's reliance on services that provide improved download times led to a number of vulnerabilities.

- We further validate Gatling by applying it to eight additional systems having different application goals and designs. First five systems include the Vivaldi [46] virtual coordinate system, the Chord lookup service and distributed hash table (DHT) [16], two multicast systems, ESM [47] and Scribe [48]. Because these systems are designed without considering defending against insider attackers, we additionally applied Gatling to three different systems that are particularly designed to combat types of attacks we target: two secure virtual coordinate systems, Vivaldi Outlier Detection [49], and Newton [50] and a byzantine fault tolerant system: PBFT [2]. Gatling found a total of 48 performance attacks across the systems tested, 20 attacks based on message lying actions and 28 attacks based on message delivery actions. Finding each attack took a few minutes to a few hours.

The rest of the chapter is organized as follows. Section 3.2 describes the system model of Gatling. Sections 3.3 and 3.4 describe the design and implementation of Gatling. Section 3.5 provides an example of how to use Gatling to find attacks in a well-known distributed file sharing system, BulletPrime [51]. Section 3.6 presents results on using our tool on eight representative distributed systems: Vivaldi [46], Chord, DHT [16], ESM [47], Scribe [48], Vivaldi Outlier Detection [49], Newton [50] and PBFT [2]. Section 3.7 provides a summary of this chapter.

3.2 Gatling system and attack model

In this section, we describe the system and the attack model of Gatling. Our model is derived from the system model of MaceMC [37], which is a model checker for implementations written in the Mace language [5]. We first provide a brief summary of the MaceMC system model and describe how our model is different from theirs.

3.2.1 System model

In the system model of MaceMC, a program execution is a sequence of system states. A system state can be considered as the combination of the program state at each of the nodes representing the value of the program variables, and the network state representing the state of connected sockets and pending messages. Connecting each pair of states in the program execution is a transition, the new state is the result of executing an event handler atomically, based on the incoming event and the previous state of the system. Events include application events, message delivery, and timer expiration. An event handler is a callback function that consists of a set of instructions to execute atomically. MaceMC explores non-determinism of event ordering and checks properties at the end of an event handler execution. MaceMC model checker checks both safety and liveness properties.

Safety properties need to be satisfied at every state while liveness properties need to be satisfied eventually. The Mace language [5] allows developers to implement their distributed systems under this model and therefore systems implemented in the Mace language follow this model automatically.

The basis of the model of Gatling is the same in that we have a sequence of system states that transit by receiving events. We introduce a new source of non-determinism in the model to include malicious transitions. Given the opportunity, a malicious node can non-deterministically behave, and Gatling explores this new type of non-determinism. Possible actions that malicious nodes can make are defined in Section 3.3.2.

Including malicious transitions in the model increases the state-space even more; state-space is often too big to search completely even before including malicious transitions. To mitigate this problem and make the search practical, Gatling only explores non-determinism introduced by malicious transitions. Gatling does not explore benign non-determinism, which can be explored by the MaceMC model checker. Instead, Gatling uses a time-based simulator that behaves predictably by network topology and random seeds.

Limiting its search space by using a simulator for benign non-determinism reduces the state space explored, therefore, the soundness of the search. Nevertheless, the space is still too huge to explore comprehensively. Also, as we discuss in Section 3.3.3, we are searching for attack strategies that are broadly effective, not specific attack paths, which would be hard to interpret; thus, Gatling uses heuristics that further make the search practical.

3.2.2 Attack model

We focus on distributed systems that communicate through messages and on malicious actions conducted by compromised peers that harm the non-malicious nodes. Therefore, among three types of events in the MaceMC model (application event, timer expiration and network event), malicious nodes only manipulate network events. Such actions target the message delivery (e.g. dropping, delaying, duplicating messages) and message contents (e.g. modifying specific fields) as defined in Chapter 2. Such actions are created by insider

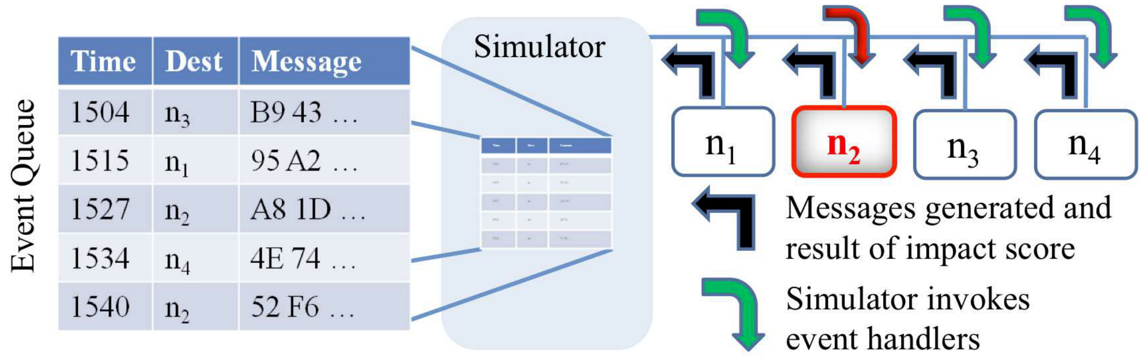


Figure 3.1.: Gatling simulator model

attacks thus a message verifier would not reject the incoming message based on a simple filter of allowable messages or authentication mechanism. Our work prioritizes those malicious actions that have the greatest detriment to the running system, so attacks that affect only a single healthy node will be considered less critical than those which affect the overall progress of the distributed system. Specific actions are described in detail in Section 3.3.

3.3 Gatling design

In this section, we provide the overview of the design, identify malicious actions that represent building blocks for an attack, and describe our algorithm that searches and discovers attacks consisting of multiple malicious actions.

3.3.1 Design overview

Model checking event-driven state machines. A well-known approach to find bugs in systems implementations is to use systematic state-space exploration based model checking, initially pioneered by Godefroid in 1997 [52], which allows a user to explore the set of all possible behaviors. This approach, when applied to systems implementations, results in a systematic state-space exploration through carefully controlled execution to determine all reachable system states. The state of a distributed system is conceptually the combination of the state maintained at each node, in conjunction with the state of the network connecting the distributed nodes in the system.

The message-event model provides opportunities for reducing the state-space. First, it avoids the complexity of simulating networking and routing layers by abstracting the network to be a basic service which either provides FIFO reliable message passing (such as TCP), or best-effort message passing (such as UDP). As a result, the network state is given by the set of messages in-flight, and the corresponding service guarantees for each message. Second, it limits the complexity model of concurrency by maintaining the event queue, and systematically executing each possible event sequence. In particular, with event-based message passing systems, a simulation of an execution entails keeping a queue of all pending events; then, for each timestep, picking an event to execute, and running the event handler for that event on the selected node. Events may be network events (e.g. delivery of a message), scheduled events (e.g. expiration of a timer), or application events (e.g. user request for an action). Since nodes only interact through messages, event handlers can be run in isolation without concern for physically concurrent event handlers running at other nodes. Concurrency within the process is not a concern because nodes run a single event handler at a time.

Several prior model checker designs [36, 37, 52, 53] have explored the capabilities of event-driven state machines to find bugs in systems implementations. Each of these designs provides mechanisms to explore complex interactions between nodes which would normally be infeasible for a user to exhaustively explore. They do not require abstract models of the system as they explore the state-space using the implementation itself. The advantage of this approach is that it eliminates the possible gap between the model and the implementation, and the disadvantage is that it increases the state-space and cause the state-space to explode exponentially. Due to the exponential state-space explosion as the search depth increases, these systems settle for heuristically exploring the state-space and locating correctness bugs that are the result of execution from benign participants (i.e. do not focus on attacks).

Our approach. The main idea of systematic state-space exploration based model checking is to search all possible states of the system and check correctness properties (safety invariants). With this idea, one could find attacks using a model checker by extending the state-machine to include malicious transitions, i.e., run the model checker on the target system implementations with the implementations of malicious participants. To find performance attacks, one should check the performance of the system instead of correctness properties.

It is infeasible, however, for multiple reasons. First, systematic exploration based model checkers already suffer from the state-space explosion problem and adding malicious transitions aggravates the problem. In fact, the newly added state-space is much bigger than the benign state-space as malicious transitions do not need to follow the protocol but can act arbitrarily. Therefore, we need to find ways to reduce the state-space aggressively while preserving the attacks we seek to find. Second, as we do not have implementations of the malicious participants, it is impossible to check malicious implementations. Therefore, we need to generate malicious implementations automatically. Moreover, finding performance problems is very challenging in an exhaustive approach because often these bugs are the result of specific timings, for which finding would require searching on the space of possible timings of events, far less practical approach. On the other hand, simulating performance of a system is straightforward - as the simulator keeps an ordered list of outstanding events, including the time at which they are scheduled to occur. Each time an event executes, the clock of that node advances, allowing the system to conduct a time-based event driven simulation. However, it does not systematically explore all the possible executions. Nevertheless, repeated simulation can be used to uncover performance variances, as identified by Killian et al. in their work of performance checking [54].

Our design, Gatling, overcomes these limitations by using a hybrid approach. Specifically, Gatling uses a time-based simulation model to provide support for detecting performance attacks, and integrates a search algorithm into the time-based simulation model to practically find such attacks. The resulting architecture is illustrated in Fig. 3.1. Gatling constructs a set of nodes, with a fraction of them flagged as being malicious. Gatling maintains an event queue sorted by event start time and simulates the event queue normally. However, when an event is executing on a node selected to be malicious, Gatling uses a model-checking exploration approach to test the set of different possibilities for what new events are scheduled by the malicious node; in particular, the set of messages sent by the malicious node. Note, Gatling does not require the developer to provide a malicious implementation. Instead, Gatling requires type-awareness of the messaging protocol, and applies the basic actions, designed to imitate malicious implementations, described in the next section to the *outputs* of a non-malicious node implementation. To measure the impact of the malicious action, Gatling executes an impact score function, considering only the nodes not flagged as malicious. In this way, Gatling conducts searches for performance attacks without a full-fledged search of possible system behaviors.

3.3.2 Malicious actions

An insider attacker can globally influence the performance of the system by misleading other honest participants through exchanged messages. We classify all malicious actions on messages into two categories, message delivery actions and message lying actions. Message delivery actions refer to *how* a message is sent, while message lying actions refer to *what* a message contains. The list we present is not an exhaustive list and can be easily extended by adding additional delivery or lying strategies. Below we describe the specific malicious actions we consider.

Message delivery actions. Performing message delivery actions does not require knowledge of the messaging protocol, because the actions are being applied to where and when the message is delivered, rather than modifying the message contents. We define the following types of malicious message delivery actions.

- *Dropping*: A malicious node drops a message instead of sending it to its intended destination. Dropping a message can prevent the intended destination node from receiving information and often it can cause benign nodes to have incorrect information about the network or the system. Particularly, probabilistic dropping can trigger a recovery process in benign nodes.

- *Delaying*: A malicious node does not immediately send a message and injects a delay. The delay amount can be one of the predefined values, which are 0.5, 1, 1.5 or 2 seconds. Once the amount is chosen, Gatling injects the chosen amount of delay every time a malicious node sends a message. We added one more option that we call "progressive delay". When this option is enabled, the amount of delay is very small at the beginning, 200ms, and slowly increases over time. In our implementation, the amount of delay increases by 100ms every time a malicious action is injected. This option can be helpful when the target system has some protection that designed to monitor the behavior of the system. Many distributed systems utilize network status, such as delay between nodes, to achieve better performance. Delaying can provide an incorrect and inconsistent view of the network and disturb performance optimization mechanisms of the system.

- *Diverting*: A malicious node does not send the message to its destination as intended by the protocol, and instead enqueues the message for delivery to a node other than the original destination. The destination is randomly chosen from the set of nodes in the system. If specified in the configuration file, the diverting target can be chosen to be only a benign node or only a malicious node. Diverting a message can cause the same effect to dropping to the original destination node and the new destination node get information that was not expected.

- *Duplicating*: A malicious node sends a message twice instead of sending only one copy, applying delay to the second copy. The amount of delay is by default one second and if desired, the user can provide a different value as a parameter. We consider two versions of message duplication. One is to send the duplicated message to the original destination again, and the other is to divert the duplicated message to another random destination in the system. The second copy of the message can cause the destination node to have an incorrect understanding about the state of the system or repeat processing of the message. The option of injecting delay or changing the destination can add delaying and diverting effect upon duplicating.

Message lying actions. We define message lying actions as actions where malicious participants modify the content of the message they are sending to another participant. An effective lying action involves intelligently modifying fields of messages to contain contents likely to cause different behaviors, which is more sophisticated than random bit-flipping. Consider, for example, a simple attack which is to modify a boolean value in a message. While random bit-flipping can easily change the right bit, knowing which bit is the right bit to flip requires either costly trial-and-error, or sophisticated static analysis of the code.

Gatling makes data-type-specific changes to message contents. These changes are dependent on the messaging protocol. As the number of possible values that the message field could contain may be extremely large, we define a few general strategies for field types that stress the system in different ways based on general experience on the kind of error cases or hand-crafted attacks observed in practice previously. We provide the following strategies for numeric types.

- *Min or Max*: A malicious node can change the value to be the minimum or maximum value for the type. These two values are useful to find attacks that exploit inadequate overflow handling.

- *Zero*: For signed types, a malicious node can additionally change the value of the field to be the number 0. The zero value is helpful to find attacks related to exception handling code, such as division by zero.

- *Scaling*: A malicious node could increase or decrease the numeric value by a percentage. The goal of scaling is for malicious nodes to attempt to lie based on the original value in order to make the value seem more legitimate. Hence we use a small percentage such as 10% of the original value. However, this amount can be too small to trigger any impact, while the lying still needs to be subtle. Therefore, we implement another option, progressive scaling, where a malicious node scales a value by a small percentage at the beginning and then increases the percentage over time. In our implementation, starting from 0.001%, the amount of scaling increments by 0.00005% every time. It allows Gatling to inject an attack that is subtle at the beginning and becomes more aggressive over time.

- *Spanning*: A malicious node can select specific values from a set which spans the range of the data type. Spanning values are important because protocols sometimes use only a subset of legal values, apply sanity checks to inputs, or fail to apply sanity checks when necessary to avoid *e.g.* overflow/underflow. Spanning values can be chosen assisted by static analysis or developer insight; we find that a range of values orders of magnitude apart are sufficient to find attacks in many systems.

- *Random*: A malicious node can select a random value from the range of the type. The random value can let the recipient believe the state of the system is unstable or can find attacks in the style of random fuzzing.

In addition to the above choices, boolean values have an additional option: toggling the value between true and false. The list can be easily extended, for example, using a "complement" strategy for integral values (a generalization of the boolean flipping).

Node identifiers, such as an IPv4 address or a hash key, are integral aspects of distributed systems. Thus, we treat them as a native type and allow lying on them as well. Malicious nodes can lie about node identifiers, where lying values are selected randomly from the identifiers of all nodes, malicious nodes, or benign nodes.

We also have special handling for non-numeric types. For simplicity, collections (e.g. list, set, map, etc.) are treated as applying one of the above strategies to all the elements within the collection. Users can further extend Gatling as needed to provide lying strategies for additional types as we have done for node identifiers.

Applying malicious actions.

By default, Gatling injects all malicious actions on outgoing messages sent by malicious nodes. Therefore, the internal states of malicious nodes do not change. To explore possibilities where a malicious node operates on an incorrect state, we implement an option to inject attacks on incoming messages. When this option is set, Gatling injects malicious actions on incoming messages upon receiving. The malicious node processes the incorrect message and operates on the resulted state that can be incorrect.

3.3.3 Discovering attacks

A naive approach to discovering attacks is executing all possible sequences of actions (malicious and benign) in the system and then finding the sequences that cause performance to degrade below the benign case scenario. However, this approach becomes intractable because of the size of the search space considering the number of possible sequences of actions. Specifically, at every time step, any benign event could execute based on timings, but additionally, any malicious node could generate any message defined by the system, performing any combination of malicious actions on it and send it to any node. Considering all possible attack values for a message containing a single 32-bit integer entails an exploration branching at the impractical rate of at least 2^{32}. Benign state-space exploration is shielded from this problem by the fact that while the message field could theoretically contain any of the 2^{32} values, at any point in time only a small subset of those values would be sent by a non-malicious node.

Attack properties. As a first step toward practical automated search of attacks, we focus on a class of performance attacks that have several properties that reduce the state-space exploration needed to discover them:

1) Single-behavior: We define a single-behavior attack as a list which describes, for each type of message, what malicious or benign action all malicious nodes will take whenever sending a message of that type. Intuitively, this attack definition is based on the principle that in some cases, past success is an indication of future success. Thus, every time a malicious node must decide how to behave when sending a message, it can choose the same action that succeeded before, and expect success again. This allows us to reduce the search space of malicious actions substantially, because once we have discovered a successful malicious action for a message type, we no longer explore other possibilities for the same type of message sent by any malicious node.

This decision also has an important usability benefit. By focusing on a clear attack strategy, attacks discovered are immediately recognizable. By comparison, the naive search strategy would instead require a very complex post-processing step which attempts to discern, “What is the generalized attack?”

Note that while a single-behavior attack has a single malicious action for any message type, since different message types can have different behaviors, thus, combination of attacks or inflation attacks can be found together by Gatling. For example, Gatling can find an attack which both drops application data and lies during system self-organization to gain the most leverage.

2) Easily reproducible: We assume attacks that are not largely dependent on the specific state of the distributed system and thus can be easily reproduced. Intuitively, the attacks we discover are those to which the system is generally vulnerable, rather than having only a small vulnerability window. Easily reproducible attacks allow us to safely ignore the particular sequence of benign actions that occur alongside the malicious actions and focus our search solely on malicious actions.

3) Near-immediate measurable effects: We consider attacks that do not have a large time-lag between when they occur and when the performance of the system is actually affected. Intuitively, focusing on near-immediate effective attacks will be ideal for finding direct attacks on system performance, but it will not allow Gatling to discover stealth attacks, where the goal is to obtain control without affecting the performance of the system under attack. The near-immediate impact on the system performance of the attacks creates the opportunity to find attacks by only executing a smaller sequence of actions for a relatively short window of time. We decide if a malicious action is a possible attack by using an impact score I_s function that is based on a performance metric of the system and is provided by the user. We require two properties of the impact score. One, that it can be evaluated at any time during the execution. Two, that when comparing scores, a larger score implies that the performance is worse.

4) Most effective minimal combination: While Gatling will discover single-behavior attacks that contain basic behaviors for many message types, some behaviors will have only a nominal impact on the performance of the system, and other behaviors may be quite effective as stand-alone attacks. To allow the developer to discern these cases, Gatling automatically determines the relative contribution of each attack action to the overall performance degradation, allowing the developer to further reduce the attack to their minimal portions that have the most significant impact.

Gatling builds up an attack by finding several instances where applying a malicious action on a message results in an increase in the impact score, then building up the maximally effective single-behavior attack across message types. Once Gatling has found the maximally effective single-behavior attack it then determines the contribution of each malicious action of the attack. If the attack is effective, i.e., the performance is worse than Δ, Gatling reports to the user the attack and the contribution of each action. We provide additional automation to help find another possible attack in the system. If this option is enabled, the entire search process is repeated to find additional attacks. However, previous malicious actions that were found to be effective are no longer tried.

Greedy action selection procedure. To find an instance where a single malicious action results in an increase in the impact score, we use the procedure depicted in Fig. 3.2. The main idea is to execute the program normally until a malicious node attempts to send a message. At this point we branch the execution and run on each branch the malicious version of the sending of the message (try all malicious actions described in Section 3.3.2) and then continue running the branch for a window of time t_w. By measuring the impact score at the end of each window of execution, we can determine whether any of the malicious actions degraded the performance relative to the baseline without a malicious action. Since we measure the impact of only a single malicious action instance, we consider any increase in the impact score to suggest that the particular malicious action could be a part of a successful attack. We greedily select the strongest such malicious action and update a tally for that message type and malicious action.

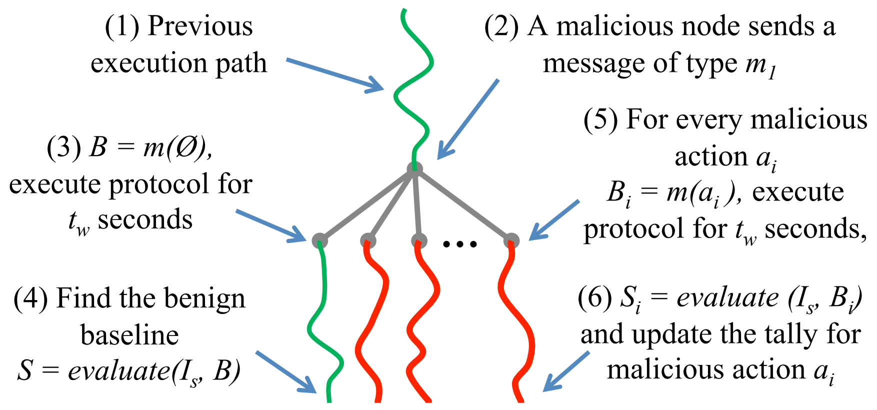

(1) Previous execution path
(2) A malicious node sends a message of type m_1
(3) $B = m(Ø)$, execute protocol for t_w seconds
(4) Find the benign baseline $S = evaluate(I_s, B)$
(5) For every malicious action a_i $B_i = m(a_i)$, execute protocol for t_w seconds,
(6) $S_i = evaluate(I_s, B_i)$ and update the tally for malicious action a_i

Figure 3.2.: Greedy action selection procedure for one instance of sending message type m_1

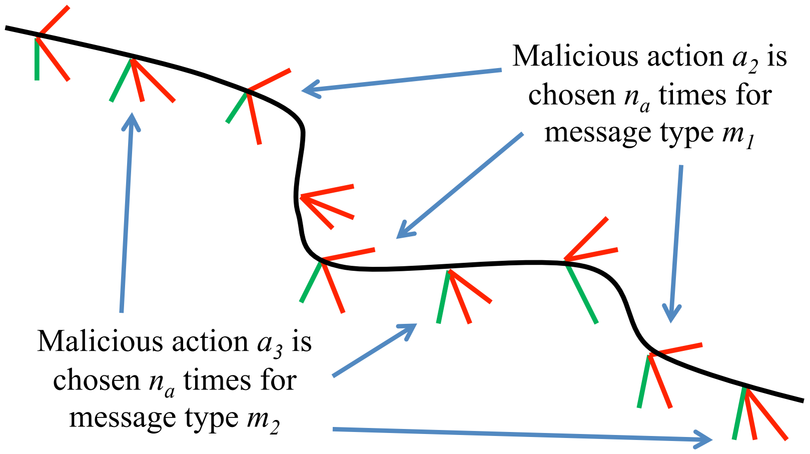

Malicious action a_2 is chosen n_a times for message type m_1
Malicious action a_3 is chosen n_a times for message type m_2

Figure 3.3.: Greedy procedure applied for several instances of message types m_1 and m_2

Table 3.1: Notations in algorithms

Notation	Description
δ	Performance difference between a malicious execution and the baseline (benign execution).
Δ	Performance difference threshold indicating an attack.
n_a	Number of times that a malicious action must be chosen to be considered an attack. It is used to determine if the attack is reproducible.
t_w	The window of time for which the system is executed after the attack injection point. At the end of it the impact score is evaluated.

Building up the single-behavior attack. The greedy action selection procedure finds the most effective malicious action for a single instance of sending a message. We report a malicious action as part of an attack once it has been selected in n_a *different instances* for the same message type. The n_a threshold allows us to avoid cases in which the impact was a random variation, and provides a balance between attacks which are more frequently successful but with a lower impact, and attacks which are less frequently successful but with a higher impact.

If we only wished to find attacks incorporating a single message type at a time, we could at this point simply iterate through the set of message types, and perform the greedy procedure each time that message type is sent. While successful in finding single-behavior,

ALGORITHM 1: Attack discovery algorithm

```
variables:
  array $A(msgType)$: Actions applicable for a message type
  vector $Attack$: Learned behaviors for most effective attack for each message type
  map $AttackAndContribution$: Attack listing relative contribution of actions
  matrix $AttackTally$: Count, for each message type, the times the attack is
determined most effective
  $IneffectiveTally$: The number of times no malicious action is chosen consecutively
(cont.)
```

```
while $\mathit{IneffectiveTally} < \mathit{HaltingThreshold}$ do
    Continue simulating system until malicious node sends a message $m$;
    $\mathit{msgType} \leftarrow \mathit{typeof}(m)$;
    $\mathit{MostEffectiveAction} \leftarrow \mathit{Attack}[\mathit{msgType}]$;
    if $\mathit{MostEffectiveAction} = \emptyset$ then
        $maxScore \leftarrow MIN$;
        foreach $a_i \in A(\mathit{msgType})$ do
            apply action
            sleep $t_w$;
            if $currentPerformance > maxScore$ then
                $maxScore \leftarrow currentPerformance$;
                $\mathit{MostEffectiveAction} \leftarrow a_i$;
            end
        end
        if $\mathit{MostEffectiveAction} \neq \emptyset$ then
            $\mathit{AttackTally}[\mathit{msgType}][\mathit{MostEffectiveAction}]{+}{+}$;
            $\mathit{IneffectiveTally} \leftarrow 0$;
            if $\mathit{AttackTally}[\mathit{msgType}][\mathit{MostEffectiveAction}] = n_a$ then
                $\mathit{Attack}[\mathit{msgType}] \leftarrow \mathit{MostEffectiveAction}$;
            end
        else
            $\mathit{IneffectiveTally}{+}{+}$;
        end
    end
    execute behavior $m(\mathit{MostEffectiveAction})$;
end
(cont.)
```

```
$AttackAndContribution, \delta$ = computeActionContribution($Attack$);
if $\delta > \Delta$ then
    output $AttackAndContribution$;
    if $continueSearch = true$ then
        Repeat, ignoring prior Attack actions
    end
end
```

single-message attacks, this approach would not find *dependent* attacks, where the success of an attack is conditional on a prior malicious action choice. Consider for example the case of a malicious node which lies to increase the number of children it has in a tree overlay. If the malicious node does not also perform an action on application data, then this kind of attack would not be discovered using single-message attacks.

To discover dependent attacks, Gatling simultaneously searches across all message types, allowing it to find combination attacks, where individual malicious actions work together to build stronger overall attacks. By applying the greedy action selection procedure to the instances as they are encountered, rather than iterating through message types, our algorithm can locate amplifying stealth attacks without prior knowledge of the order in which malicious actions must occur. Specifically, the system is simulated normally until a malicious node attempts to send a message of a type for which an attack has not been identified. The greedy selection procedure is used to determine the best action to take for this instance, and a tally is kept of the times each malicious action was chosen. The number of times no malicious action is selected in a row is also tallied, as a means to halt the search. We show in Fig. 3.3 the greedy procedure being applied to several instances for two different types of messages.

Once the search halts, the contribution of each of the actions is computed, and if the attack impact is greater than some Δ, the user is notified, and the algorithm repeats but does not search on previously used malicious actions. Computing the action contribution involves running the system again for an extended period both with and without the determined attack. This allows Gatling to verify that the attack satisfies the requirement that its

ALGORITHM 2: Compute action contribution

```
variables:
    vector $Attack$: Learned behaviors for most effective attack for each message type
    vector $perfSingle$: Individual performance for each behavior in the vector
    vector $perfVector$: Performance vector
foreach behavior in $Attack$ do
    perfSingle[behavior] = executeWithBehavior(behavior);
    if $perfSingle[behavior] > \delta$ then
        Report($perfSingle[behavior], behavior$) ;
    end
end
foreach behavior in $Attack$ do
    perfVector[behavior] = executeWithBehavior($Attack$ - behavior);
    if $perfVector[behavior] > \delta$ then
        Discard($Attack, behavior$) ;
    end
end
return $Attack$;
```

impact is greater than Δ. Δ is a threshold to determine if an attack is successful and it needs to be decided by the user because how much performance degradation is tolerable varies for each system, metric, and even for the environment. For example, for the same multicast system, a longer latency to deliver a lecture can be tolerable than a latency acceptable to deliver a sports game. Gatling then determines the relative contribution of each component by running additional tests, subtracting out the least effective contributor until it is no longer an attack. This computation and sorting procedure is important for three reasons. First, as a greedy approach, it is possible that Gatling finds a local maximum, but that the order in which malicious actions were selected diminished the overall impact (e.g. an attack may later be found which by itself is more potent than when combined with the earlier attack). Second, some malicious actions may depend on other malicious actions for success, this search will order them accordingly. Third, some malicious actions may have only a minor impact, or a strong enough impact to be used in isolation, this post processing can provide this information. In fact, we often find multiple attacks from a single run of the Gatling search algorithm. We present our attack discovery in Algorithm 1 and the computation of each action contribution in Algorithm 2. Notations in the algorithms are summarized in the Table 3.1.

Impact score and parameter selection. The user must specify an impact score. As stated, the impact score must be able to be evaluated at any time, rather than only at the completion of an execution, and must let greater values indicate a greater impact. Consider, for example, an impact score for a file download system. Using total download time as an impact score would satisfy the requirement that bigger numbers indicate more impact (slower download times), but fails the requirement that it can be evaluated at any time (it can only be evaluated once the file is downloaded). The current average goodput of the file download satisfies the requirement that it can be evaluated at any time, but in the case of goodput, bigger numbers actually mean less impact. An alternative might include an inversion of the goodput, or instead it could simply be a measure of how much of the file is left to download.

Gatling requires the setup of three parameters, $/Delta$, the performance threshold indicating an attack, t_w, the window of time for which some path will be executed and the impact score evaluated, and n_a, the number of times a particular action has a negative effect on the impact score. Larger values of t_w increase the search time while smaller values may not capture the effects of the malicious action on performance. In the case of n_a, its setup should take into account the normal variability of performance in the system that is evaluated.

Repeating action. As noted in Section 3.3.3, Gatling tries to find an action that has a near-immediate measurable effect. It is not ideal for finding stealth attacks as they will not show an impact in a short window of time. Moreover, unless learned, Gatling injects a malicious action only once during a window of time to see the impact of a single action.

These two properties, near-immediate impact and single injection in a window, combined together prevent Gatling from finding stealth attacks. In stealth attacks, attackers inject malicious actions where the impact will be subtle at the beginning and become more significant over time to avoid being caught. Though the requirement of near-immediate impact can be addressed by increasing t_w, Gatling will not find such an action if the action needs to be repeated.

To allow Gatling to find such an attack, we implement a new option to repeat an action during the window of time. When this option is enabled, Gatling injects the selected action repeatedly every time the chosen message type is sent.

Halting Strategy. The initial halting strategy was to stop searching when Gatling selects a benign action a certain number of times in a row. Gatling then stops the search and evaluates the contribution of each action in the current attack vector. At times, however, depending on the system and performance impact of actions, we observed that it ran several hours without halting nor adding new attacks.

The delayed halting can happen because the benign action is not always chosen when there is no effective malicious action on a message type. If no action can cause a significant impact on the performance, all actions will show similar performance scores. Note that the performances will be similar, not identical. The states of nodes can be different at the beginning of each window and Gatling does not care if the amount of performance difference at a specific moment is significant. Therefore, it is possible that an action that is not actually malicious to be occasionally chosen. As Gatling counts the number of consecutive benign selections, this scenario can delay halting significantly.

The delay of halting causes another problem. After the system reaches the local maximum performance, it can continuously add up minor impact actions. This does not cause a correctness problem as Gatling computes the contribution of each action afterwards. However, as Gatling does not halt easily, many minor actions can be added to the attack vector and this increases the time to compute contributions as well.

To address the issues caused by the delay in halting the search algorithm, we added new halting conditions. Other than Δ, a threshold to decide if the performance is bad enough to be defined as an attack, we require the user to provide another threshold $perfLocalMax$, which specifies if the system has a reached a local maximum. When Gatling evaluates the performance score, if a system reaches $perfLocalMax$, Gatling halts the search as the attack is significant enough, and start to compute the contributions of the elements in the attack vector.

Note that some systems require stabilization time; for example, a peer to peer file sharing system needs time for nodes to join before it can start to transfer files. In such systems, the initial performance before stabilization will be poor and it can cause Gatling to halt the search prematurely although it is not a result of attacks. To address this problem, we allow a user to provide a stabilization time and let Gatling only uses $perfLocalMax$ after that stabilization time has passed. If a stabilization time has passed and the performance score is bigger than $perfLocalMax$, it means the system failed to stabilize or after it stabilized the performance degraded due to an attack.

Finally, the algorithm can halt prematurely when some actions have very significant impacts with only few tries. For example, if an action m_a can cause a system to reach $perfLocalMax$ with only three tries and the user provided n_a with a value of five, Gatling can halt the search before it learns the action. To prevent premature halting before learning, Gatling waits until at least one action is learned before evaluating if the performance met $perfLocalMax$.

The resulting halting strategy along with the halting conditions are presented in Algorithm 1.

3.4 Implementation

In this section we describe our implementation of the Gatling design. We created a concrete implementation of Gatling for the Mace [5] toolkit. Mace is publicly available and was designed for building large-scale, high-performance distributed systems implementations based on C++. It consists of a source-to-source compiler, a set of runtime libraries, as well as a model checker and time-based simulator. The release also includes several distributed systems implementations. The Mace compiler enforces the *message-event model* and generates implementations of message serialization, both useful for Gatling. Specifically, the message event-model allows us to influence message delivery, while message serialization allows us to implement message lying without modifying the target system code, just by defining specific lying actions for different types. The Mace toolkit also includes a model checker and time-based simulator, which the authors demonstrated can find high-level bugs in distributed systems implementations [37, 54]. When a system is written in Mace, one can run the system both on the simulator or on real environment, and it makes it an ideal platform for Gatling.

To implement Gatling we made the following changes to Mace. We added an interposition layer between Mace services and the networking services, we modified the Mace compiler to include a message serialization code injector, we added supporting serialization code in the Mace runtime library, and we modified the simulator to implement our attack discovery algorithm. The user provides an implementation of the distributed system in Mace and specifies an impact score in a simulation driver that allows the system to run in the simulator. The Mace compiler will generate the message serialization injected code in the user code.

This modular design allows code reuse and allows Gatling to focus attacks on modules independently. The interposition layer implements *malicious message delivery actions*. When a node requests sending a message, before providing the message to the network messaging services, Gatling consults the attack discovery algorithm to decide whether to take any message delivery action. Message dropping, delaying, diverting, and duplicating are provided by either not making the call to the messaging services, queueing the message for sending 0.5 to 2 seconds later, calling into the messaging services multiple times, or passing a different destination to the messaging services. To support diverting messages, the simulator provides lists of malicious and benign node identifiers.

Malicious nodes can be randomly selected or manually specified. In the random selection case, a user can specify the percentage of malicious nodes in the system. In the manual specification case, a user can provide a list of malicious nodes in the configuration file. The latter approach is useful to find attacks that leverage certain node distributions or roles of nodes.

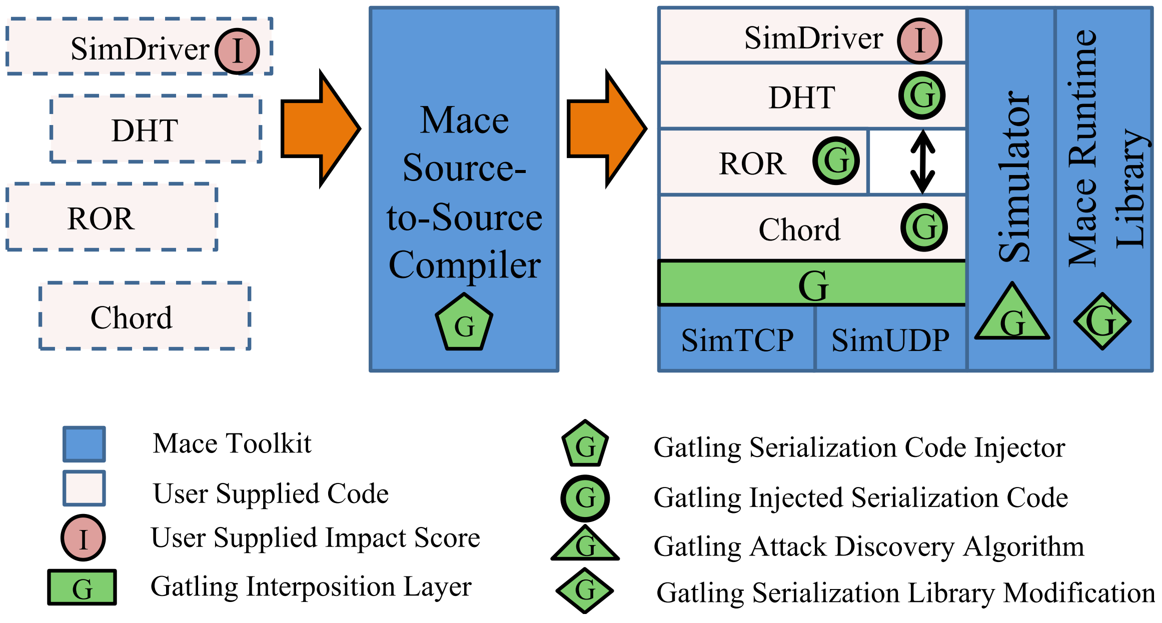

SimDriver
I
DHT
ROR
Chord
Mace Source-to-Source Compiler
G
SimDriver
I
DHT
G
ROR
G
Chord
G
G
SimTCP
SimUDP
Simulator
G
Mace Runtime Library
G
Mace Toolkit
User Supplied Code
User Supplied Impact Score
Gatling Interposition Layer
Gatling Serialization Code Injector
Gatling Injected Serialization Code
Gatling Attack Discovery Algorithm
Gatling Serialization Library Modification

Figure 3.4.: Gatling implementation for one node: DHT example

The injected serialization code component implements *malicious message lying actions*. The injected code similarly consults the attack discovery algorithm to determine if a lying action should be taken. As we are searching for single-behavior attacks, the simulator directs only a single field in a message to be lied about during one branch of the greedy selection procedure. If any lying does occur, when serializing the appropriate field of the message a simulator chosen value is used instead of the one provided. The user-written code is not modified, nor are any user-visible variables. Simulator-provided lists are similarly used to lie about node identifiers.

Fig. 3.4 shows the architectural design of Mace and Gatling when testing a layered DHT application. The parts noted with G represent the Gatling additions and modifications. The user provides each DHT component layer in the Mace language (shown at left): a simulation driver (*SimDriver*), containing the impact score function; the storage layer (DHT); a recursive overlay routing layer (ROR); and the Chord lookup service layer. The Mace compiler then translates each layer into C++ code, injecting message lying actions into each layer tailored to the messages defined by that layer. Standard C++ tools then compile and link the generated code with the Gatling interposition layer, Mace runtime library, simulated TCP and UDP messaging services, and the Mace simulator application. *SimDriver* allows the application to run in the simulator; to deploy the DHT application, the C++ code need only be re-linked with the real TCP and UDP messaging services, and a C++ user application in lieu of *SimDriver*.

3.5 Case study: BulletPrime

In this section we demonstrate how to use Gatling to find attacks on a real system implementation. For our case study we apply Gatling to an implementation of the BulletPrime peer-to-peer file distribution protocol [13, 51] that we received from the authors of the system. We selected BulletPrime as a case study because it uses a more complex design involving several services. While we illustrate how a developer might use Gatling to find attacks arising from a malicious or simply misconfigured node, our intention is not to criticize BulletPrime's design. Instead we explore its behavior in an adversarial environment that many practical uses might require.

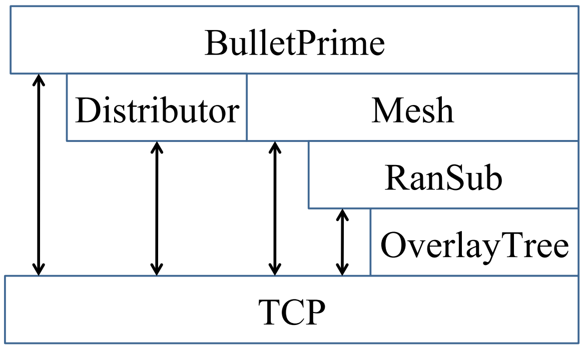


Figure 3.5.: BulletPrime design

3.5.1 BulletPrime

BulletPrime is a file distribution system similar to BitTorrent [55]. However, where BitTorrent focuses on local optimizations that greedily benefit each node individually, BulletPrime uses a more collaborative set of algorithms that are geared towards global optimization. For example, while both BitTorrent and BulletPrime implement mesh-based strategies for peering, and use rarity as a mechanism for increasing block diversity, BulletPrime learns about new peers by using a gossip protocol that guarantees each node receives a uniformly random distribution of peers and their current download status. BulletPrime also searches independently for peers that can provide maximal download or upload bandwidth, as opposed to BitTorrent's symmetric block exchange algorithm.

The BulletPrime component design is illustrated in Fig. 3.5. The BulletPrime service manages the state of the file download, implements sending Diff messages to connected peers with information of newly available blocks of the file; and tracks the performance of the peers. It utilizes the Distributor service to manage the queued Data messages to each peer, keeping the network buffers full without sending excess data. BulletPrime uses the Mesh service to learn of new peers and maintain active connection to upload and download peers. The Mesh service sends Join and JoinReply messages, and uses the RanSub [8] service to discover potential peers. RanSub, meanwhile, uses an overlay tree to perform a specialized type of aggregation, proceeding in periodic phases that Collect candidate sets of information to the root, and then Distribute uniformly randomized candidate sets to all peers.

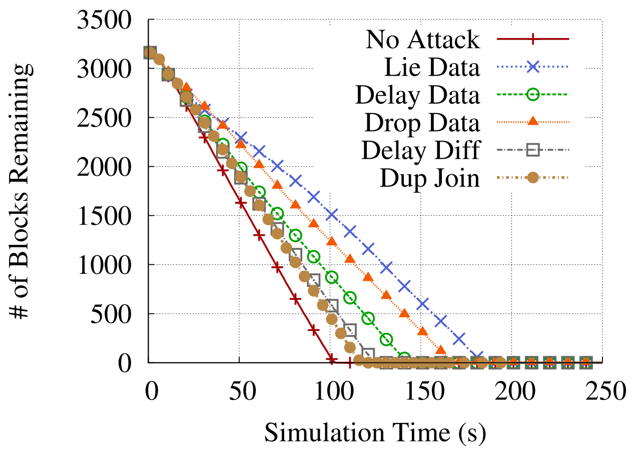
3.5.2 Applying Gatling on BulletPrime

To run Gatling on BulletPrime, we first had to prepare it to run in the simulator and implement an impact score. We wrote an 85 line simulated application driver that provides the basic functionality of having the source node distribute data to others and having the client nodes download and participate in the file-sharing protocol. When choosing an impact score, our first intuition was to use the file download completion times, since this is the metric on which BulletPrime is evaluated. However, the completion times are not known until the run is completed, while the impact score needs to provide meaningful information over any practical time window. Instead, we chose for the impact score a performance metric which captures the progress of node downloads; namely the number of blocks of the file downloaded. To satisfy the requirement that a higher score indicates more attack impact, we instead use the total number of blocks *remaining* before completion. We had to modify BulletPrime slightly, adding the 8-line impact score function, because it did not expose enough information to the simulated driver to compute the score. We simulated BulletPrime with 100 nodes disseminating a 50MB file. We use a small 5 sec t_w as nodes download blocks quickly, starting nearly at the beginning of the simulation. Due to some variable system performance, we set n_a to 5, allowing Gatling to explore a few instances per message type. We set Δ to be zero for all our experiments to find as many attacks as possible.

3.5.3 Attacks found using Gatling

Assertions and segmentation faults: As we began to use Gatling on the BulletPrime implementation, we encountered nodes crashing due to the fact that BulletPrime assumes peers to act correctly. For example, we found nodes crashing due to assertions and segmentation faults when receiving a malicious FileInfo message. This message defines the file and block size and is created by the source. Intermediate nodes that forward the message can lie about its contents when passing it along. We found another crash scenario when a malicious node requests a non-existing block, causing the recipient to crash by assertion attempting to retrieve the block. We implemented checks in the code to prevent crashing and we disabled any attack on FileInfo for further Gatling simulations.

We found another crash scenario where a malicious node lies on incoming RequestDataBlocks message. We first searched attacks on outgoing messages only then searched again on incoming messages. In most of the cases, effective attacks on incoming attacks have the same semantic to those attacks found on outgoing messages. For example, we found an attack that drops incoming RequestDataBlocks and an attack that drops outgoing Diff. Although they seem like different attacks, RequestDataBlocks are required to send Diff and thus both attacks cause the same result. Although most of attacks discovered



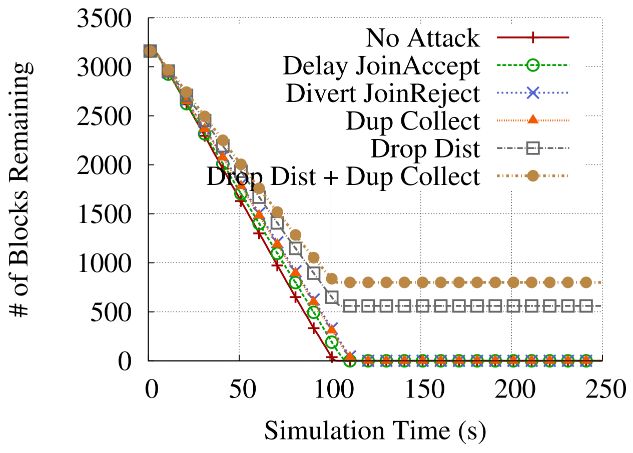
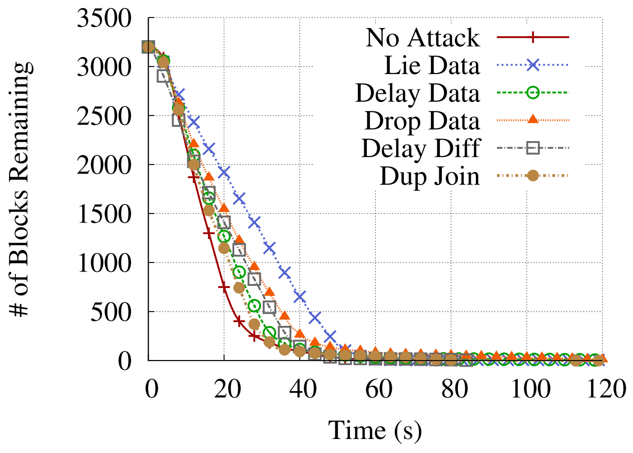


Figure 3.6.: Remaining blocks for the attacks found on BulletPrime



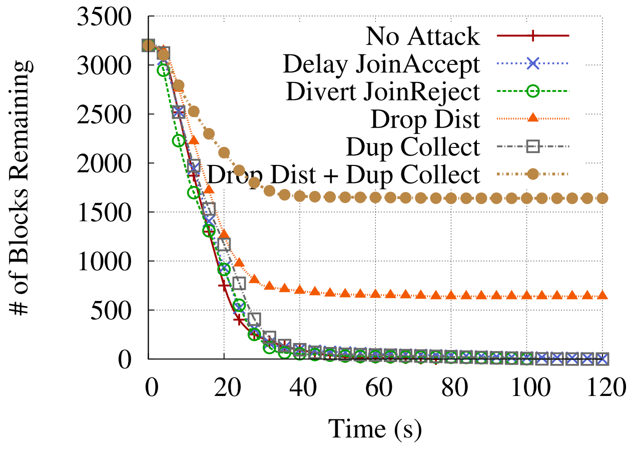


Figure 3.7.: Remaining blocks for the attacks found using Gatling on Planet-Lab deployment of BulletPrime

using incoming message are semantically the same attacks as the attacks discovered using outgoing messages, we discovered an attack that causes an assertion error on a benign node that was not found by only searching outgoing messages.

Fig. 3.6 shows the performance of the system under the attacks we discover. To give a baseline comparison, we also show the benign scenario when there is no attack. We have found attacks against four of the five services.

Distributor service: We found several attacks on Data messages. Lying on the id field of a Data message degrades the performance significantly. We also found dropping or delaying Data causes performance degradation.

BulletPrime service: Furthermore, Gatling found a delaying attack on the Diff message which causes a performance penalty, since peers cannot request the block until receiving the Diff message.

Mesh service: Gatling also reported an attack vector that is a combination of 1) duplicate Join message and divert the second copy to a random destination, 2) delay JoinAccepted message for 0.5s, and 3) divert JoinRejected message to a random destination. Gatling computed the most effective minimal combination found that all actions are effective even when they are used alone, however the combination of the three was the most effective. We show the individual attacks in Fig. 3.6.

RanSub service: Gatling found an attack which was a combination of dropping Distribute messages that are disseminated from the root toward the leaves over the control tree and also duplicating Collect messages that are collected from leaves towards the root. Gatling found that both actions alone degrade performance and furthermore dropping Distribute messages causes nodes to never be able to download a number of blocks.

While some of the attacks found on BulletPrime were expected, such as delaying or dropping data messages, less obvious was the impact of the attacks on the Mesh and RanSub services. Although BulletPrime gains nice mathematical properties by using RanSub, it seems that a BulletPrime implementation robust to insider attacks may be better served by a gossip service re-design. As an extra benefit, Gatling also identified several cases where insiders can crash system nodes due to a lack of input validity checking.

To validate that the attacks are not a result of lack of fidelity in the simulator we ran real-world experiments with the discovered attacks on the PlanetLab testbed, with the same number of nodes and file size. We confirmed that all attacks have a similar effect as in the simulator and we show graphs in Fig. 3.7.

Fig. 3.6 and Fig. 3.7 show that the attack that malicious nodes dropping Distribute and duplicating Collect has much bigger impact.

3.5.4 Discussion

Here we describe how we can make BulletPrime more secure using attacks Gatling has found.

The first attack Gatling found was lying about file information and the best way to protect a file distribution system from untrustworthy file information is to give one that is safe. The attacks Gatling found show that because nodes in BulletPrime trusts each other, not only it is possible to slow down the progress, but also attacks can prevent benign nodes finishing file download and this is clearly a problem that has to be resolved.

BulletPrime nodes cut peers only when the bandwidth receiving of a peer is significantly (1.5 standard deviations) lower than the average and only when the total number of peers is bigger than half of the maximum number of peers. Therefore, when there are many peers that do not give enough bandwidth, the average becomes low, and the algorithm does not work well. Likewise, since the number of maximum peers changes based on change of recent incoming bandwidth over the incoming bandwidth of the previous window, once the receiving bandwidth drops down to 0, it is not possible to increase the maximum number of peers to add a new peer.

BulletPrime manages the number of outstanding blocks, however, the purpose is not to issue requests efficiently and the mechanism does not protect a node from waiting on a malicious node after requesting its last block. These strategies based on trust become problematic when nodes downloaded almost all the blocks and they have malicious peers. To prevent nodes relying on malicious nodes when they are not actually giving data, we added a strategy that cuts peers based on the difference between estimated performance and recent performance. If a peer is identified as not giving enough performance, we cut the peer though we have less number of peers so that nodes connected to the malicious peer will get a chance to find another peer.

3.6 Results

We further validate the Gatling design by applying it on eight systems with different application goals and designs where three of them are specially designed to tolerate insider attacks. Specifically, we evaluate the Vivaldi [46] virtual coordinate system, the Chord lookup service and distributed hash table (DHT) [16], and two multicast systems: ESM [47] and Scribe [48]. Then, we evaluate secure systems: Vivaldi Outlier Detection [49], Newton [50] and PBFT [2]. Chord, DHT, and Scribe were previously implemented for Mace; we implemented Vivaldi, ESM, Vivaldi Outlier, Newton and PBFT according to published papers. We set the number of malicious nodes to be 20% and we select malicious nodes randomly except in the case of PBFT. For all experiments, we set stabilization time to be the malicious action start time + t_w and $perfLocalMax$ a value that will be too big to serve the purpose of the system. For example, 50 unreachable nodes for Chord and 1000000 ms for Vivaldi.

Gatling found performance attacks in each system tested, taking from a few minutes to a few hours to find each attack. Gatling was run on a 2GHz Intel Xeon CPU with 16GB of RAM. Gatling processes are CPU bound, so parallelizing the search could further reduce the search time. We discovered 48 attacks in total, however due to lack of space we only present in detail a subset of attacks that illustrate the capabilities of Gatling. In Table 3.2 and Table 3.3, we summarize all the attacks.

Table 3.2: Attacks found using Gatling on general distributed systems: 41 attacks in total (17 lie, 12 drop, 6 delay, 5 duplicate, 1 divert)

System	Metric Used	Attack Name	Attack Description	Known Attack
Bulle Prime	Number of Blocks Remaining	Lie Data	Lie data message distribution	
		Delay Data	Delay data message distribution	
		Drop Data	Drop data message distribution	
		Delay Diff	Delay diff information	
		Dup Join	Duplicate join message and send copy to another	
		Delay JoinAccept	Delay join accepted	
		Divert JoinReply	Send join rejected to another node	
		Drop Dist	Drop information distributed	
		Dup Collect	Dup information collected	
Vivaldi	Prediction Error	Overflow	Lie about coordinates, setting them to maximum value	[49]
		Inflation	Lie about coordinates, setting them to large values	
		Oscillation	Lie about coordinates, setting them to random values	
		Delay	Delay probe reply messages 2s	
		Deflation	Do not initiate request (Drop probes)	
Chord	Number of Reachable Nodes	Drop Find Pred	Drop query to find predecessor	[56]
		Drop Get Pred	Drop query to get predecessor and successor	
		Drop Get Pred Reply	Drop the answer to find predecssor	[57]
		Lie Find Pred	Lie about key that is in query while forwarding queries	
		Lie Predecessor	Lie about predecessor in response while forwarding	
		Lie Successor	Lie about successor candidates in response while forwarding	

continued on next page

System	Metric Used	Attack Name	Attack Description	Known Attack
DHT	Lookup Latency	Delay Msg	Delay recursive route messages	
		Drop Msg	Drop recursive route messages	[57]
		Dup Msg	Delay recursive route messages and divert second message	
		Lie Msg Src	Lie about the source of recursive route messages	
		Lie Msg Dest	Lie about the destination of recursive route messages	
		Lie SetKeyRange	Lie about what keys are stored	
		Lie Reply Key	Lie about the key in get reply messages	[57]
		Lie Reply Found	Lie about finding the value in get reply messages	
		Lie Reply Get	Lie about the request wanting the value in get reply messages	
ESM	Throughput	Dup Parent	Duplicate parent reply messages, drop data	
		Drop Data	Drop data messages	[58]
		Lie Latency	Lie about measured latency, duplicate probe messages, drop data	
		Lie Bandwidth	Lie about received bandwidth, duplicate probe messages, drop data	
	Latency	Drop Parent	Drop parent reply messages	
Scribe	Throughput	Drop Data	Drop data messages	
		Drop Join	Drop join messages	
		Dup Join	Duplicate join messages	
		Dup Data	Duplicate data messages, sending second message to random node	
		Drop HB	Drop heartbeat message	
		Lie GroupId HB	Lie about the group identifier in a heartbeat message	

continued on next page

System	Metric Used	Attack Name	Attack Description	Known Attack
		Lie GroupId Join	Lie about the group identifier in a join message	

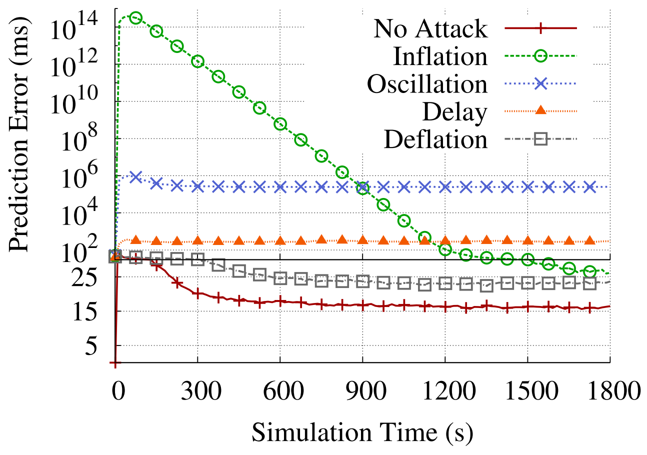
3.6.1 Vivaldi

System description. Vivaldi [46] is a distributed system that provides an accurate and efficient service that allows hosts on the Internet to estimate the latency to arbitrary hosts without actively monitoring all of the nodes in the network. The system maps these latencies to a set of coordinates based on a distance function. Each node measures the round trip time (RTT) to a set of neighbor nodes, and then determines the distance between any two nodes. The main protocol consists of Probe messages sent by a node to measure their neighbors RTTs and then a Response message from these neighbors is sent back with their current coordinates and local error value.

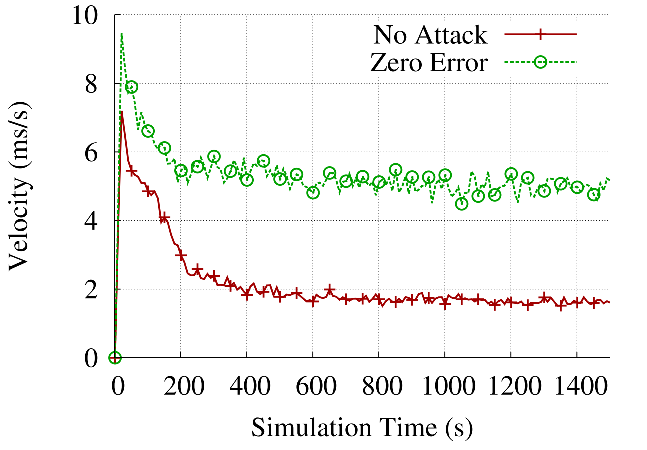
Impact score. We use the *prediction error* [46] which describes how well the system predicts the actual RTT between nodes and *velocity* [59] which measures system stability. Prediction error is defined as *median*$(|RTT^{i,j}_{Est} - RTT^{i,j}_{Act}|)$, where $RTT^{i,j}_{Est}$ is node i's estimated RTT for node j given by the resulting coordinates and $RTT^{i,j}_{Act}$ is the most recently measured RTT. Velocity is defined as *mean*$(\frac{\Delta x^i_{t_w}}{t_w})$, where t_w is the time window, and $\Delta x^i_{t_w}$ is the sum of changes in coordinates for node i over the last t_w seconds.

Experimental setup. We simulated 400 nodes and randomly assign RTT values for each node from the KING data set [60] which contains pair-wise RTT measurements of 1740 nodes on the Internet. Malicious nodes start their attacks from the beginning of the simulation. We set t_w to be 5 sec and n_a to be 5.

Attacks found using prediction error. We found five attacks using the prediction error impact score. In Fig. 3.8(a) we show how each attack affects Vivaldi prediction error over time. The Overflow attack is omitted because the prediction error was *NaN* (not a number).



(a) Prediction error for the attacks found



(b) Velocity for the attacks found

Figure 3.8.: Attack impact on Vivaldi

As a baseline we also present Vivaldi when there are no attacks, in which case we find the system to converge to a stable set of coordinates with 17 ms of error.

Overflow. We first found two variations of an attack where malicious nodes lie and report DBL_MAX for their coordinate and their local error, respectively. In both cases the result is that honest nodes compute their coordinates as *NaN*. We implemented safeguards to address the overflow.

Inflation, oscillation, deflation. We then found three previously reported attacks against Vivaldi [49]. The first, known as inflation, occurs when malicious nodes lie about their coordinates, providing larger than normal values from the spanning set without causing overflow. The second, known as deflation attack, occurs when where malicious nodes drop outgoing probes, thereby never updating their own coordinates. They then respond truthfully to probe requests, stating they are at the origin of the coordinate space. Benign nodes converge to positions with more error than the baseline case. This attack increases the error modestly to 23 ms. The third, known as the oscillation attack, occurs where attackers set their coordinates to random values. This is a very effective attack in which nodes cannot converge and the prediction error remains high, about 250,000 ms.

Oscillation. This attack is not initially as effective as the inflation attack, however it still does not show up on Fig. 3.8(a) and causes the prediction error to converge to about 250,000 ms.

Delay. We then found a delay attack, where malicious nodes delay responding to Probe messages for 2 seconds. This causes benign nodes to measure a greater RTT than malicious nodes, causing them to think the attackers are farther away than what they should be, thus they update their coordinates to incorrect positions. This attack remains effective over time, continuously causing Vivaldi to have an error near 300 ms.

Attacks found using velocity. We re-discovered the inflation, oscillation, and delay attacks and found one new attack using velocity as impact score. As shown in Fig. 3.8(b), when there is no attack the velocity of the coordinates converge to about 2 ms/s.

Zero Error. We discovered a combination attack where nodes drop their outgoing Probe messages and lie about their local error values as being zero. Even though dropping

Probe messages is an effective deflation attack alone, this combined attack is significantly more damaging than the deflation attack, as it causes velocity to more than double.

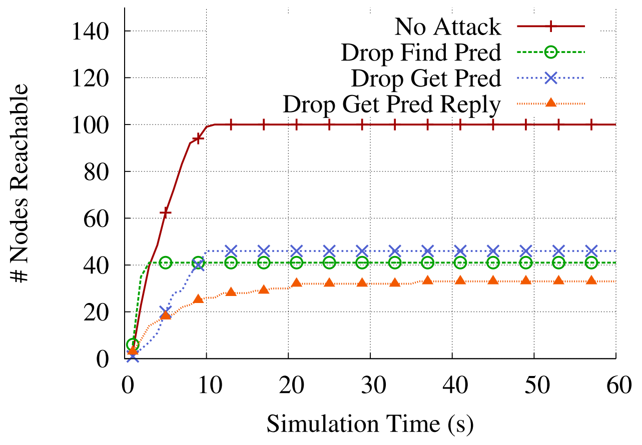
3.6.2 Chord

System description. Chord [16] is an overlay routing protocol that provides an efficient lookup service. Each node has an identifier that is based on consistent hashing, and is responsible for a range of keys that make up that space. Nodes in Chord construct a ring and maintain a set of pointers to adjacent nodes, called predecessors and successors. When a node i wants to join the ring, it will ask a node already in the ring to identify the correct predecessor and successor for i. Node i then contacts these nodes and tells them to update their information. Later, a stabilization procedure will update global information to make sure node i is known by other nodes in the ring.

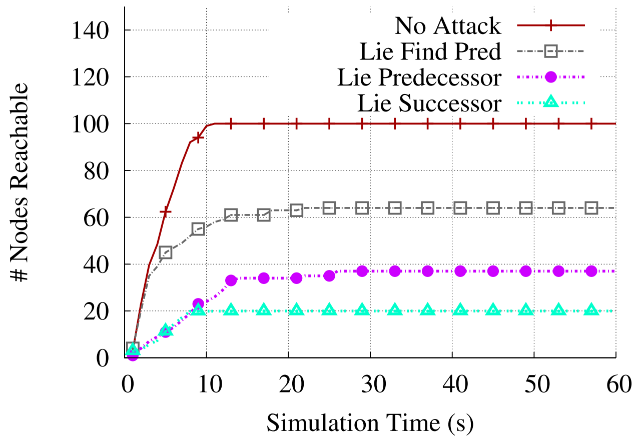
Impact score. We use an impact score which measures the progress of forming a correct ring. Since Chord correctness depends on being able to reach every node by following the successor reference around the ring, we use as the impact score the average number of nodes each node can reach by following each node's successor. Formally, define the relation $succ(i)$ to be the successor of node i. The impact score is the average size of the closure of the relation $succ$ on each node i. For a benign case, the impact score should be equal to the total number of nodes.

Experimental setup. We simulate Chord with 100 nodes. Malicious actions start immediately as the goal of Chord is to construct a properly functioning ring and thus we want to find attacks on that construction process. We set t_w to be 2 sec as ring construction takes only 10 sec in the benign case and set n_a to 5.

Attacks found using number of reachable nodes. We found six attacks against the Chord protocol. In Fig. 3.9 we show the effects of the attacks and illustrate the resulting ring for one attack. As a baseline we verify that when there is no attack all 100 nodes are able to form a ring in less than 10 sec.

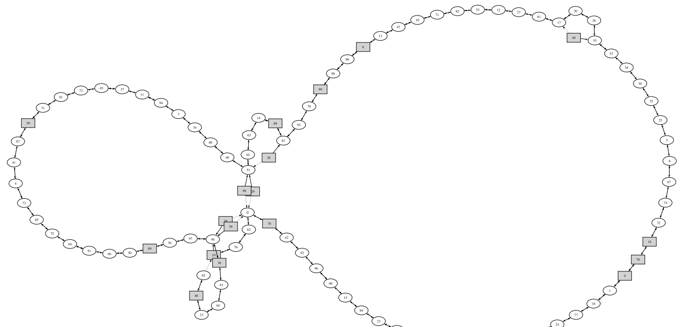


(a) No. of reachable nodes for the attacks found

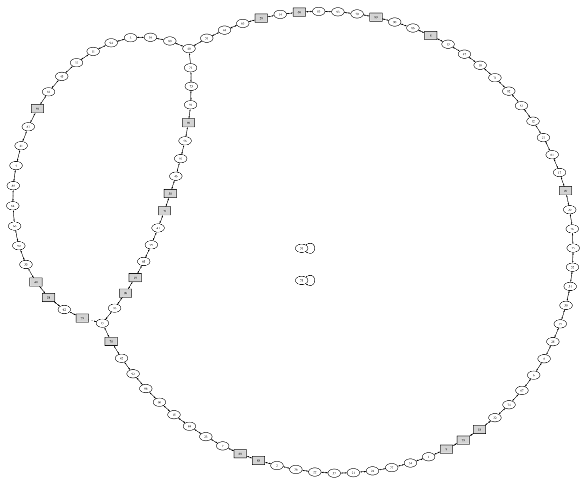


(b) No. of reachable nodes for the attacks found

Figure 3.9.: Attack impact on Chord performance



(a) Chord ring under Drop Get Pred attack



(b) Chord ring under Lie Predecessor attack

Figure 3.10.: Visualization of Chord rings under attack

Dropping attacks. We found three attacks where malicious nodes drop responses or do not forward replies to requests for predecessor and successor information. The attacks prevent a correct ring from forming. We show in Fig. 3.9(a) (*Drop Find Pred*, *Drop Get Pred*, and *Drop Get Pred Reply*) that when malicious nodes drop predecessor related messages, less than half the nodes are reachable.

Lying attacks. We found three lying attacks that prevent a correct ring from forming. The join protocol first locates the predecessor of a given address i by forwarding a FindPred message through the Chord overlay. If a malicious node modifies the address i in the message, it effectively redirects the node joining to an incorrect place in the ring, and can cause inconsistent state in the nodes, which can lead to a failure to properly join (*Lie Find Pred*). We found similar attacks when a malicious node, during stabilization, queried as to who are its predecessors and successors, lies and gives incorrect information (*Lie Predecessor*, *Lie Successor*). We show impact scores of these attacks in Fig. 3.9(b). The effect of *Lie Predecessor* on the ring can be seen visually in Fig. 3.10(b), where some nodes failed to join, and others are confused about their relationships to adjacent nodes.

3.6.3 Distributed Hash Table

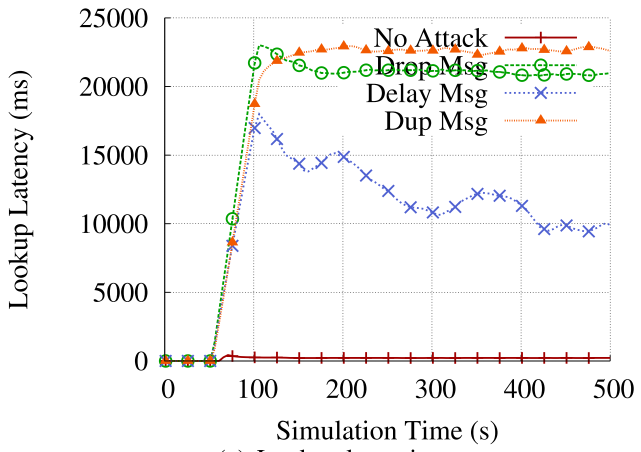
System description. A Distributed Hash Table (DHT) provides a scalable key-value storage service, where nodes self-organize so that each node is responsible for storage of a portion of the key-space. DHTs are often built on top of overlay routing protocols, thus taking advantage of their consistent hashing feature and allowing them to also provide load-balancing. DHTs expose at least two operations to the user, a put operation that stores a value based on its key in the overlay and a get operation that retrieves key-values that are previously stored. The DHT implementation used is a basic one based on the outline in the Chord paper [16], structured as the example described in Figure 3.4. When an application node requests an operation (get or put), the storage layer routes the operation to the responsible node using the recursive routing layer. The recursive overlay routing layer forwards any message to the destination by forwarding it hop-by-hop along the Chord overlay links. The DHT also responds to changes in the responsible address space by sending a SetKeyRange message to the new owner to notify it of the keys it should now manage.

Impact score. For an impact score we use *lookup latency*, which measures the amount of time passed between a node issuing a get request on a key and when it actually receives the corresponding value. Formally, the impact score is the average time spent on lookups that either completed in the last t_w or elapsed time of pending lookups.

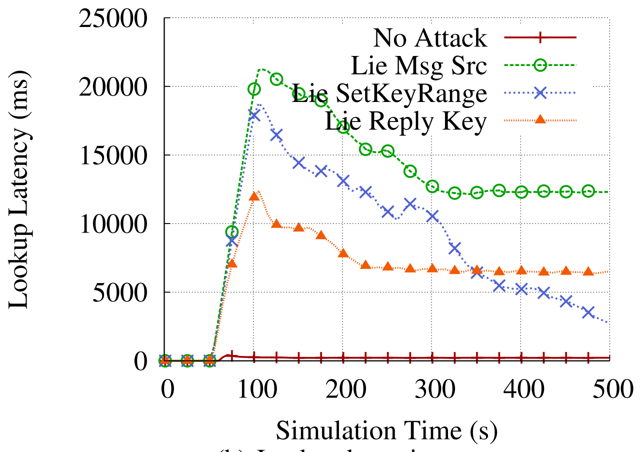
Experimental setup. We simulated 100 nodes and each node randomly generates 100 key-value pairs which it puts around the DHT, thus we expect that each node stores one of its values on every node. Each node then tries to retrieve 2 values every second, and tries to retrieve the whole set of values 10 times. A request is timed-out if no response is received before the next retrieval attempt. Most experiments allow Chord 10 sec to form the overlay before beginning to put data. It then uses 50 sec to put data, putting only 2 values every second, before beginning to lookup data. The remaining lookups take 500 sec. We set t_w to be 70 sec, which allows Gatling to find attacks during the Chord setup and DHT put phase; n_a was set to 5.

Attacks found using lookup latency. We show lookup latency over time for each attack in Fig. 3.11. As a baseline we show DHT with no attack and find it converges to 215 ms. We found a total of seven attacks (and several variants) against DHT and rediscovered some attacks against Chord.

Recursive Overlay Routing Attacks. We first run Gatling on the recursive message routing layer that routes all DHT messages. We begin malicious actions after 10 sec, after the Chord ring converges. We found two attacks where delaying or dropping messages causes an increase in lookup latency (*Drop Msg, Delay Msg*). We also found a third attack where duplicating the message and diverting the second copy to a random node causes network flooding and congestion due to malicious nodes repeatedly replicating the same messages (*Dup Msg*). Finally, we found an attack where in forwarding messages, an attacker provides a false destination key for the message, causing the *next hop* of the message to forward it incorrectly (*Lie Msg Dest*).



(a) Lookup latencies



(b) Lookup latencies

Figure 3.11.: Lookup latency for the attacks found on DHT

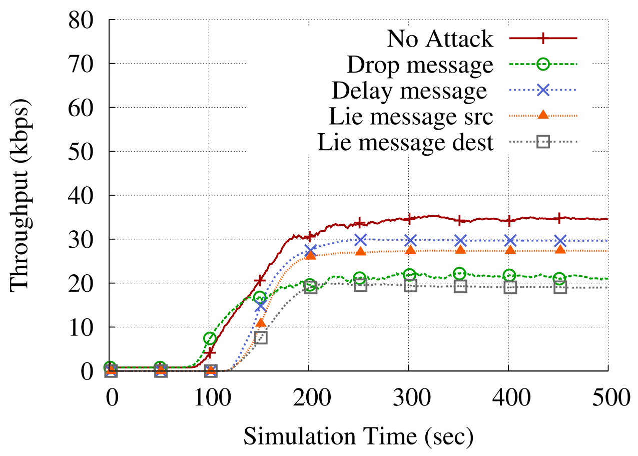


Figure 3.12.: Throughput for the ROR attacks found on DHT

Storage Attacks. We found two lying attacks. The first one, *Lie Reply Key* occurs when a node responds to a DHT get request and it lies about the key it is responding about. The second one, *Lie Set Key Range* occurs during the setup phase of the DHT, considering a scenario where nodes start putting data into the DHT at the beginning of the simulation, before the Chord ring can stabilize. We found that attackers can subvert the process of load-balancing and cause many key-value pairs to go missing. This occurred when an attacker notified another node of what key-value pairs it had. The attacker lied about what keys it was responsible for, then when another node takes over a part of that key-range, he will not know the real values that it should store, thus losing them.

3.6.4 ESM

System description. ESM [47] is a multicast system that efficiently disseminates video streaming data broadcast by a single source. ESM accomplishes this by building up a tree, rooted at the source, where each child node forwards on the data to its own children. Each node maintains a set of neighbors periodically reporting their throughput and latency. With this information, a node can change parents to maintain desired performance.

Impact score. We use two scores [47]: *throughput* and *latency*. Throughput as described in [47], is the amount of data received over the last 5 sec. We use as impact score the streaming rate minus the throughput, to satisfy requirements that larger means more impact. Latency is the amount of time it takes for data to reach each node after being initially sent by the source, and the impact score is the average latency of data in the last t_w.

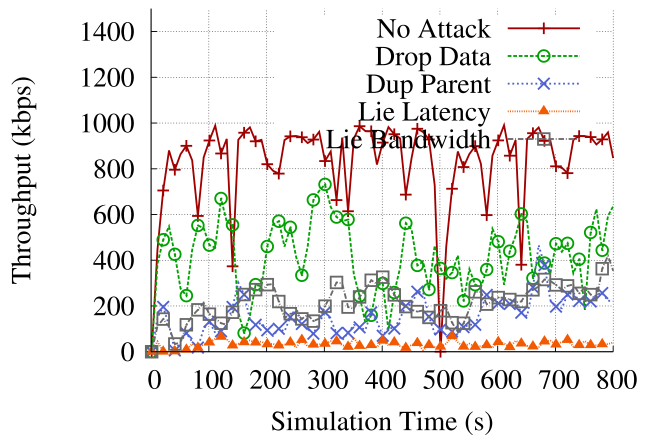
Experimental setup. We simulated ESM with 100 nodes and one source streaming data at 1 Mbps. As the goal of ESM is both to form an efficient tree and to stream data to all participants, we use two different settings for the time (i.e., 0 sec and 10 sec) when attackers start their malicious actions. Thus, we can find attacks both against tree formation and data delivery. We use a t_w of 5 sec and n_a of 5.

Attacks found using throughput. We found four attacks using throughput as an impact score. Fig. 3.13(a) shows the results of how each attack affects ESM where we plot the throughput over time. For a baseline we also have ESM in the benign case when there is no attack, delivering average throughput near 900 kbps.

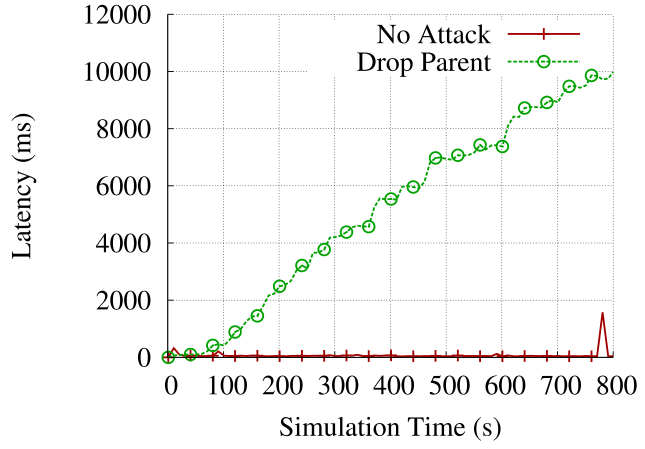
Drop Data. We delay malicious actions until 10 sec into the execution, to allow ESM to build a tree first, and test the steady state. Despite ESM using an adaptation mechanism to switch to parents that give them good performance, dropping data was an effective attack.

Dup Parent. We then examined attacks that targeted the tree formation and adaptation process. We increased the window size t_w to 10 sec, and had attackers immediately start trying malicious actions once the simulation started. Gatling again added dropping data as an attack action, then proceeded to amplify that attack with another malicious action—duplicating messages that tell a node that it is accepted as a child, sending the duplicate message to a random node. With this amplification, the throughput drops to below 200 kbps.

Attraction attacks. In these attacks malicious nodes amplify dropping streaming data by lying about their performance metrics, making them look better than what they actually are. This causes benign nodes to continually ask malicious nodes to be their parents. The first attraction attack found is where nodes lie about their latency (*Lie Latency*), setting it to zero. This causes nodes to think the attacker is close to the source. The second attraction attack is when malicious nodes lie about their bandwidth using scaling (*Lie Bandwidth*), increasing it to appear they are receiving much of the streaming data. To further amplify the



(a) Throughput impact score



(b) Latency impact score

Figure 3.13.: Attack impact on ESM

attack the attackers also duplicate probe messages, diverting the second message to random nodes, causing the attackers to be more well-known in the overlay, thus more likely to be asked to be a parent. These two attacks are very effective, causing all nodes to have a very low throughput of less than 100 kbps when lying about latency and 300 kbps when lying about bandwidth.

Attacks found using latency. We found one attack using latency as an impact score function. We compare in Fig. 3.13(b) the latency when there is no attack with the attack we found.

Drop Parent. We found an attack where malicious nodes drop replies to parent request messages. This results in increased latency due to malicious nodes gaining spots high up in the tree over time by simply following the adaptation protocol, and then never allowing the tree to grow beyond them. Furthermore, as benign nodes never get a response, the protocol dictates that they wait 1 sec to send another parent request to a different node, further slowing down their attempt to find or change parents.

3.6.5 Scribe

System description. Scribe [48] is an application-level multicast system that organizes each group into an overlay tree to efficiently disseminate data. To send data, a node sends the message toward the root, and each node forwards it to its parent and children. Scribe is built on top of Pastry [11], an overlay routing protocol with similar functionality to Chord. Scribe trees are built based on reverse-path forwarding combined with load balancing: the multicast tree is the reverse of the routes Join messages take when routed from tree participants to the Pastry node managing the group identifier. For load balancing, a node whose out-degree is too high will push its children down in the tree.

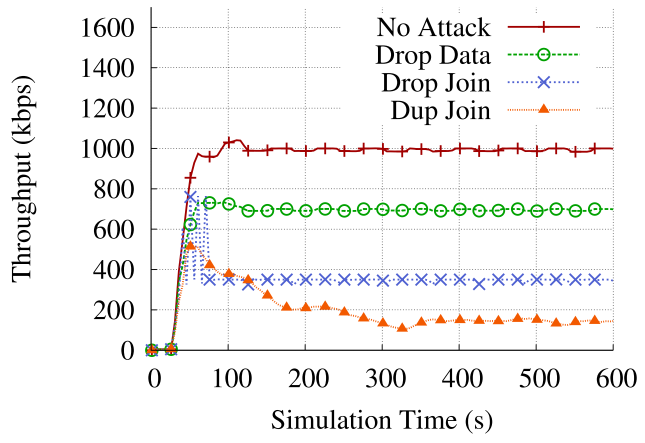
Impact score. We use *throughput*, which measures the average amount of data received over time. As with ESM, the impact score is the streaming rate minus the average throughput over the last t_w seconds.

Experimental setup. We simulated Scribe with 50 nodes and test it under the scenario where a source node creates a group, publishes streaming data at a rate of 1 Mbps, and all other nodes subscribe to that group. We start malicious actions immediately after the experiment starts so we can attack tree construction, however as we find the tree takes up to 30 sec to form in our test environment, we set t_w to be 35 sec. We also find malicious actions have a high probability of being effective the first time tried and thus set n_a to be 1.

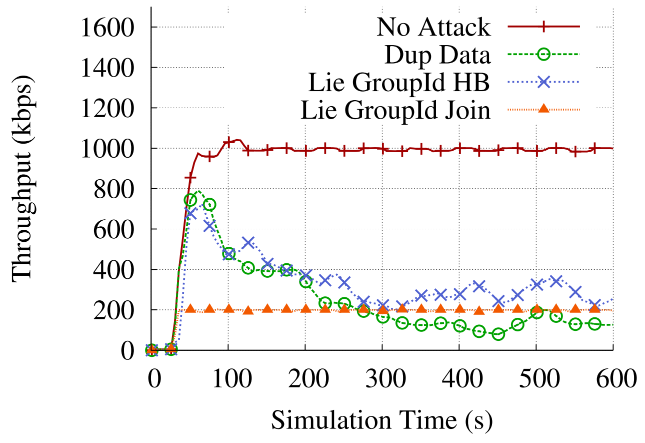
Attacks found using throughput. We found seven attacks using throughput as an impact score. Fig. 3.14(a) shows the effects of the different attacks. As a baseline we run the system with no attack and find that nodes are able to consistently receive 1 Mbps of data.

Drop Data and Dup Data. First, we found two obvious attacks where nodes do not forward the data or they duplicate data messages and send the second message to a random node. In the latter case, loops can occur, causing significant system load as data is increasingly replicated, resulting in throughput to decrease below 200 kbps.

Dup Join, Lie GroupID Join, Drop Join. We found that when malicious nodes duplicate Join messages and divert the second message to a random node this causes the throughput to drop below 200 kbps. This drop is due to temporary forwarding loops when a tree node is a child of multiple parent nodes. This temporary error will be corrected by a heartbeat protocol, but only after a period of time in which forwarding loops can cause damage. Gatling found two additional attacks that cause the tree to not be formed properly. If malicious nodes lie about the group identifier in the Join message, then effectively the malicious nodes are joining a different group, while believing they are joining the requested group. Malicious nodes still respond normally to other nodes' requests to join the correct group. This lie led the system to a situation that all malicious nodes fail to join, and some benign nodes build a tree under malicious nodes as seen in Fig. 3.15(b). Since the tree is split, only nodes in the tree that have the source node inside can receive data and nodes in other tree(s) can not receive any data. Gatling also finds an attack of dropping Join messages, causing the same effect, but in the explored simulation, more benign nodes happened to be a part of the tree with the source, allowing better throughput.

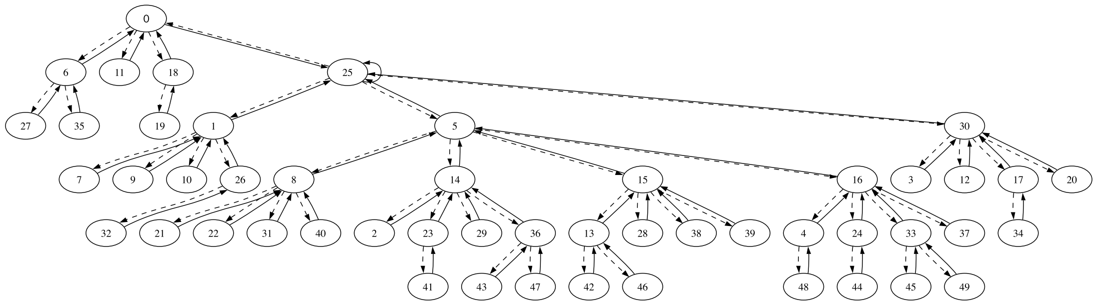


(a)



(b)

Figure 3.14.: Throughput for the attacks found on Scribe


(a) Scribe tree without attack

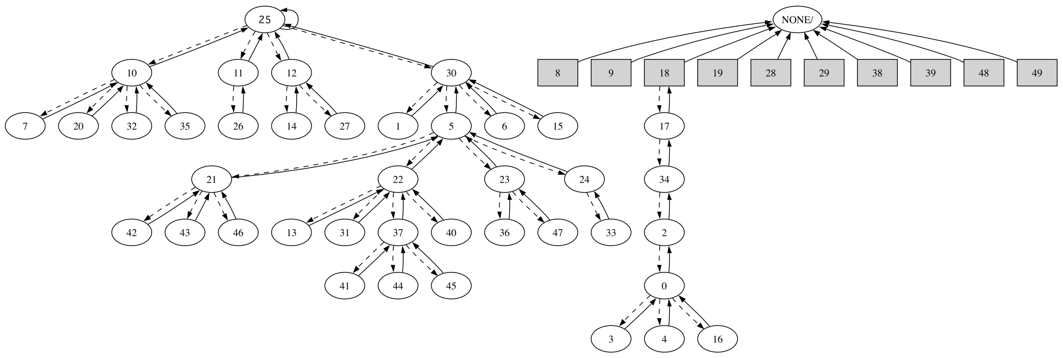

(b) Scribe tree under Lie GroupId Join attack

Figure 3.15.: Attack impact on Scribe

Lie GroupID HB. Gatling also found an attack where malicious nodes change the group identifier in HeartBeat messages. Normally, Heartbeat messages inform the recipients that the sender is the parent of the receiver for the indicated group identifier. Heartbeat messages serve both as the acknowledgement of a successful join, and a periodic liveness test. As malicious nodes send an incorrect group identifier, the Heartbeat message is discarded by the child, which will attempt to re-join periodically. As with the previous attack, the resulting effect is temporary forwarding loops, while causing throughput to decrease to below 400 kbps.

Forwarding loops are simple enough to defend against by preventing the application from forwarding duplicate messages; however, it is worth noting with the exception of the data duplication attack, these attacks actually caused undesirable temporary loops in the underlying tree data structures.

Many distributed systems only tolerate fail-stop failures or very limited malicious behaviors, mainly because it is a very challenging task to build an efficient and robust distributed system to protect against such faults. However, some distributed systems are specifically designed and implemented to be secure and robust against insider attacks. In this section we apply Gatling to such secure distributed systems. Table 3.3 summarizes the attacks we found on secure systems.

3.6.6 Secure virtual coordinate systems - Outlier detection

Vulnerabilities of Vivaldi have been studied and a number of defense mechanisms have been proposed. We can classify defense mechanisms into two categories; landmark based and distributed. Landmark based techniques require a set of trusted nodes that will never be compromised while distributed techniques do not have such a requirement. Because having a set of trusted nodes often can be difficult to achieve in practice, distributed techniques are widely used.

System description of Outlier Detection. Zage et al. [49] developed a defense technique that uses Outlier Detection to protect Vivaldi. Data points that deviate by a large

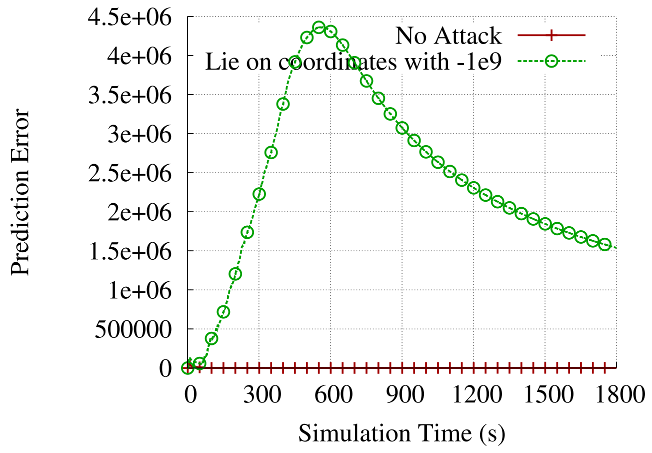
Table 3.3: Attacks found using Gatling on secure systems: 7 attacks in total (1 progressive delay, 3 lie, 1 delay, 2 duplicate)

System	Metric Used	Attack Name	Attack Description	Known Attack
Outlier Detection	Prediction Error	Frog-boiling Delay	Delay probe reply, starting with a small amount and increase the amount of delay over time	[61]
		Frog-boiling Lie	Lie about coordinates starting with a small scaling and increase the amount of scaling over time	[61]
		Large Lie	Lie about coordinates with a very large number when $timestep$ is constant	
		Random Lie	Lie about coordinates with a completely random values	
BFT	Throughput	Delay PrePrepare	A primary delays PrePrepare	[62]
		Dup Prepare	Duplicate Prepare message	
		Dup Commit	Duplicate Commit message	

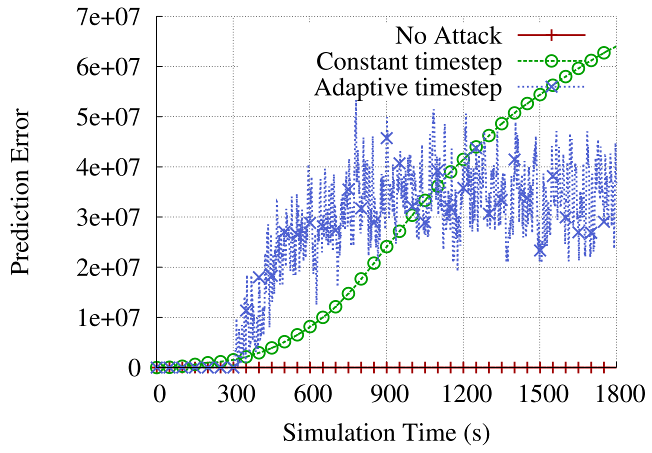
amount from rest of the data set are called outliers or anomalies. Outlier detection has been used in many contexts to distinguish abnormal data points from the rest. If some nodes report data that is very different from other nodes are reporting, it is likely that such nodes are malicious nodes lying to confuse the system, therefore we can project the system by discarding their reports. The authors use outlier detection to detect malicious nodes by examining reported data points in terms of temporal, spatial, centroid and distance. Specifically, all nodes first learn good behavior then discard outliers.

Impact score. As for Vivaldi, we use the *prediction error* [46] which describes how well the system predicts the actual RTT between nodes. Prediction error is defined as *median*$(|RTT_{Est}^{i,j} - RTT_{Act}^{i,j}|)$, where $RTT_{Est}^{i,j}$ is node i's estimated RTT for node j given by the resulting coordinates and $RTT_{Act}^{i,j}$ is the most recently measured RTT.

Experimental setup. To allow nodes to learn good behaviors, we start malicious actions after 60 seconds from the experiment start time for both finding attacks and verifying



(a) Lie about coordinates with very large numbers



(b) Lie about coordinates with random numbers

Figure 3.16.: Attack impact on Vivaldi Outlier Detection

attacks. As we used in the Vivaldi setup, we simulated 400 nodes and randomly assign RTT values for each node from the KING data set [60] which contains pair-wise RTT measurements of 1740 nodes on the Internet. Malicious nodes start their attacks from the beginning of the simulation. We set t_w to be 5 sec and n_a to be 5 and increased t_w later to give an opportunity to find stealth attacks.

Attacks found using prediction error. Fig. 3.16(a) shows an attack that is effective when the $timestep$ is constant. $timestep$ is a parameter defined in Vivaldi to adjust speed of convergence and oscillation. A large $timestep$ can help fast convergence yet it can

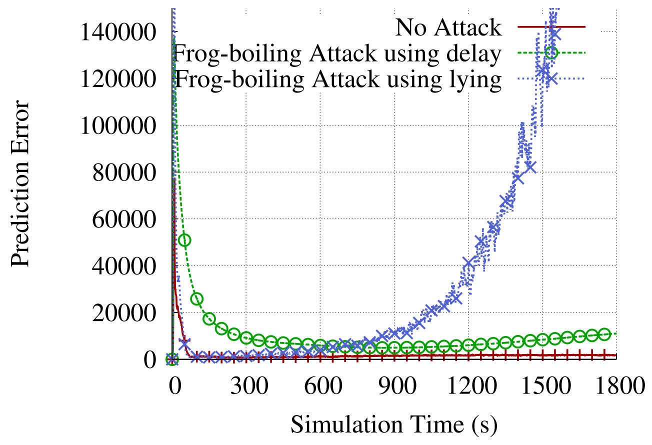


Figure 3.17.: Frog-boiling attacks on Vivaldi Outlier Detection

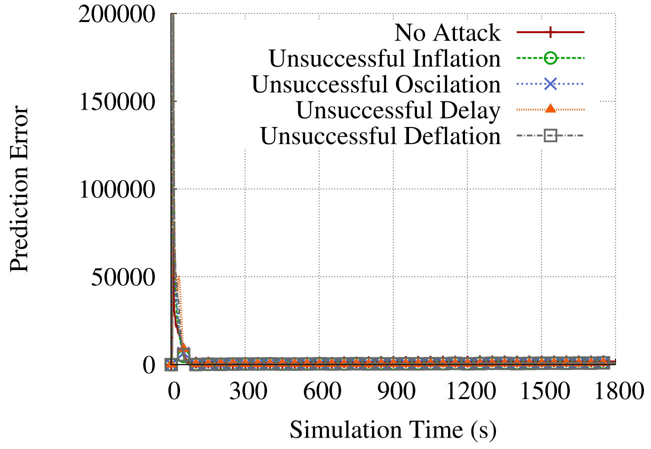


Figure 3.18.: Outlier Detection protects Vivaldi from other attacks

prevent Vivaldi from finding low-error coordinates and oscillate in large steps. A small $timestep$ will cause the opposite. Vivaldi supports both constant and adaptive $timestep$ and constant $timestep$ provides fair result with a reasonable choice of $timestep$. However, in the presence of attackers, especially when they lie about coordinates with large numbers such as $1e{+}09$, the constant $timestep$ approach does not work well. A larger non-changing $timestep$ causes the springs to oscillate between coordinates and never actually stabilize.

Fig. 3.16(b) shows an attack that is effective under both constant and adaptive $timestep$. Sometimes some malicious coordinates fall into non-outlier range, when this happens, it distorts the centroid. This causes all benign coordinates have a longer distance from the centroid than normal. Thus, in the next update, the attacker gains a better chance to avoid Outlier Detection. Once a large error coordinate is accepted, Outlier Detection can not filter malicious updates very well. Based on our observation, some fraction of nodes have a larger error due to their location, making the average prediction error very high. As seen in Fig. 3.16(b), adaptive $timestep$ does not suppress the attack impact enough.

Fig. 3.17 shows a stealth attack that Gatling found. In this attack, Gatling injects progressive actions. This particular attack starts with a very subtle disruption and becomes aggressive over time. This attack is called a frog-boiling attack. It was previously found in [61]. Gatling found this attack with the repeating parameter option, with a very large t_w, 400 seconds. In delaying, the prediction error becomes longer than 10,000 ms and increases even more over time. In lying, the prediction error increases exponentially. Without the repeating parameter option Gatling cannot find this attack even with a very large window. Because the beginning of the attack is very subtle, i.e. the malicious node lies with a small amount undetectable for the outlier detection mechanism, injecting a single action does not cause any observable impact.

3.6.7 Secure virtual coordinate systems - Newton

Recently, Seibert et al. [50] developed a new secure virtual coordinate system, Newton. Newton is another type of virtual coordinate system that is designed to protect the system from insider attackers without requiring some trusted points.

System description of Newton. Instead of using statistical methods, Newton uses safety invariants derived from physical laws of motion. The main idea is based on the observation that Vivaldi is an abstraction of a real-life physical system. Therefore, participating nodes should follow physical laws of motion. Therefore, the authors define invariants based on Newton's three laws of motion: 1) a body should remain at rest unless acted upon by an external, unbalanced force, 2) a force F, on a body of mass m with an acceleration a, is the vector sum of m and a, 3) when a first body exerts a force on a second body, the second body exerts an equal amount of opposite force on the first body. The authors apply these three laws and identify three invariants. If a node violates these invariants, Newton identifies the update malicious and discards them to protect the system.

Result. Gatling could not find any effective attacks. Newton is resilient to attacks that we found including the frog-boiling attack and no effective attack has been found. The result of applying Gatling on Newton shows two limitations of Gatling. First, Gatling does not prove absence of attacks. What we can learn from our result is that Newton defeats the types of attacks that we injected and it is still unknown if Newton is completely vulnerability-free.

Second, to attack a system like Newton that is resilient to simpler message manipulation, malicious nodes need to create attacks that appear legitimate to the defense mechanism, i.e. attacks that do not invalidate through their results Newton's laws of motion. Such a requirement is too complicated and too specific and it is not supported by Gatling.

3.6.8 Byzantine Fault Tolerant System

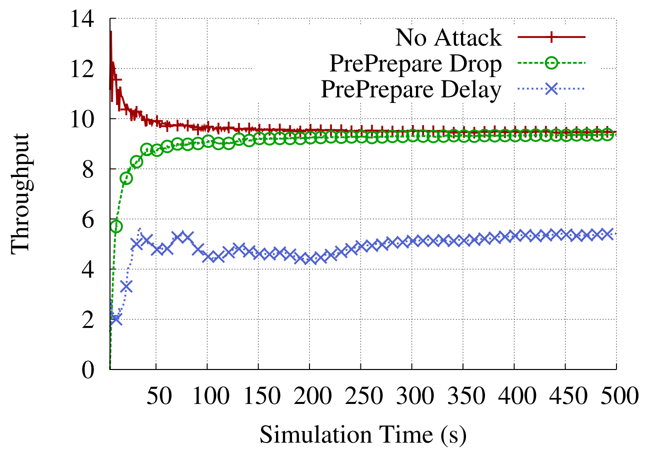
To defend distributed systems from arbitrary byzantine attacks, many state-machine replication protocols have been proposed. Such protocols are designed to provide a correct service to correct participants, even when a fraction of the components are compromised and act arbitrarily, as long as certain assumptions are met. While there are several variants of systems in this category, we use PBFT [2] as our target system because it is the first practical byzantine resilient system and it is representative for this class of systems because of its design.

System description. PBFT is a state-machine replication algorithm that provides correct service to correct participants when there are at least $3f+1$ total nodes while at most f nodes are faulty. Faulty participants can behave arbitrarily. The algorithm offers both safety and liveness properties where the safety property ensures that the system will not have inconsistency and the liveness property means that the system will make progress eventually. Specifically, PBFT requires acknowledgments from $2f+1$ out of $3f+1$ servers to tolerate the incorrect behavior of f Byzantine servers. A client must wait for $f+1$ identical responses to be guaranteed that at least one correct server assented to the returned value. All replicas in PBFT participate in several protocols. We implement the main protocols described in the main paper [2] without optimization protocols that are extended after the publication.

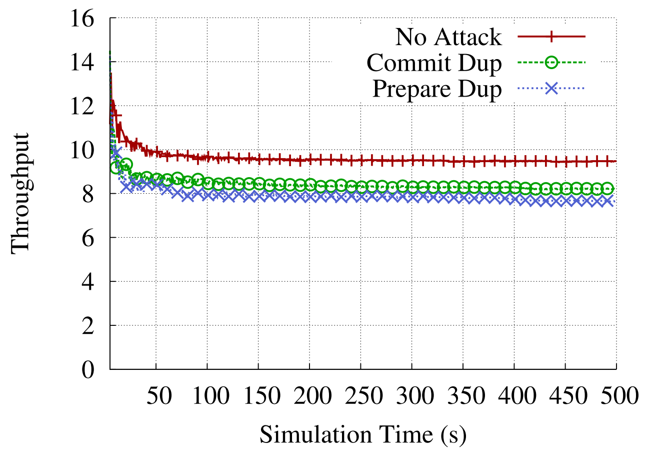
Impact score. We use the *throughput* to describe how many requests were fulfilled in a second. Because throughput is a metric the bigger number represents the better performance, we multiply with -1.

Experimental setup. We simulated one client, one Byzantine server and three benign servers, a typical setup of this system. The client is benign and it does not pipeline requests as our focus is not to stress the system with benign workloads. Unlike other systems, the primary and a replica have different functionality and thus malicious behavior of each of them has a different impact on the system. We ran experiments first with a malicious primary and then with a malicious replica.

Attacks found using throughput. Fig 3.19(a) shows the performance impact when the primary was malicious. Delaying PrePrepare degrades the throughput from 10/sec to 5.5/sec. While PBFT suffers from this delaying attack, if the malicious node just drops the message, the system can quickly cause a view change and thus it recovers the performance



(a) Performance impact when the primary is malicious



(b) Performance impact when the replica is malicious

Figure 3.19.: Attack impact on PBFT

quickly. Gatling initially found dropping PrePrepare, however this attack was discarded in the validation process as the system recovered its performance.

When the malicious node is a replica, it does not have a chance to send PrePrepare, thus we did not find this attack. However, we found that duplicating Commit and Prepare messages are effective attacks conducted by a malicious replica.

3.7 Summary

Securing distributed systems against performance attacks has previously been a manual process of finding attacks and then patching or redesigning the system. In a first step towards automating this process, we present Gatling, a simulation based framework for automatically discovering performance attacks in distributed systems. Gatling uses a model-checking exploration approach on malicious actions to find behaviors that result in degraded performance. We provide a concrete implementation of Gatling for the Mace toolkit. Once the system is implemented in Mace the user needs to specify an impact score in a simulation driver that allows the system to run in the simulator.

To show the generality and effectiveness of Gatling, we have applied it to nine distributed systems that have a diverse set of system goals. We were able to discover 48 attacks in total, and for each system we were able to automatically discover attacks that either stopped the system from achieving its goals or slowed down progress significantly. While some of the attacks have been previously found manually through the cleverness of developers and researchers, we show that the amount of time Gatling needs to find such attacks is small. Therefore, we conclude that Gatling can help speed up the process of developing secure distributed systems by finding attacks automatically from the implementations of distributed systems.

4 TURRET: FINDING ATTACKS IN AN EMULATED ENVIRONMENT

In the previous chapter, we presented Gatling, a simulated method to find attacks in distributed system implementations. While Gatling allows us to find many design or implementation level attacks from the implementation directly, it has important limitations. Because Gatling requires the implementation to run on a simulator, unless the target systems is written in the language that is supported by the simulator, the application needs to be re-written to run with the simulator. This requirement limits target applications significantly. In this chapter, we investigate how to test applications that are written already in their language of choice without requiring modification.

4.1 Introduction

Finding performance attacks in distributed system implementations has been done mostly manually by an expert knowing in depth the system and the environment. Even for experts, finding attacks is a painstaking task due to the complexity of the implementations and the fact that vulnerabilities often lie in corner cases. While manual testing is useful to inspect deep corner cases that developers are already aware of, given the wide range of vulnerabilities that can occur in a distributed system, manually writing testing scenarios to discover all possible vulnerabilities is not cost effective. Thus, there is a need for *automated generation of meaningful scenarios and testing* based on minimal input from the system developer.

Automated testing of implementations of distributed systems has focused on bug finding. Model checking using a systematic state-space exploration [32, 35–38] has been used on system implementations to find bugs that violate specified invariants. This approach helps to check implementations under benign failures. However, many of them do not consider distributed systems [32, 35, 36], require the target implementation to be written in a specific language for a specific programming model [37], or require a particular target operating system and the implementation to be instrumented [35].

Automatically finding performance attacks in implementations of distributed systems has received very little attention in the past. Finding attacks against performance is a challenging task due to both the difficulty of expressing performance as an invariant of the system and the state-space explosion that occurs as attackers are more realistically modeled. There are only three works that we are aware of that focus on automatically finding performance attacks in network protocols or distributed system implementations [3, 4, 63]. The work in [3] finds attacks by maliciously modifying the headers of the packets of a two-party protocol and exploring all the states, but it does not scale for a distributed system. MAX [4] also focuses on a two-party protocol, but it scales better because it limits the search space by asking the user to supply information about the vulnerability of the system. This approach is fitted for corner cases but does not attempt to find attacks systematically. Finally, our previous work, Gatling [63] does not require a priori knowledge about specific vulnerabilities in the system. However, it relies on the Mace [5] framework which requires the tested systems to be written in the Mace language. To the best of our knowledge, there is no work that focuses on automatically testing unmodified distributed system implementations without modifying the implementations or limiting the language of implementations, used libraries or operating systems.

In this chapter, we focus on finding performance attacks in distributed system implementations automatically, without requiring the user to provide information about vulnerabilities of the system, and without modifications of the implementation or the operating system. We focus on performance attacks since today no distributed system can be expected to be practical without maintaining some level of performance in stable networks. Our tool is intended as a preventative, pre-deployment attack finding tool and complementary to approaches that test for particular corner cases [64]. We do not restrict libraries or operating systems because we want our solution to be applicable to already existing implementations in various languages, for various operating systems and to reproduce deployment conditions. The attacks we consider are performed by compromised participants that manipulate protocol semantics by interfering with the content and delivery of the messages. The contributions are as follows:

- We present the design of Turret, a platform for performance attack discovery in unmodified distributed system implementations running in a user-specified operating system images subject to controlled, realistic network conditions. Turret leverages virtualization to run arbitrary operating systems and applications, and uses network emulation to connect these virtualized hosts in a realistic network. Turret requires only a description of the external API of the service, *i.e.* the message protocol, and the ability to observe the application-performance of the system. Turret uses a new attack finding algorithm, *weighted greedy*, that improves over previous solutions [63,65] by reducing the time required to find an attack to minutes. Low time overhead is critical for platforms that run systems in realistic conditions, like ours, where the systems run in real time and not simulated time.

- We present an implementation of Turret that uses the KVM virtualization infrastructure and the NS3 network emulator. To support execution branching of the entire distributed system, we design and implement checkpointing and restoring the state of its execution. We leverage snapshot functionality of KVM to save, pause, load, and resume virtual machines (VMs). In addition, we create a page sharing aware snapshot management functionality that leverages the shared pages between the different VMs to reduce the total footprint of snapshots and thus reduce the time taken to save them. Our optimization reduces the time to take snapshots of VMs between 34.5% to 40.3% for five to 15 virtual machines, respectively. We also added save, load, freeze, and resume functionality to NS3 to capture the network status, including packets in the network.

- We show the effectiveness of Turret by applying it to 5 systems written in C and C++ (PBFT [2], Steward [66], Zyzzyva [67], Prime [68], and Aardvark [69]) and found 30 performance attacks, 24 of which were not previously reported to the best of our knowledge. We also describe how we used Turret in a graduate distributed systems class.

The rest of this chapter is organized as follows. We present the system and attack model in Section 4.2 and our design in Section 4.3. We describe implementation issues in Section 4.4 and the results we obtained running Turret on five systems in Section 4.6. Finally, we present our conclusion in Section 4.7.

4.2 Turret system and attack model

We focus on performance attacks in distributed systems caused by compromised participants. Below, we describe the system and attack models we consider in this work.

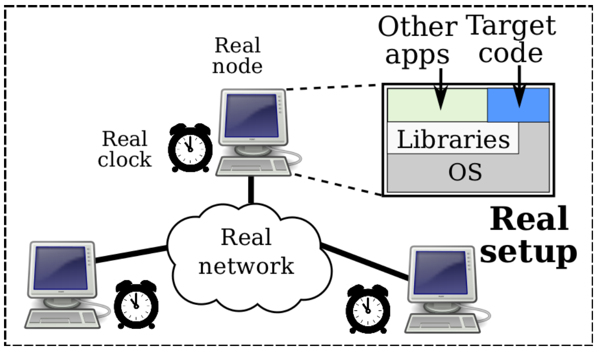
4.2.1 System model

We focus on distributed systems in *message-event model* and assume the participants communicate via messages through TCP or UDP communication. As we consider that the distributed system implements its protocol at the application level; thus, a message might be contained in several network packets. We consider a network event as an event to deliver a message, not an event to deliver a packet if a message is contained in several packets.

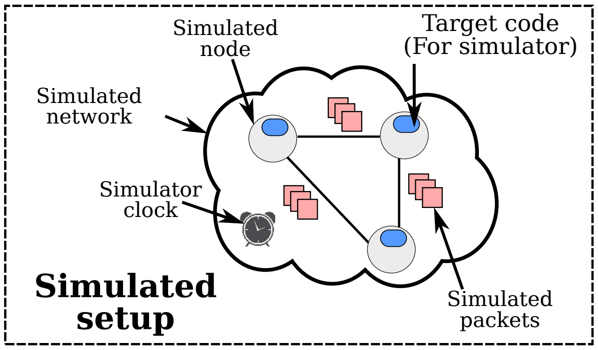
We focus on systems that have measurable performance metrics, such as throughput and latency. We assume that there is a way to collect information about the performance from outside of the application, and most of the distributed systems have ways to export their performance through logs or some monitoring tools.

4.2.2 Attack model

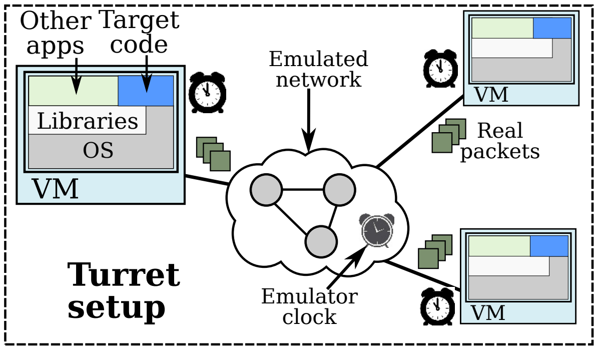
We focus on attacks against system performance where compromised participants running malicious implementations try to degrade the overall performance of the system through actions on exchanged messages. Such attacks require that the system has a per-



(a) Real network of computers running the target distributed system



(b) Simulated network of nodes running the target distributed system



(c) Emulated network of virtual machines running the target distributed system

Figure 4.1.: Approaches for finding performance attacks in distributed systems implementations.

formance metric that gives an indication of the progress the system has in completing its goals.

We focus on attacks specific to message-passing distributed systems, but general enough for any of such systems. The attacker does not know the internal of the system, but it knows the type and the format of the messages and uses this information to have a global impact on the system performance. We classify all malicious actions on messages into two categories: *message delivery actions* and *message lying actions*.

Message delivery actions. Message delivery actions are the most general type of malicious actions and performing them does not require knowledge of the messaging protocol, because the actions are being applied to where and when the message is delivered rather than modifying the message contents. We define the following types of malicious message delivery actions:

- *Dropping*: A malicious node drops a message instead of sending it to its intended destination.

- *Delaying*: A malicious node does not immediately send a message and injects a delay.

- *Diverting*: A malicious node does not send a message to the destination intended by the protocol and enqueues the message for delivery to a node other than the original destination.

- *Duplicating*: A malicious node sends a message multiple copies instead of sending only one copy.

Message lying actions. Message lying actions are conducted by a stronger attacker that knows the type and the format of the messages. However, we assume that the attacker does not know the semantics of the messages, just the types of different fields, so the attacks are still general enough. The attacks do not randomly modify values; instead they follow the field types, and they use values meaningful for that type.

We define message lying actions as actions where malicious participants modify the content of the message they are sending to another participant. An effective lying action involves intelligently modifying fields of the messages likely to cause different behaviors, which is more sophisticated than random bit-flipping. As the number of possible values that the message field could contain may be extremely large, we define a few general strategies for field types that stress the system in different ways based on previously found attack practices and the type of the message field. An attacker can lie about a field based on absolute and relative values. For absolute value based lying, we assume min, max, random and spanning where spanning means values from a set that spans the range of the data type. For relative value based lying, we assume addition, subtraction and multiplication of the original value. We assume that the attacks support boolean, the signed/unsigned integers of size 8, 16, 32, 64 bits, the float and the double types.

4.3 Turret design

We provide the overview of the design of Turret. We then give more details about three system components: attack finding algorithm, execution branching, and malicious actions.

4.3.1 Design overview

Fig. 4.1(a) depicts an ideal environment to find attacks in a distributed system. In this environment, the unmodified code runs in a software context that is exactly the same as the deployment environment and on the actual (or replica of the) production physical hardware. Unfortunately, when testing for performance attacks, this approach is infeasible because: (1) testing for attacks in real-time on the real application without interfering with the user experience is not possible; (2) maintaining an exact replica of the system and network deployment is costly; and (3) ensuring reproducible network conditions when using a shared network infrastructure is challenging, and the unpredictable performance and the variability of the traffic can lead to false alarms.

One alternative to overcome these challenges is to use a simulated approach as the one in Fig. 4.1(b), where the entire system is run in an event-based simulator. In this approach, it is relatively trivial to inject malicious message delivery actions by modifying the event behavior in the simulator. Message lying attacks are also easy to implement if the language of the simulator supports structured message serialization by modifying the compiler or the serializer to alter messages. Testing timing related operations such as delays and timeouts are also efficient because timing is simulated, and experiments can run faster than the real time. We used such an approach in our previous work, Gatling [63].

While a simulated approach provides a first step towards automatically finding performance attacks in distributed systems, it requires the system to be implemented in the language supported by the simulator. In addition, the messages do not go through the network stack of the operating system. Thus, such an approach will not capture the intricacies of the interactions between the operating system and the libraries that the distributed system requires in a real deployment. Moreover, many distributed systems are already implemented in various languages and often depend on some specific environments, such as a certain operating system or libraries. Solutions that limit the implementation language or the environment are less general and require the user to modify their applications or the environment which could be infeasible or very expensive.

Our Approach. To address the challenges of testing systems in the wild and the limitations of simulated approaches, we propose to use an emulated approach (Fig. 4.1(c)). In an emulated approach, a distributed system code is run in the same operating systems and libraries as in the deployment, generating real network events with real network packets, and running in real time. The network communication is emulated such that the network conditions are reproducible and controllable. Virtualization enables a realistic environment for the tested system while network emulation enables controllable network communication. Moreover, network emulation can accurately simulate the behavior of deployment networks and network technologies and can interface with VMs running real applications on real operating systems.

While an emulated approach captures the characteristics of real deployments, it also brings new challenges. One of the main challenges is how to design an attack finding algorithm that has little time overhead, as the time in which the system is running is real time and not simulated time. Efficient attack finding requires controlling the execution of the entire environment consisting of several VMs running the distributed systems and a network emulator. This represents a challenge as there is no global state/queue of events that can be easily accessed and modified. Finally, another challenge is how to create malicious message actions automatically in the system without modifying the application code or requiring the user to implement a malicious version of the software, as there is no support for message serialization.

4.3.2 Attack finding algorithm

Consider the point where an attack is injected, we refer to it as an *attack injection point*. Because we consider only attacks conducted through messages, an attack injection point occurs when sending a message. Performance attacks manifest through observable degradation of performance in an execution with attacks compared to an execution without attacks. Thus, one way to determine if a malicious action results in an attack is to measure performance of the execution with that malicious action and compare it with the performance of execution without attacks. Assuming that the difference will be observable within a window of time w after attack injection, it is sufficient to measure those performances from the attack injection point till w elapsed. We refer to the performance with no attack as *baseline* and to the performance corresponding to the malicious action as *performance for malicious action*.

The simplest approach is a *brute-force search algorithm* as the one showed in Fig.4.2 (a). The algorithm relies on a list of all possible attack scenarios (malicious actions for

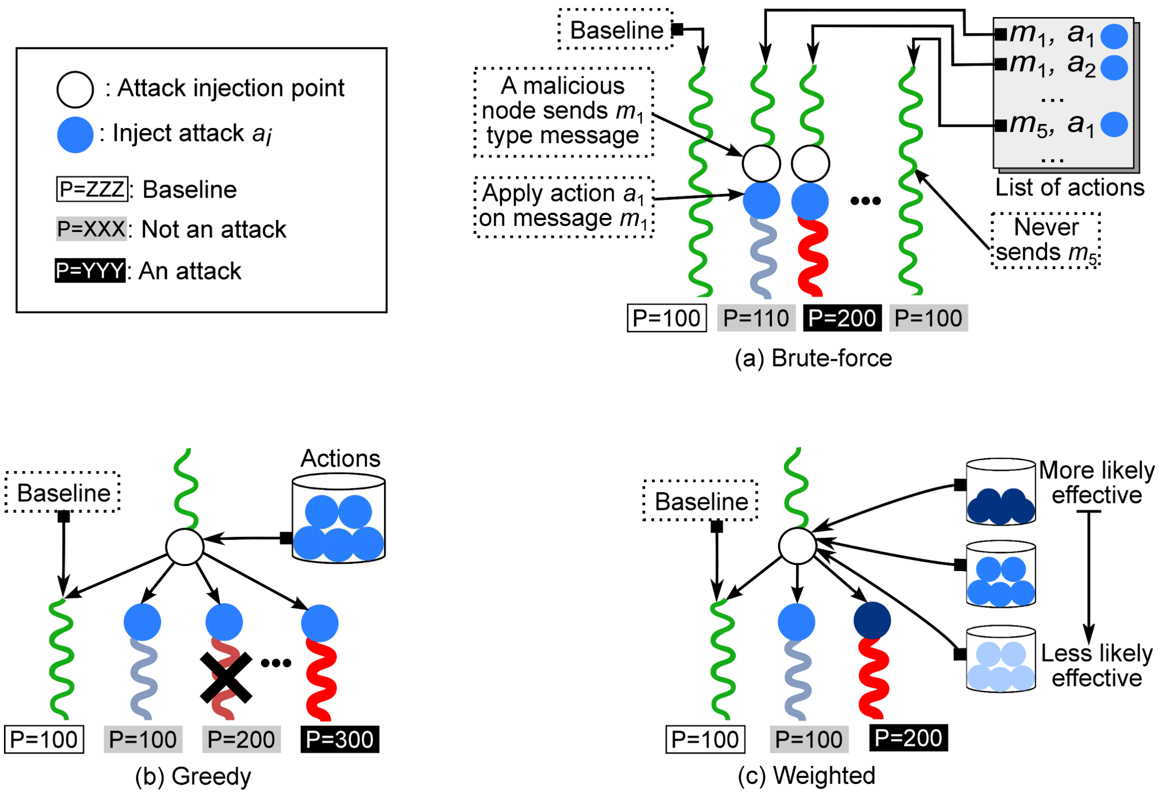


Figure 4.2.: Attack finding algorithms

each message type). [1] It first obtains a baseline, then for each attack scenario in the list, and it runs the system until it encounters an attack injection point, i.e. the sending event of a message with the type from the attack scenario, and obtains the performance for each one. An action is determined to be an attack if the difference between the baseline and the performance for that action was higher than the threshold Δ. The approach has the advantage that it is simple and does not require control during the execution of an attack scenario, but just the ability to start the system and stop it.

However, the approach wastes time due to some unwanted executions: (1) in case no attack injection point is encountered, i.e. no message of the type listed in the attack was actually sent, this approach still runs the entire execution, and (2) in case an attack injection point is encountered, for every malicious action scenario, this approach runs the execution from the beginning even if the performance is needed to be measured only from the attack injection point, not from the beginning of the execution. As the system is run in real time, this can represent a significant overhead.

One approach to avoiding wasted execution time is to use a dynamic *greedy search algorithm*, similar to the one we previously used in a simulated approach [63] and showed in Fig.4.2 (b). When the algorithm encounters an attack injection point, it branches the execution and obtains a baseline as well as the performance for each malicious action for that message type. It then selects the action that caused the largest performance degradation. As an aggressive approach can also make mistakes, higher confidence is obtained by deciding that a scenario is an attack if it was selected more than a certain number of times, which in turn requires additional executions. This approach requires the ability to compare the performance for non-malicious and malicious executions that are branched from the same attack injection point. This requirement was easy to support in a simulator where there is a global state of events, and the entire history of the execution is available for the entire duration of the testing experiment. Supporting execution branching in an emulated approach has additional overhead as either log-replay or checkpointing functionality needs

[1]The malicious actions that can be applied to each messages type depends on the message format, i.e. number of fields and types for each field.

to be added, overhead also amplified by the fact that experiments run in real time. We discuss execution branching support in Section 4.3.3.

Weighted Greedy. The greedy search algorithm also wastes some execution time. The reason is that the greedy search algorithm tries to find the strongest action for each message type. Therefore, if there are multiple effective actions for a message type, the greedy algorithm compares all actions and chooses the strongest attack, and actions that are effective but not the strongest will be discarded as seen in Fig.4.2 (b). The intuition is that the user will be more interested in a stronger attack. In reality, however, the user is more likely interested in finding all attacks that can be discovered. Therefore, the user will repeat the attack finding process again after finding the strongest attack — until the method does not find any more attacks. Time is much more critical in an emulated approach; thus, the order of attacks is less important than the total time required to find attacks.

We propose a *weighted greedy* search, showed in Fig.4.2 (c), that relies on the observation that certain types of malicious actions are more effective than other, regardless of message types. The algorithm attempts to learn what actions are more likely effective and use the information to improve the next search. The algorithm clusters malicious actions into several categories. The weight of each cluster can be preloaded. When a malicious action is tested, the ones belonging to the cluster with the highest weight are tried first. *Weighted greedy* also relies on execution branching and compares performance of the baseline and malicious runs from the same attack injection point. However, it stops the moment it encounters a malicious action that can be classified as an attack, i.e. the performance damage is bigger than Δ. To speed up convergence, the cluster weight corresponding to that malicious action, which was classified as an attack, is increased. If there is no action of which performance damage is larger than Δ, all the actions are evaluated, and the worst performance action will be chosen.

4.3.3 Execution branching

To effectively find performance attacks, we need to compare attack impact for attack and non-attack executions that are branched from an identical checkpoint (attack injection).

One approach to support execution branching is logging all the events and restoring the state of the system by replaying all events that lead to a particular execution point. In an emulated setting, this means logging either at the hyper level or at the application level, which requires modifications of the application, library or operating system. Both options are not feasible, the first because it requires modifying the environment, the second because of the high overhead, as the hypervisor can not separate which instructions are relevant to the application and thus needs to log every instruction executed for a VM.

A more feasible approach is to save and reload snapshots of the entire system [28, 70]. A snapshot of a distributed system includes the local states of all individual nodes and the state of the messages in transit [71, 72]. Therefore, to take a snapshot of a distributed system, we need to be able to save the state of all the nodes and the network and make sure they are consistent to each other. While the goal of taking a snapshot of a distributed system is the same as the Chandy-Lamport distributed snapshot algorithm [71], our environment makes the problem less complex because (1) the initiator of the snapshot process is not one of the participants, (2) participants in our environment have synchronized clocks, and (3) we have control over packets in flight because of the network emulator. Functionalities that need to be supported by the VMs and the network emulator are:

• *Saving and loading states*: Such features allow the creation of a snapshot and preparing the system to restart execution from certain states.

• *Pausing and resuming execution*: Pausing is needed in order to ensure the saved states of VMs and the network emulator are consistent to each other, while resuming is needed to restart execution.

Saving and loading states as well as pausing and resuming execution are important properties of VMs and supported by most hypervisors. However, typical network emulators do not provide such features.

To create a snapshot of the entire system, first the network emulator is paused, which will stop the virtual time from advancing, the emulator will continue creating objects for packets it is receiving, however, no more packets will be sent to any VM from the network emulator. Next, all the VMs will be paused, which will stop them from generating and sending more packets. A snapshot of each VM is taken first, then the snapshot of the network emulator is taken. Restoring happens in the reverse order of the snapshot process. The network emulator state is restored first, then the states of VMs are restored, the execution of VMs is resumed, and finally, the network emulator execution is resumed. For the approach to work correctly, the VMs and the network emulator must have the same perception of time.

4.3.4 Generating malicious message actions

We considered several design options for the component that generates malicious actions. As the goal is to create a realistic testing environment, we can not change the application, the libraries used, or the operating system. Thus, we design a malicious proxy inside the network emulator that intercepts packets outside of the application and applies malicious actions to those packets.

The network emulator needs to know what messages to intercept and what actions to perform on each message. The attack injection consists of changing the delivery of the action for the message delivery actions and changing the message for message lying actions. To inject message delivery actions, the proxy needs to recognize message boundaries, while to inject message lying actions, the malicious proxy needs to understand the message format and the types of different fields.

4.4 Implementation

In this section, we describe the implementation of Turret. We first give an overview of the implementation choices, then we describe how we implemented the malicious proxy

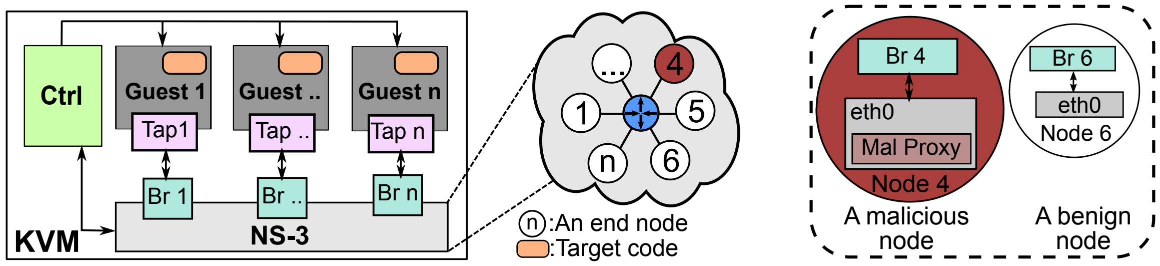


Figure 4.3.: The Turret Platform, illustrated for n nodes with node 4 designated as malicious

and the execution branching, including a snapshot management scheme that we develop for performance optimization.

4.4.1 Overview

We choose KVM as our virtualization technique and the NS3 emulator for the network emulation. Fig. 4.3 depicts Turret considering a distributed system with n participants. Each participant runs in its own VM, and each VM is mapped to an end node in NS3. The malicious proxy is placed in NS3 which makes coordination simpler, as all inter-node data flow is controlled by NS3. We modified NS3 so that it knows from the network configuration file, what end nodes are malicious, and it intercepts only those messages.

The attack finding algorithm is implemented by a *controller* which is a separate process that communicates with the network emulator and each individual VM. When NS3 intercepts a message from a malicious node, it asks the controller what actions it needs to perform on the message. The controller generates all the possible malicious actions based on the message type and message format and instructs NS3 what action to perform on the message.

It then monitors the applications running in VMs and stops execution after the window w which is used to measure performance. All applications running in VMs will report the observed performance back to the controller. To test each of the malicious actions for a message type, the controller also implements the execution branching by instructing the VMs and NS3 to pause, freeze, save the state, restore the state, and resume the execution.

To ensure coordination between NS3 and the VMs when restarting the execution from a branching point, the NS3 and the VMs must have the same perception of elapsed time and be synchronized. In Turret, this is done by using the host time as a reference and recalibrating the Time Stamp Counter(TSC) used by the VMs and modifying the internal clock of NS3. An alternative is to synchronize NS3 and the VMs with an external timing service.

4.4.2 Malicious proxy

The malicious proxy has two components: one that intercepts the packets and one that processes them according to a malicious action. We first describe how the malicious proxy intercepts packets then describe how to inject a malicious action.

We considered several options for implementing the malicious proxy, including inserting it inside the application or using library interposition to intercept system calls. One option is implementing the proxy within the application itself, where messages are created, but this restricts the applications we can test. Second option is implementing the proxy within the operating system, either by modifying the OS or by using library interposition to intercept calls into the operating system. However, library interposition limits the set of operating systems we can test, each requiring its own interposition framework. Outside the operating system, network traffic can be intercepted either by modifying the virtualized network hardware or placing a network trap along the data flow. However, since most of the data flow is already within NS-3, it is a convenient place to put the malicious proxy.

To ensure that all traffic is available to the proxy while not restricting the simulated network topology, we allow the malicious proxy to intercept packets along the ingress path of packets inside NS-3. We modify NS-3 to allow us to designate who the malicious nodes are, and only those corresponding VM packets are intercepted.

For TCP, we intercept packets inside the simulated network card, as shown in Fig. 4.3 for Node 4. Using the simulated network card is more convenient than the simulated network links, because the Ethernet device in NS-3 is an object that can provide a TCP stack. Once a TCP packet is intercepted it is forwarded up to the IP layer that is installed in the node. If the destination port of the packet matches one of the port numbers we are interested in, it forwards the packet up to TCP layer so that it can be passed to our malicious proxy.

Since UDP is stateless, we are able to optimize for this case, and can instead intercept packets in the Bridge module without the overhead of passing the packet up multiple layers. In this case, we simply strip off the UDP headers and pass the data on to the malicious proxy. Note that while this succeeds for the systems we considered for this chapter, when protocol messages may be larger than one packet, this optimization would be disabled, falling back to NS-3 standard processing as used for TCP.

As malicious actions are based on message types, the malicious proxy needs to determine the type of a message to inject malicious actions. The proxy thus requires the format of messages in order to act on different message types and to lie on fields of messages. We developed a small compiler that reads a message format description and generates C++ code compatible with a large set of binary wire protocols. The generated code is compiled and linked to the malicious proxy which uses it to identify message types and modify fields. Once the malicious proxy receives a message it determines the message type and contacts the controller who will provide an attack strategy. It then injects the specified attack on the message.

Here we provide a simple example of a message format that is required by the compiler:

```
MessageName {
  char first[128];
  uint16_t second;
  uint32_t type = 2;
  uint8_t numStructs;
  struct MessageSubStructure third[numStructs];
}
struct MessageSubStructure {
  uint32_t fourth;
  char fifth[20];
};
```

In this case, the user has defined the message MessageName. The message can contain any fields that the user defines, but we require one special field named “type”. This special field must be assigned some value (2, above), which uniquely identifies the message.

Our compiler can also handle complex fields, represented by defining a C struct. In case the message has a variable length, the compiler can interpret fields as indicating how long other fields are. It can also handle encapsulated messages, in case a distributed system is running on top of some other service. For example, a distributed system might utilize a reliable broadcast service to disseminate its messages. In either case, the generated code allows us to modify specific basic types that are inside the message.

The proxy obtains the list of malicious actions from the controller. The controller generated this list based on the message format that was provided by the user. Whenever the malicious proxy processes a message with possible malicious actions, it consults the controller to determine which attack, if any, should be conducted presently.

4.4.3 Support for execution branching

We describe modifications we made to KVM and NS3 to support execution branching. KVM hypervisor supports the pause, resume, save, and load operations needed to implement execution branching. However, some of the operations can be quite expensive and we implemented a page sharing aware snapshot management to reduce the overhead. NS3 does not provide operations for execution branching, so we implemented save, load, freeze, and resume functionality.

KVM - Page Sharing Aware Snapshot Management. Executing a branch can be quite expensive. A significant part of the overhead is due to saving snapshots of VMs. For example, for an application using 5 VMs, all operations take less than a second, while saving snapshots of VMs takes 15–17 seconds depending on what application is used.

Previous studies showed that VMs have many identical pages among them [73, 74]. This observation has been used to put more VMs in a host by sharing identical pages. This suggests that by sharing pages among snapshots, we can reduce the size of snapshots of VMs and the time to save and load as well. To reduce the cost of saving snapshots of VMs, we propose *page sharing aware snapshot management*. Specifically, we generate a shared snapshot across VMs for sharable pages and let each VM have an individual snapshot which contains only pages that can not be shared.

To implement our optimization we modified both KSM and KVM. KVM uses the KSM [74] Linux kernel feature to share identical pages across VMs. KSM takes care of merging and breaking pages transparently; it finds identical pages, tracks them, and merges them in one page if they are sharable and not frequently updated. Shared pages share a pfn (page frame number). To let KVM be aware of shared pages, we modified KSM to expose shared page information by adding an interface that verifies if a page is shared or not. Then, we modified KVM to use page sharing during save and load operations. During *save*, in addition to a snapshot for each VM, KVM will also create a shared page map file which contains the pfns and the contents of pages that are shared across VMs. While saving memory pages for the VMs, KVM queries KSM to find out if a page is shared, using the interface we added in KSM. If the page is shared, KVM adds the pfn and the page content to the shared page map . Then, it marks the address as shared and puts the pfn in the snapshot file of the VM. For non-shared pages, KVM puts the address and the content of the page in the snapshot file for the VM. During *load*, KVM first loads the shared page map into memory and indexes the contents by pfns. When loading the pages for VMs, if the flag of a page in the VM snapshot indicates that the page is shared, it retrieves the page contents from the shared page map using the pfn and copies it at the corresponding memory location. If the page is not shared, its contents are retrieved from the VM snapshot.

NS3 - Save, load, freeze, and resume. We implement save, load, freeze, and resume operations in NS3. To save and load the state of the network, we add save and load methods for all objects, events and the event queue of NS3. When saving a snapshot, NS3 iterates over its event queue and saves all events and related objects. Similarly, when loading, NS3 restores its event queue from all the saved events and objects. To freeze and resume the execution of the network emulator, we first let the network emulator be synchronized to a virtual clock instead of the system clock. When the network emulator is in the frozen mode, the virtual clock does not advance. As the virtual clock does not advance, NS3 does not process future events; however, NS3 continues its execution to safely finish operations such as copying buffers from VMs and generating packet objects if it was receiving packets from VMs at the moment it started the frozen operation.

4.5 Case study: PBFT

In this section, we demonstrate how to use Turret to find attacks in a distributed system implementation. We choose PBFT [2] as a target system. PBFT is designed to meet two standard criteria in distributed systems: safety and liveness. Safety ensures the system never reaches an inconsistent state, while liveness ensures the system makes progress eventually. As asynchronous distributed systems cannot be both safe and live at the same time, intrusion tolerant systems are usually designed to always maintain safety, and guarantee liveness only during periods of sufficient synchrony and connectivity. First we provide a description of the system PBFT, describe how to use Turret, then show our result.

4.5.1 PBFT

PBFT is a leader-based, state-machine replication protocol that provides correct service to correct participants, even when a fraction of nodes are compromised. Specifically, PBFT requires acknowledgments from $2f + 1$ out of $3f + 1$ servers to mask the behavior of f Byzantine servers. A client must wait for $f + 1$ identical responses to be guaranteed that at least one correct server assented to the returned value.

All replicas in PBFT participate in several protocols. When no faults occur all replicas participate in the *Normal Case* which ensures strong consistency by totally ordering all client updates. If no progress takes place in the system, to ensure liveness, replicas will initiate and participate in a *View Change* protocol intended to change the primary. Period-

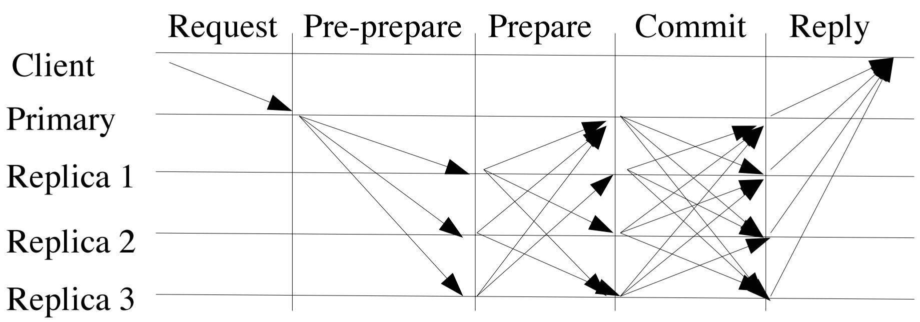


Figure 4.4.: PBFT normal case protocol

ically, each replica also participates in a *Status* protocol which allows a replica to notify other replicas what are the messages it has received so far. Finally, each replica participates in a *Garbage Collection* protocol. Table 4.1 presents all messages used by PBFT.

System description.

Normal case: This protocol is summarized in Fig. 4.4. A client initiates an update by sending a Request to the primary replica. The current primary then sends a Pre-Prepare message to all other replicas. If a replica accepts the Pre-Prepare it sends a Prepare message to all other replicas. When a replica receives $2f$ Prepare messages that match the local Pre-Prepare message, it can send a Commit to all replicas. A replica can order and execute the update when it receives $2f + 1$ Commit messages and then will send a Reply message to the client. The client waits for $f + 1$ Reply messages with the same values and then it will accept the result.

Garbage collection: This protocol allows replicas to discard messages from their logs. To do this safely, a replica has to be sure that the requests associated with these messages have been executed by at least $f+1$ replicas. Each replica periodically creates a checkpoint by saving the current state for a particular sequence number and sends a corresponding Checkpoint message to all other replicas demonstrating that its state was correct. Once a replica receives $2f + 1$ Checkpoint messages with the same values, it can independently prove that the state is correct and can safely discard the messages associated with that sequence number and earlier.

Table 4.1: Messages in PBFT protocol

Message	Sent by	Purpose
Request	Client	Request an operation
Pre-Prepare	Primary	Notify a new operation
Prepare	Backups	Let others know that the node received Pre-Prepare
Commit	All replicas	Agree on the message
Reply	All replicas	Reply to client
View-Change	All replicas	Suspect the primary may not be good
New-View	New primary	Notify that the view has changed
Status	All replicas	To keep the node up-to-date, send current sequences
Checkpoint	All replicas	Proof of a correct state, help garbage collection and efficient update

View change: This protocol evicts a primary when it is faulty, thus allowing the system to continue to make progress. Specifically, a backup replica will send a View-Change message if it has received a request from a client, and a progress timer expires before the request has been executed. When the primary of the new view has received $2f$ valid View-Change messages, it will send a New-View message to all other replicas. The New-View message contains all $2f$ valid View-Change messages, thus proving that it is the correct primary for the new view.

Status Protocol: This protocol keeps nodes up-to-date. A replica sends its latest sequence and stable checkpoint number in a Status message to all replicas in the system periodically. Upon receiving a Status message, if the sender is missing anything, the recipient will reply with the appropriate Checkpoint, View-Change, Pre-Prepare or other types of messages. The out-of-date replica can then use these to bring its state back to the current, agreed upon state. Additionally, if a replica receives a message that is in a different view or whose sequence number is significantly larger than the last checkpoint, it will also send a Status message.

Implementation.

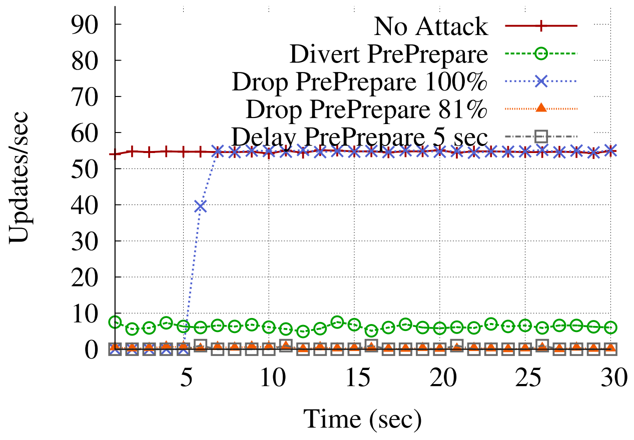
We use the C++ implementation of the original PBFT [2] available at [75]. The code uses UDP as its communication protocol. The client sends the Request to all replicas. A Checkpoint message is sent every 128 processed updates. The timer used to monitor progress in the system and to trigger view changes is 5 seconds and is not dynamically adjusted every time the view changes. Each replica periodically sends a Status message about every 200 ms. In case a Status message needs to be sent because of the difference with respect to the view or checkpoint, replicas rate-limit themselves to sending it only once every 100 ms.

4.5.2 Applying Turret on PBFT

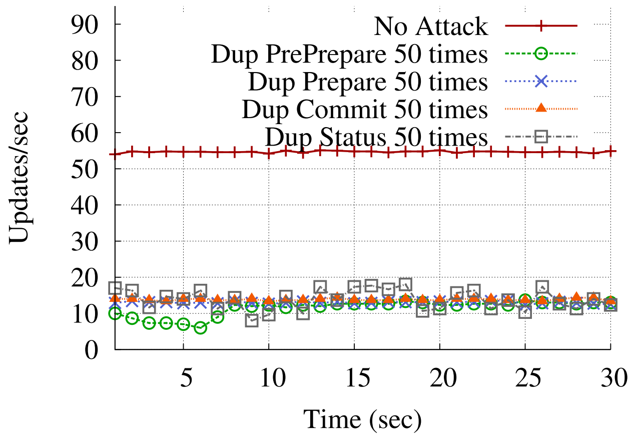
To apply Turret on PBFT, we first prepare it to run in the virtual machines and wrote scripts to launch the system and collect performance. The system launching script initiates the program with proper configuration. There are two performance scripts where one that passes the performance log to Turret (log update script) and one that parses the content and produces a performance score (performance script) that meets performance score requirements of Turret.

Turret requires two main properties for the performance score: 1) the bigger number should represent the worse performance and 2) the change of performance should be observable during the execution. To meet the second requirement, we used updates per second as our performance score metric. To satisfy the first requirement, we converted it by subtracting the number from a big enough fixed value that can not be achieved (9999 in this experiment). Our log update script was 70 lines long, and the performance script was 107 lines long.

Finally, to make the malicious proxy recognize message formats, we need to provide a message format file. As our format file resembles C/C++ style structure definition and PBFT was written C++, we mostly copied structure definitions from their header files then put the type identifiers. The format file we wrote for PBFT was 133 lines long.



(a) Attacks that limit progress



(b) Duplication attacks that cause DoS

Figure 4.5.: Updates per second for PBFT for the attacks found.

4.5.3 Discovered attacks

We test PBFT in two different configurations. The first one consists of 4 servers, thus able to tolerate one faulty server, and we consider both cases when the initial primary is malicious or a backup. The second one consists of 7 servers, thus able to tolerate two faulty servers, and we use this configuration to find attacks on View-Change messages. As we are not attempting to stress the system, we only use one client and do not pipeline requests. We use the default values for parameters, as used in [2].

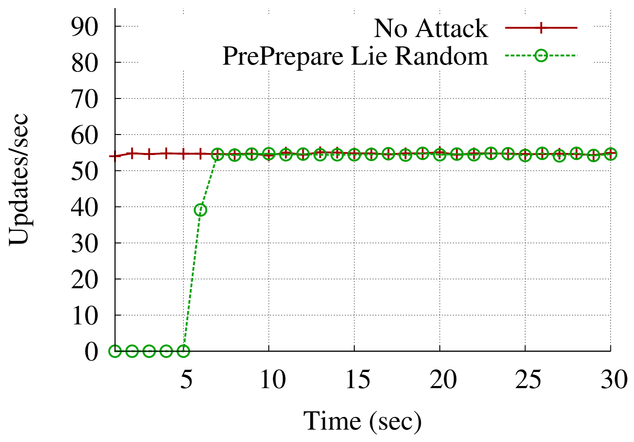


Figure 4.6.: Updates per second for PBFT for an attack that causes a view change

Attacks limiting progress.

Delay Pre-Prepare attack: This attack has been found previously by Amir et al. [62] where the primary can slow down the entire protocol by delaying the Pre-Prepare messages. Since the primary broadcasting Pre-Prepare is the first step to execute an update, no progress can be made without this message. If the delay is too long or the primary does not send out this message at all then replicas can detect that the primary is not good and change the view. Delaying Pre-Prepare longer than the timeout has the same effect as dropping the Pre-Prepare since the replicas will detect that the primary is faulty and change the view. Fig. 4.5(a) shows that the protocol can detect and recover from the case when the primary drops 100% of Pre-Prepare messages. However, if the primary delays the message shorter than the timeout triggering a view change, it can delay every execution continuously. Under the default configuration, we find that delaying the transmission of Pre-Prepare by 1 second drops the system throughput from 54.7 updates/sec to 0.9 updates/sec.

Drop Pre-Prepare attack: When a malicious primary needs to send Pre-Prepare messages, if the primary drops probabilistically the Pre-Prepare it can also cause degradation of throughput. Note that only two replicas need a copy of the Pre-Prepare to make progress. However, for many other cases a retransmission of Pre-Prepare is required to make progress. These attacks result in a performance of 1.32 updates/sec for the drop 81% attack, as seen in Fig. 4.5(a).

Attacks causing Denial of Service (DoS).

Delay Status attack: Upon receiving a Status, a node compares the sequence number in the message to its local sequence number. If the sender does not have the latest information then, the receiver retransmits that information. This mechanism is helpful for nodes to catch up on missing messages quickly since the protocol does not assume an environment guaranteeing message ordering or delivery. When a node receives a delayed Status message, the message has stale information, causing the receiver to believe the sender is out-of-date, triggering retransmission of all messages and updates that the sender is possibly missing. A malicious node can exploit this property by simply delaying Status messages, and causing receivers to retransmit many messages. Longer delays cause more retransmissions, but if the delay becomes too long, the receiver transmits a stable checkpoint instead of sending all individual messages and updates. By delaying Status by 1 second, the system throughput dropped down to 29.8 updates/sec, as seen in Fig. 4.7.

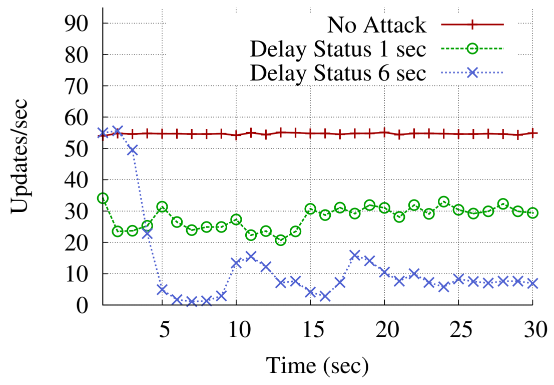


Figure 4.7.: Updates per second for PBFT for attacks on Status

Duplication attacks: We find that duplicating a few different types of messages causes a denial of service on replicas. If a primary duplicates Pre-Prepare messages, if a backup duplicates Prepare messages, or if any replica duplicates Commit or Status messages, then throughput decreases. We investigated more deeply and found increasing it was more effective. Duplicating Pre-Prepare, Prepare, and Commit messages 50 times result in throughput around 15 updates/sec. We show these results in Fig. 4.5(b). The decrease in throughput can be attributed to nodes having to process all the extra copies of the messages. We have also confirmed that these attacks are even more effective when verification of digital signatures is turned back on.

Attacks causing crashing.

These are attacks that crash nodes. We found that there are a few integer fields in Pre-Prepare and Status, that if any of them are replaced with negative values, all non-faulty replicas will experience a segmentation fault. Investigating deeper, these fields deal with sizes of variable length data structures that are contained within messages. However, the implementation trusts that these values will always be positive and does no error checking before utilizing the values. Using the configuration of 7 servers to trigger a view change, we found two different fields of the View-Change message where lying on them causes an assertion and a segmentation fault in all other replicas.

4.5.4 Discussion

Here, we discuss some of the insights revealed from using Turret on PBFT and extrapolate some of these observations to intrusion tolerant systems design and testing.

Tolerating malicious replica induced denial-of-service.

A very effective and not previously reported attack we found is related to the *Status* protocol. Specifically, one replica delayed a Status message, causing the system to essentially attack itself, by overloading the network and the other replicas with messages. As a result, one malicious replica caused non-faulty replicas to be so busy making sure that the malicious replica is up-to-date that they could not make real progress. Note that the design of PBFT [2] discusses the basic functionality of this protocol, but the *Status* protocol is not needed for ensuring and proving that PBFT provides safety and liveness. However, for practical purposes an implementation will need to use such a protocol. This attack illustrates the importance of testing real implementations since as we have shown, an attacker can abuse this functionality to make the system unusable. To protect itself against this attack, the *Status* protocol will need to be redesigned. A practical approach would be to limit how often a node resends the same data to a replica. Furthermore, as the attacker might send other messages indicating his current status (such as Pre-Prepare or Prepare messages), this information can also be used to cross-check that the replica is lying or simply does not need the data anymore. Other intrusion tolerant systems that use similar mechanisms to help slow replicas to keep up, will need to redesign such mechanisms since they will be vulnerable to similar attacks.

Tolerating message delivery attacks in stable networks.

One of the most effective attacks we find when running Turret on PBFT are attacks where the malicious primary delays or drops Pre-Prepare messages. Effectively, what the primary does is make the rest of the nodes believe that the network is slower than it actually is. As a result, the nodes will observe slower progress than what could have been possible. Note, that while this is not in contradiction with PBFT's liveness property which promises to eventually make progress, an attacker can still make the system practically unusable. We believe that systems implementing similar mechanisms will suffer from the same problem, and next-generation intrusion tolerant systems have to use mechanisms that will allow nodes to be able to discern the state of the network in a more collective way instead of relying on the perception of a single node. In fact, several recent research efforts have been in this direction [62, 69, 76]. We further evaluate these systems to verify that their implementation provides performance guarantees in Section 4.6.

Tolerating message lying attacks.

Our systematic evaluation of PBFT using Turret found very few attacks related to message lying. This was in part expected given that intrusion tolerant systems aim to give strong safety guarantees in the presence of compromised attackers that lie about their actions through messages. However, the effectiveness of delaying the Status message also suggests that there exist message lying attacks that our current search strategy simply has not found. To understand this, consider that a delayed Status message is essentially the same as a non-delayed later sent Status message fabricated to have the same content as the delayed message. The challenge with finding these remaining lying attacks is in the inherent redundancy in the message protocol, necessitating a coordinated lie told about multiple fields of a message. In our brute-force approach, we only attempt lies in a single field, because the number of attacks to try when lying on multiple fields grows exponentially in the number of fields. In the case of the Status message, if a malicious node lies about its current view, but correctly reports each Pre-prepare message, then only a minor amount of additional work will be performed. By contrast, the delayed Status message contains, in effect, coordinated lies about all values in the message, prompting more work by the handler. Automatically crafting messages that cause a particular control flow to be followed requires intelligent reasoning about what values should be placed in each message. We would expect that static analysis tools, such as symbolic execution, can be useful in finding such particular values.

Message-format aware fault injection.

We found several cases when message lying in PBFT lead to server crashes. Specifically, using Turret on PBFT, we were able to find 8 bugs that were not previously reported, 6 of which resulted in segmentation violations and 2 which resulted in assertion violations. While the main goal for designing Turret was the search for attacks focused on performance, one of the benefits of our fine-granularity message format manipulation is that we can inject more sophisticated faults than random bit-flipping [77]. Note that the PBFT code was not intended for production use so our findings should not be taken as a critique to PBFT implementation, but as a proof to the benefits of leveraging structural information for bug finding in intrusion tolerant systems implementation and making them ready for production use.

4.6 Results

In this section, we present results using Turret. First, we show the performance of Turret. Then, we show our results of using Turret for four other systems (written in C or C++). Finally, we describe our experience with using Turret in a distributed system class.

In all our experiments, we used an eight core machine that has an Intel(R) Xeon(R) CPU, with 8GB of RAM as a host machine and guest VMs are configured to have 128MB of RAM. We configure the emulated network to be a LAN setting with 1 ms delay between each node. For both greedy and weighted algorithms, we used a window of 6 seconds to observe the performance effect of an action. The systems we tested had timers of 5 seconds to start their recovery protocols. We chose a window of 6 seconds to give an opportunity for systems to use their recovery protocols. For every attack we find, we repeat the experiment 10 times and report the average of updates completed per second.

Our malicious proxy intercepts packets after they leave the VM and then modifies them. Since messages are digitally signed, benign nodes would simply discard modified messages upon receiving them. In order to explore lying attacks using our malicious proxy, we turn off the verification of digital signatures of messages. To prevent breaking any system assumption, we do not spoof or lie about who sent packets, i.e. our proxy lies about fields that the application itself could have lied about when creating the messages.

In addition to PBFT, we used Turret on four systems, Steward [66], Zyzzyva [67], Prime [68], and Aardvark [69]). We selected these systems because they are representative: they scale to wide area networks (Steward), they tolerate some performance attacks (Prime and Aardvark). We summarized all discovered attacks in Table 4.2.

Table 4.2: Summary of attacks found using Turret

System	Attack Name	Attack Description	Known
PBFT	Delay Pre-Prepare 1s	Delaying Pre-Prepare messages causes a significant performance degradation	[62]

continued on the next page

System	Attack Name	Attack Description	Known
PBFT	Drop Pre-Prepare 50%	Unlike dropping 100% of Pre-Prepare, which can be recovered by a view change, dropping 50% of Pre-Prepare messages causes a significant performance degradation	
	Delay Status 1s	Delaying Status messages by 1 sec makes benign nodes to believe the malicious node is behind, causing them to send many messages, resulting in performance degradation	
	Dup Pre-Prepare 50	Duplicating Pre-Prepare messages 50 times degrades the performance	[69]
	Dup Prepare 50	Duplicating Prepare messages 50 times degrades the performance	[69]
	Dup Commit 50	Duplicating Commit messages 50 times degrades the performance	[69]
	Dup Status 50	Duplicating Status messages 50 times degrades the performance	[69]
	Lie Pre-Prepare	Lying on some integer fields of Pre-Prepare messages causes benign nodes to crash	
	Lie Status	Lying on some integer fields of Status messages causes segmentation faults in benign nodes	
	Lie View-Change	Lying on view number field of View-Change messages causes segmentation faults in benign nodes	
Steward	Delay Pre-Prepare 1s	Delaying Pre-Prepare messages by 1 sec causes performance degradation	[62]
	Delay Proposal 1s	Delaying Proposal messages by 1 sec causes performance degradation	
	Delay Accept 1s	Delaying Accept messages by 1 sec causes performance degradation	
	Divert Accept	Diverting each Accept message to a randomly selected destination causes performance degradation	
	Drop Accept 100%	Dropping each Accept message causes performance degradation	
	Lie Accept	Lying on the global view field of Accept messages slows down progress	

continued on the next page

System	Attack Name	Attack Description	Known
	Dup GlobalViewChange	Duplicating GlobalViewChange messages a few times slows down progress	
	Dup CCSUnion	Duplicating CCSUnion messages a few times slows down progress	
Zyzzyva	Drop Reply 100%	Dropping Reply messages degrades system performance	
	Lie Reply	Lying on the size of Reply messages causes benign nodes to crash	
	Lie Order-Request	Lying on the sequence number of Order-Request messages causes benign nodes to crash	
	Lie NewView	Lying on the size field of NewView messages causes benign nodes to crash	
	Lie about message types	Lying on the type of messages causes benign nodes to crash	
Prime	Validation Errors	Several attacks cause benign nodes to crash or prevent progress due to validation errors	
	Drop PO-Summary	Dropping PO-Summary messages stops progress	
	Lie Pre-Prepare	Lying on the sequence number field of Pre-Prepare messages stops progress	
Aardvark	Delay Status 1s	Delaying Status messages slows down of the system	
	Lie Reply	Lying on the view number of Reply messages with random or negative values causes benign nodes to crash	
	Lie Status	Lying on the size related fields of Status messages causes benign nodes to crash	
	Lie PrePrepare	Lying on the number of large requests or non-deterministic choices of PrePrepare messages causes benign nodes to crash	

4.6.1 Performance

Here we present the performance of Turret in three different aspects: the network emulator, VM snapshots and attack finding algorithms.

Bundled net device.

NS3 provides a CSMA network device which supports emulation, but performs unnecessary processing. We implement a bundled network device which has less overhead. As shown in Fig. 4.8, while CSMA network device can not process more than 1000 packets per second, the bundled network device can process 2500 packets per second. We also measured the performance when injecting attacks, and observed that the overhead was similar to the benign case.

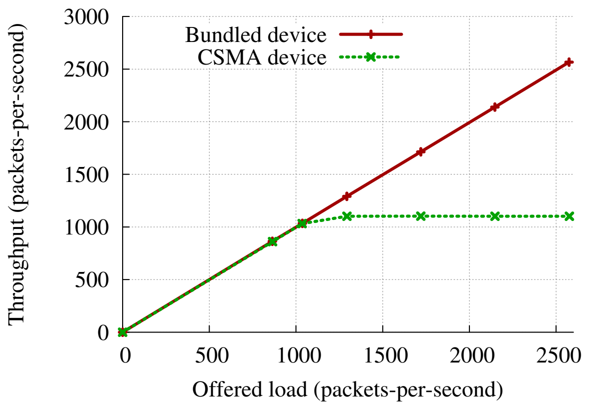


Figure 4.8.: Throughput for bundled and CSMA devices

Load/save for VM and NS3 snapshots.

We show the reduced overhead of our shared page optimization in Table 4.3. We ran an application that sends a monotonically increasing sequence to a server, with its hostname every second and measured the time to save and load snapshots of all VMs. Before saving, we paused VMs first according to our distributed snapshot scenario. We also measured the total size of snapshots of all VMs. We used 5 to 15 VMs and all reported numbers are averaged from 5 executions. The table shows that with shared snapshots, we reduce the

Table 4.3: Performance of save and load snapshot of VMs.

# of VMs	KVM with max BW			With Shared Snapshot			% Reduced	
	Time (sec)		Size (MB)	Time (sec)		Size (MB)	Save Time	Size
	Save	Load		Save	Load			
5	5.76	0.038	532	3.44	0.038	318	40.3	40.2
10	9.88	0.046	1,044	6.47	0.048	556	34.5	46.7
15	14.63	0.057	1,453	9.30	0.057	782	36.4	46.2

Table 4.4: Performance of the weighted greedy and the greedy algorithm

Attack name	Greedy (sec)	Weighted (sec)	% Reduced
Delay Pre-Prepare 1s	11444.5	70.7	99.4
Drop Pre-Prepare 50%	4407.6	63.0	98.6
Delay Status 1s	11649.6	110.4	99.1
Dup Pre-Prepare 50	15936.4	194.0	98.8
Dup Prepare 50	7258.0	71.0	99.0
Dup Commit 50	2642.9	612.5	76.8
Dup Status 50	6191.3	958.8	84.5
Lie Pre-Prepare	8579.5	113.8	98.7
Lie Status	18194.3	2552.3	86.0
Lie View-Change	1423.8	43.6	97.0

time to take snapshots of VMs between 34.5% to 40.3% for 5 to 15 VMs. Our executions show that to save states of 5 VMs, without any modification, it took 5.76 seconds with the maximum migration bandwidth and 3.44 seconds with shared page aware snapshot management. Because KVM limits the saving bandwidth by default, we let KVM use the maximum bandwidth in our experiments. With the default bandwidth, it took 15.24 seconds to save states of 5 VMs. Loading states of 5 VMs took 0.038 sec regardless of our modification.

Attack finding algorithm.

We compare the performance of the weighted greedy algorithm with the greedy algorithm for all the attacks found against PBFT. As shown in Table 4.4, weighted greedy algorithm found identical attacks 84.5 – 99.1% faster than the greedy algorithm.

4.6.2 Steward

Steward extends PBFT to scale to wide-area networks. Steward accomplishes this by taking a hierarchical approach where servers are grouped into different sites. This allows each site to be able to tolerate f Byzantine servers with $3f + 1$ total servers. Within each site servers run PBFT to replicate a state machine. Each site can then localize malicious behavior to its own site, thus requiring the inter-site state machine replication algorithm to not need to deal with Byzantine failures. Therefore, across sites servers run leader-based Paxos [78]. Steward servers participate in both a local and also a global *View Change* protocol when leaders do not provide progress. Servers also participate in a reconciliation protocol to recover missing globally ordered updates.

System description.

Normal case: Each site contains a representative that acts as a leader for that site. One site also acts as the leader site. When a server receives an update from a client, it will forward it to the representative of the leader site. The leader site will run PBFT, thus locally ordering the update. After this occurs, the site will produce a Proposal message that has been threshold-signed [79] by $2f + 1$ servers, proving that a majority of correct servers in the site agree on the ordering. This message is the first phase of the leader-based Paxos protocol, and the representative forwards this message to the other site's representatives. Each site will then produce a threshold-signed Accept message, and the representative of each site will forward it to the representatives of all other sites. After receiving a majority of sites' Accept messages, a server can globally order the update.

View change: Steward has a local and global view. Servers maintain a local timer L_t and global timer G_t. If no global progress is made in L_t time, servers send a Local-New-Rep message, and if $2f + 1$ are received, servers move to the next view. Similarly, if G_t expires, $2f+1$ servers in a site will threshold sign a Global-View-Change message, and if a majority of sites send such a message, the global view will change. Steward ensures that if any server globally ordered some update in a view, that all servers will become aware of that update during the view change. Furthermore, it prevents two Proposal messages from being constructed with different updates but the same sequence number. This is accomplished by running a *Construct Collective State* subprotocol, which gathers information on server's current state.

Reconciliation: If a server receives a Proposal message whose sequence number suggests a previous Proposal message is missing, it will send a Local-Reconciliation message to other servers in its site, they will then respond with the corresponding Proposal message if they have it. If no other local server has such a message, they will produce a threshold-signed Global-Reconciliation message, which they will send to the other sites. Servers from the other sites will then send a Complete-Ordered-Update which contains enough information to globally order the missing update.

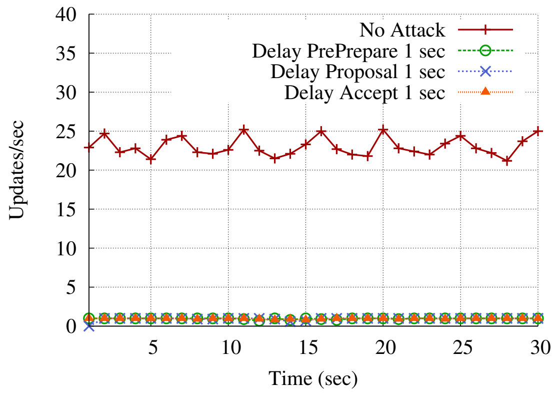
Implementation.

We use the C implementation of Steward that is available at [80]. The implementation uses as default values 600 ms for L_t and 1.3 sec for G_t. The implementation uses UDP as its communication protocol. Steward uses a custom version of PBFT and Paxos. As Steward is designed to scale to many servers, we test Steward in a 3-site configuration, where each of the 3 sites contains 4 servers. Since each site is able to tolerate one faulty server we run the experiment where each of the three initial primaries in each site is set to be malicious.

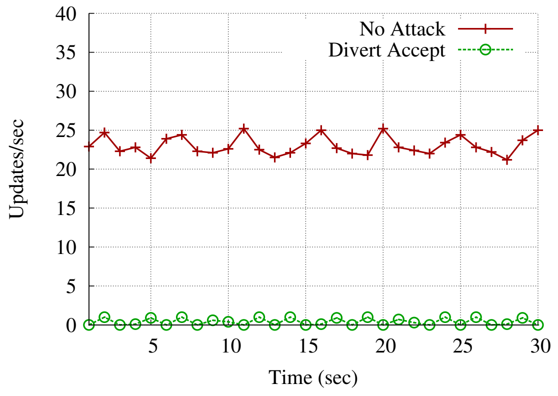
Attacks found.

We found several attacks that limit progress by delaying, diverting, dropping or lying actions. For example, we found the same attacks against the Pre-Prepare message in Steward as we found in PBFT, and as expected throughput is severely degraded, from 19.6 updates/sec to 0.9 updates/sec.

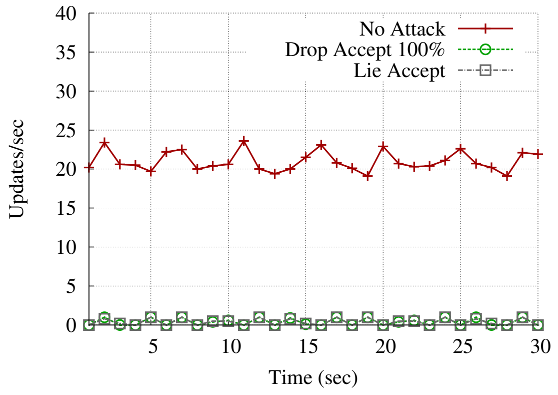
Another interesting attack was dropping Accept messages which resulted in continual performance degradation to 0.4 updates/sec. This result was counter-intuitive to us, since we expected a view change to occur, causing a benign node to become representative, and thus, performance would recover. We found instead that this did not occur due to fault tolerant code masking the behavior of the malicious node. We also found two attacks



(a) Delay attacks that limit progress

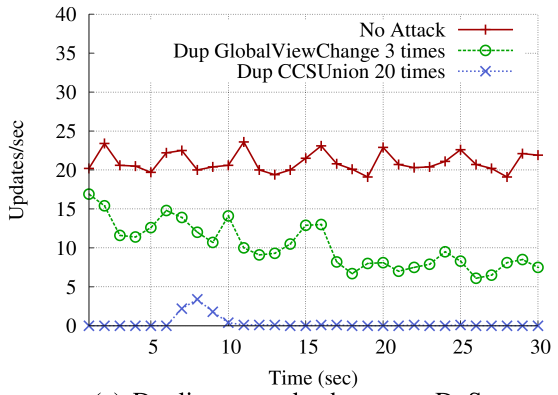


(b) Divert attack that limit progress



(c) Drop and lie attacks that limit progress

Figure 4.9.: Updates per second for Steward for the attacks found



(a) Duplicate attacks that cause DoS

Figure 4.10.: Updates per second for Steward for the attacks found

causing DoS. The attacks involved duplication of two different types of messages causing the performance to drop to 0.27 updates/sec.

4.6.3 Zyzzyva

Zyzzyva [67] is an optimized version of PBFT. In Zyzzyva, instead of first running a commit protocol to reach agreement, replicas optimistically adopt the order proposed by the primary and immediately respond to the client and the client checks if the agreement is correctly achieved.

System description.

Normal case: The key of Zyzzyva agreement protocol is to speculatively execute first. A client sends a request to the primary, the primary assigns a sequence and forwards it to all replicas. The replicas speculatively execute and respond to the client. The client gathers speculative responses, and if there was no fault, the client receives matching $3f+1$ Response messages to complete the request. When a client sees a fault in the responses (missing a message or inconsistency), the client sends a Commit to the replicas which ack with a Local-Commit message.

View change: Due to the speculative execution, Zyzzyva's view change protocol differs from PBFT. In PBFT once a replica suspects the primary, the replica stops progress

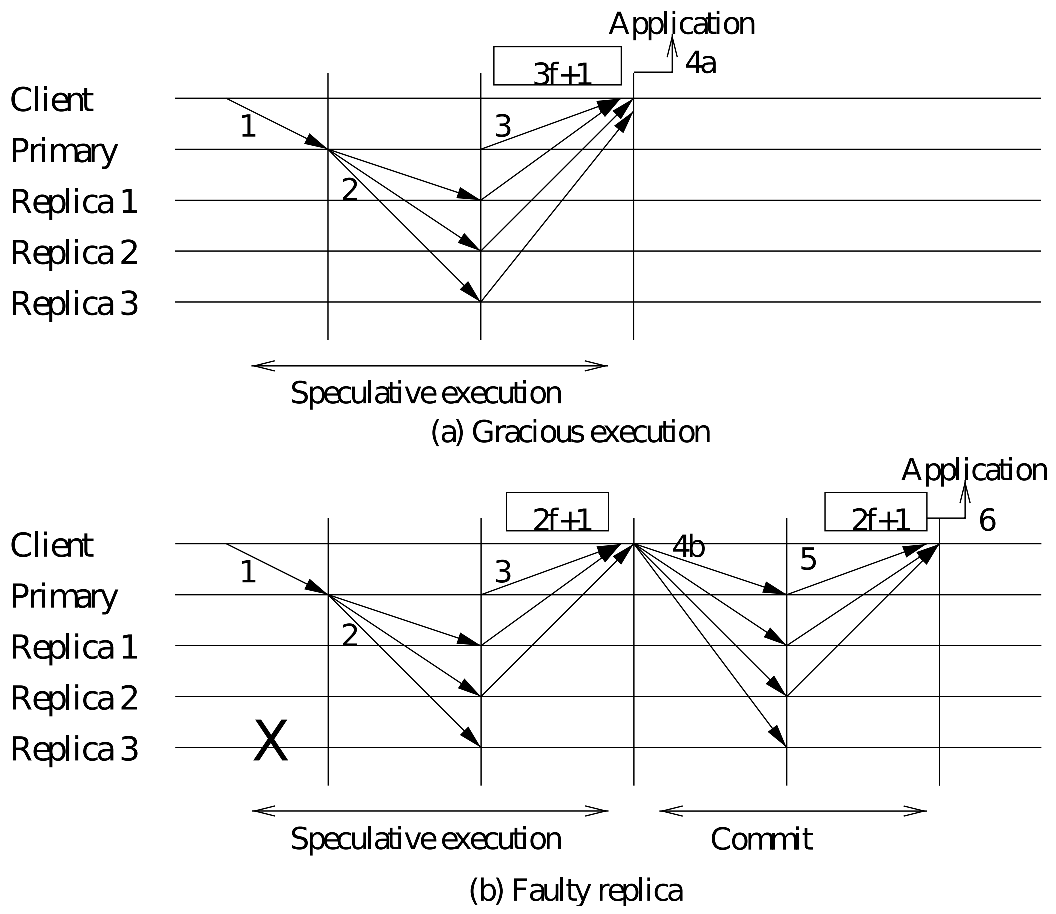


Figure 4.11.: Zyzzyva speculative execution

and participates in the view change protocol until a new primary is selected. In Zyzzyva, when a correct replica i suspects the primary of view v, it continues to participate in the view and sends an I-Hate-Primary message to all and continue to make progress. When replica i receives $f+1$ votes for I-Hate-Primary, it commits the view change. As Zyzzyva replicas never learn the outcome of the agreement, replicas need a way to learn a safe history. To deal with uncommitted requests in the View-Change, replicas add all Order-Req messages to the history since the last commit certificate, and a correct new primary extends the history in the new view.

Implementation: We use the C++ implementation of Zyzzyva obtained from its authors [67]. The implementation uses UDP as its communication protocol.

Attacks found.

We found several lying attacks that caused benign nodes to crash. We also found several attacks that slowed down the system. Dropping Reply removes the benefit of speculative execution and thus increases the latency of most requests. In the benign case, the minimum, average, maximum of the latency was 3.90 ms, 3.95 ms, and 4.02 ms, respectively. Dropping Reply from one node caused the minimum, average, and maximum latency to increase to 3.95 ms, 5.32 ms and 5.40 ms, respectively. Similarly, lying about the size of the message, making it a large negative value causes a similar effect and increases the latency to similar values.

4.6.4 Prime

As shown in [62], a malicious leader can reduce the throughput of PBFT and Steward greatly by delaying Pre-Prepare messages. Prime was designed to be able to tolerate such attacks and give stronger performance guarantees than PBFT. Similar to PBFT, Prime can tolerate f Byzantine servers with $3f + 1$ total servers.

System description.

Prime extends PBFT's main ordering protocol (shown in Sec. 4.5.1) by prepending a *Pre-order* protocol beforehand, which is depicted in Fig. 4.12. To ensure that a satisfactory amount of progress takes place in the system, each replica participates in a *Suspect Leader* protocol, and if no or slow progress takes place in the system they participate in a *View Change* protocol. Similar to the Status protocol of PBFT, Prime replicas participate in a *Reconciliation* protocol to ensure that all correct replicas stay up-to-date.

Normal case: Prime adds a pre-ordering protocol before the main ordering protocol as shown in Fig. 4.12. Each replica, upon receiving a client operation, broadcasts a PO-Request message, which binds it to a local sequence number. Upon receiving this message, each replica will broadcast a PO-Ack message, signifying that they are aware of the

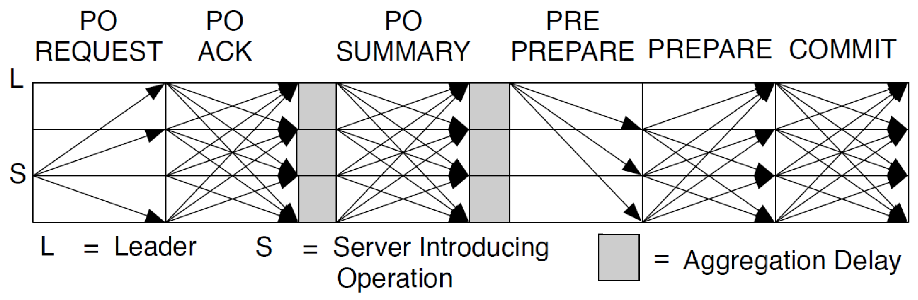


Figure 4.12.: Prime normal case protocol

client operation. Periodically, each replica will broadcast a PO-Summary message, which summarizes the largest local sequence number it is aware of for each replica. Each backup will also periodically send a Summary-Matrix message to the primary, which contains $3f + 1$ PO-Summary messages.

Suspect leader: If $2f+1$ replicas suspect that the current primary is too slow, they will initiate a view change. To accomplish this, each replica will measure the turnaround time between when it sends a Summary-Matrix message and when it receives a Pre-Prepare message that fully acknowledges all operations included in that Summary-Matrix message. At the same time, each replica measures round-trip times between itself and other replicas and computes how long it would take other replicas to send Pre-Prepare messages. If a replica finds that there are $2f + 1$ other replicas that would be faster than the current primary, it will send a New-Leader message, and if $2f + 1$ of these are received it will start the view change protocol.

View change: The view change protocol for Prime differs from that of PBFT in that it ensures that no one replica can slow down the process [81]. To facilitate this, Prime uses a reliable broadcast service to transmit messages about previously collected, but not totally ordered operations. After $2f + 1$ replicas have collected enough state, the replicas will agree on a total order for the operations, successfully moving into the next view.

Reconciliation: When servers receive Pre-Prepare messages for which they do not have the corresponding client operation, they will broadcast a Reconciliation message which indicate the missing operations. Upon receiving a Reconciliation message, a server will resend the corresponding PO-Request to that server.

Implementation.

We obtained an implementation of Prime from its authors [62] then updated it to include the suspect leader and view change protocols [81]. Prime uses UDP as its main communication protocol between replicas.

Attacks found.

Crashing attacks: We found 7 different lying attacks that resulted in messages with incorrect values causing the system to crash or stop making progress. Some of these validation errors were subtle, such as making sure that Pre-Prepare messages start with sequence number 1 instead of 0, or validating one message field before using it to validate other data.

Dropping attacks: One type of attack revealed an issue that disallowed replicas from forming a quorum when there was one attacker. For some attacks, the attacker crashed while he was the leader. Normally, a view change would occur as replicas would realize that there are $2f + 1$ other replicas that have faster turnaround times. However, replicas were not measuring the RTT to themselves, and as specified in [81] they were initially setting their RTT to infinity. Therefore, after the leader crashed, each replica would believe that only $2f$ replicas can communicate and that no replica would be faster than the failed replica. This resulted in no view change occurring, and as depicted in Fig. 4.13

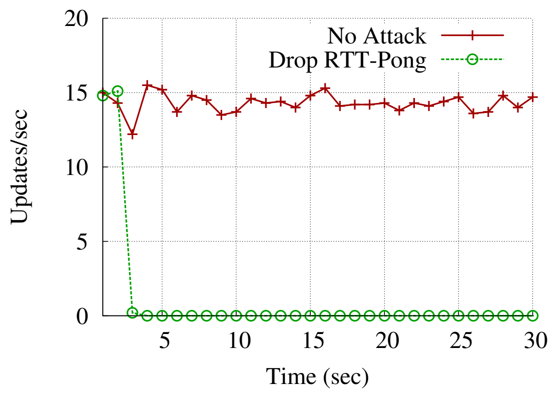


Figure 4.13.: Updates per second for Prime for quorum related attacks

We found several attacks in which dropping the PO-Summary message halted progress because a quorum could not be formed even if one existed.

Lie Pre-Prepare attack: The most interesting attack we found is one that involved lying on sequence numbers on a Pre-Prapare message that created a sequence of events that caused the suspect leader protocol to never be initiated. This protocol is the protection mechanism that allows the system to remove a leader that attempts to slow down the entire system.

4.6.5 Aardvark

Aardvark [69] is a system that is designed to tolerate performance attacks on PBFT and likewise systems. The authors notice that one faulty server or client can degrade the performance significantly and corner cases are not handled well. Aardvark follows the PBFT protocols and makes different design choices to make the system robust.

System description.

Normal case: The basic communication pattern of Aardvark is the same as in PBFT. However, Aardvark limits a client's ability to disrupt the service and adds resource isolation in the protocol. Aardvark does not trust clients and adds a non-repudiation mechanism and black-list filtering in the processing using digital signatures. Also, to prevent flooding problems, Aardvark uses separate network interface controllers (NICs) for each connection. Aardvark uses separate work queues for message processing from clients and replicas. If a faulty server causes network flooding, Aardvark server shuts down the associated NIC.

View changes: Aardvark uses the same view change mechanism of PBFT, but it is different in how a view change is initiated. While other protocols regard a view change as an expensive cost that needs to be paid upon the presence of faulty primary, Aardvark invokes view changes on a regular basis to prevent a faulty primary achieving absolute control over the system. Replicas monitor the performance and slowly raise the level of minimal acceptable throughput. Replicas initiate a view change when the primary no longer provides the acceptable throughput. Periodic view changes bring short slow downs; however, these slow downs can prevent the danger of significant performance damage.

Implementation.

We use the C++ implementation of the Aardvark paper [69] obtained from the authors. The authors used PBFT implementation as their base. We use one client, 4 servers and set other parameters as the authors provided.

Attacks found.

Like other systems, we run Aardvark with 4 servers and one client. Due to Aardvark forcing clients to only submit one request at a time, we do not pipeline requests.

Lie Reply Attack: The Reply message contains the current view. Lying on this value with a random or the lowest negative number can cause benign nodes to crash.

Lie PrePrepare Attack: The PrePrepare message contains the number of large requests. This is then used as an index to an array. However, this index is a signed type, and the code does not check if this value is positive. Lying about this value causes benign nodes to crash. Similarly, the message has a field for size of non-deterministic choices and lying on this with negative values causes benign nodes to crash.

Lie Status Attack: The Status has two size fields and lying about those fields with negative values also causes benign nodes to crash.

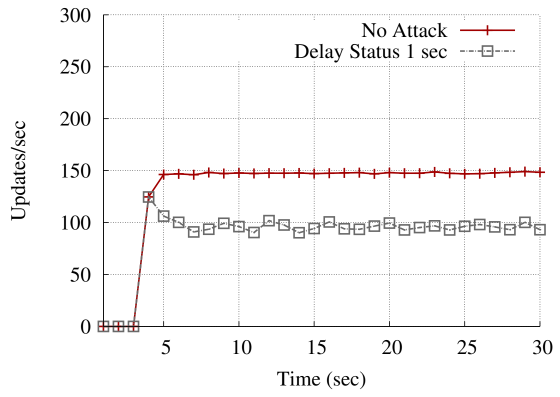


Figure 4.14.: Updates per second for Aardvark for Status delay attack

4.6.6 Using Turret in a distributed systems class

We used our platform Turret in a graduate distributed systems course during Spring 2013. Turret was used as a testing platform for the programming assignments given in this course. The projects were designed based on Paxos [82], Byzantine Generals Problem [83], and Total Order Multicast [84]. Students were given access to Turret since the beginning of the semester so that they could leverage the platform to test the robustness of their ongoing assignment. Both the students and the instructor tested the same unmodified binary under the same conditions without the need to implement the malicious test case scenarios code. To prepare for the platform, students were given a 30-min demo and supplied with an instruction manual on how to use the platform. Based on the anonymous, *IRB-approved* surveys collected from this course, students rated the effectiveness of Turret in finding vulnerabilities and bugs as, on average, 4.4 out of 5.

4.7 Summary

In this chapter, we proposed Turret, a platform that can automatically find attacks on unmodified implementations of distributed systems. We demonstrate Turret, by applying it to several intrusion tolerant systems, PBFT [2], Steward [66], Zyzzyva [67], Prime [68], and Aardvark [69]. We were able to find several protocol-level attacks that were previously undocumented and also some implementation bugs that caused all non-faulty replicas to crash. We believe Turret can greatly enhance the development process of building robust distributed systems.

5 RELATED WORK

In this chapter, we present related work, including automatic debugging techniques, automatic attack finding techniques and other tools to test unmodified implementations.

Automated debugging techniques, such as model checking, have been in use for many years. Most similar to our work is CrystalBall [38], where Yabandeh *et al.* utilize state exploration to predict safety violations and steer the execution path away from them and into safe states in the deployed system. Nodes predict consequences of their actions by executing a limited state exploration on a recently taken snapshot, which they take continuously. Since a violation is predicted beforehand, it is possible to avoid actions that will cause the violation. The Mace model checker is utilized for safety properties and state exploration. However, they do not consider performance metrics and thus can only find bugs or vulnerabilities that cause safety violations but not performance degradation.

Along with model checking, symbolic execution is another popular automatic debugging technique [39, 85, 86]. Symbolic execution focuses on generating test cases that will maximize the code coverage; symbolic execution engines treat inputs as abstract symbols rather than real values and build constraints that are achieved from conditional branches. Symbolic execution does not consider attacks or performance problems, and it suffers from path explosion similarly to model checking suffering from state explosion.

Automatic debugging techniques are used to find performance problems [29, 87] and their focus is to help diagnosis of problems that happen under deployment. Attariyan *et al.* instrument binaries to find performance anomalies and diagnose the root-cause of the problem using information flow tracking [29]. Nagaraj *et al.* use log analysis to understand the cause of performance problems in distributed systems by comparing good and bad executions using machine learning techniques. While our goal is to find problems under realistic test environments to deploy safe implementation, their goal is to develop techniques that can be deployed with the implementation to help diagnosing the cause of the problem when a problem happens.

Many previous works have also used debugging techniques for the purpose of automating the process of discovering or preventing attacks. Proving the absence of particular attacks has also been explored in the context of limited models and environments [88–90]. These debugging techniques include static and dynamic analysis [4, 40–44], model checking [88–90], and fault injection [3,45,91]. Below we summarize the works that are focused on discovering attacks and are most similar to our own.

Finding vulnerabilities in distributed systems has recently been explored by Banabic *et al.* [45]. They employ a fault injection technique on PBFT [2] to find combinations of MAC corruptions that cause nodes to crash. They require the user to configure the fault injector so that it can execute attacks that a user expects and it can not lie based on message fields. Our work is more general, as we do not require the user to provide specifics. Also, while they focus on finding all possible inputs that cause a single kind of vulnerability, our solutions search on basic malicious actions to find new attacks and vulnearbilities.

Stanojevic *et al.* [3] develop a technique for automatically searching for gullibility in protocols. They experiment on the two-party protocol ECN to find attacks that cause a malicious receiver to speed up and slow down the sending of data. Their technique uses a brute force search and considers lying about the fields in the headers of packets and also drops packets. As they also utilize Mace they are able to conduct protocol dependent attacks. Our work differs in that we focus on large-scale distributed systems, incorporate a fault injector that includes more diverse message delivery and lying actions, and use a greedy approach to avoid brute forcing.

Kothari *et al.* [4] explore how to automatically find lying attacks that manipulate control flow in implementations of protocols written in C. Their technique focuses on searching for a sequence of values that will cause a particular statement in the code to be executed many times, thus potentially causing an attack. Their method first utilizes static analysis of the code to reduce the search space of possible attack actions and then uses concrete execution to verify the attack. However, to utilize the technique the user must know ahead of time what parts of the code, if executed many times, would cause an attack, which may not always be obvious. Our systems, on the other hand, utilize an impact score to direct its search. Furthermore, some distributed systems, such as Vivaldi, do not have attacks on them that manipulate control flow, but only attacks that involve lying about state. Such attacks would go undiscovered by this technique.

Fault injectors often focus on testing correct recovery of the system when benign faults occur. Thus, they do not always consider Byzantine faults nor try to find attacks. For example, Marinescu *et al.* [92] develop a library level fault injection engine and automated techniques for reliably identifying errors that applications may encounter. Their techniques analyze the target program binary and compose a scenario that indicates when, where, and which fault to inject. They evaluated their techniques on the PBFT implementation and found two crash problems that can happen with benign faults. Gunawi *et al.* [34] develop a framework FATE that can systematically test faults and recovery of cloud systems. FATE allows to inject faults and test if cloud systems can recover from those faults correctly in a systematic way.

Yang *et al.* develop a model checker MODIST for unmodified distributed system implementations [35]. MODIST uses library interposition (WinAPI) to control non-determinisms in the system and use logging and replaying to reproduce a state. Because MODIST uses library interposition, it only supports Windows operating system, and it does not consider malicious components.

Fuzzing is another technique used to find vulnerabilities in implementations [41, 93]. While traditional fuzzing has been a black-box approach, new fuzzing techniques leverage code analysis. Notably, Sage [41] combines fuzzing with symbolic execution to extend code coverage (test wider). Dowser [93] leverages symbolic execution to guide fuzzing to examine corner cases of buffer overflow (test deeper). Fuzzers focus on finding bugs in processing input strings or input files and do not consider timings of messages. Timings are important for performance problems or distributed systems.

Abdelnur *et al.* developed KiF, a fuzzer to find vulnerabilities in SIP protocol [94]. KiF uses the state machine of SIP protocol to guide their fuzzing. KiF is similar to our approach in that it tries to find vulnerabilities in protocols. However, it is different from our work because it focuses on a specific protocol rather than general distributed systems.

6 CONCLUSION

Securing distributed systems against performance attacks has previously been a manual process of finding attacks and then patching or redesigning the system. Manually finding attacks in distributed system implementations is a painstaking task that requires an expert of the implementation and the environment to spend a long time due to the complexity of the implementations. This thesis tries to take a step to automate the process of finding performance attacks in distributed system implementations.

In a first step towards automating this process, we presented Gatling, a simulator based framework for automatically discovering performance attacks in distributed systems. Gatling uses a model-checking exploration approach on malicious actions to find behaviors that result in degraded performance. We provide a concrete implementation of Gatling for the Mace toolkit. Once the system is implemented in Mace the user needs to specify an impact score in a simulation driver that allows the system to run in the simulator.

To show the generality and effectiveness of Gatling, we have applied it to nine distributed systems that have a diverse set of system goals. We were able to discover 48 attacks in total, and for each system we were able to automatically discover attacks that either stopped the system from achieving its goals or slowed down progress significantly. While some of the attacks have been previously found manually through the cleverness of developers and researchers, we show that the amount of time Gatling needs to find such attacks is small. Therefore, we conclude that Gatling can help speed up the process of developing secure distributed systems.

Despite the effectiveness of Gatling, a simulation based approach has a fundamental limitation that it can only test implementations that are written in a specific language, and it can not test vulnerabilities that are caused by interaction of the implementation and the environment. Therefore, as a second step, we proposed Turret, a platform that can automatically find attacks on unmodified implementations of distributed systems. Turret takes an emulation based approach to test binaries that are written in any language that is running on an environment that is as similar to the deployment environment as possible.

We demonstrate Turret, by applying it to several intrusion tolerant systems, PBFT [2], Steward [66], Zyzzyva [67], Prime [68], and Aardvark [69]. We were able to find several protocol-level attacks that were previously undocumented and also some implementation bugs that caused all non-faulty replicas to crash. We believe Turret can greatly enhance the development process of building robust distributed systems.

Our work does not provide a complete solution to the problem of finding attacks in implementations of distributed systems automatically. Rather we show the possibility of effectiveness and importance of automatic testing of real implementations under adversarial scenarios. We leave a few remaining problems as future work. The first issue is handling encrypted messages. There is a trade-off between the generality of the solution and handling message encryption: a more general solution like Turret can not handle encryption, and Gatling has the opposite problem. Finding a sweet-spot between generality of the tool and ability to handle encryption can enhance testing distributed systems with message encryption. The second issue is scalability, especially for the emulation base approach. There are two different types of scalability problems to tackle: the number of nodes the framework can test, and the number of attack scenarios to test. To allow testing more number of nodes, Turret needs to use several machines and execution branching will require synchronizing and manipulating clocks of all virtual machines across multiple machines and the network emulator. To test more attack scenarios, we can use multiple machines for parallelism, and it requires some global controller that will orchestrate multiple Turret instances. Finally, the problem of crafting smarter attack strategies. To achieve this goal, one could take more white-box style approaches. For example, if one can track the internal state of nodes, the information can help eliminating strategies that are irrelevant or find better ways to lead nodes into an incorrect status.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Stephanie Mlot. Infographic: Just how big is amazon.com? http://www.pcmag.com/article2/0,2817,2413305,00.asp, December 2012.

[2] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI)*, 1999.

[3] Milan Stanojevic, Ratul Mahajan, Todd Millstein, and Madanlal Musuvathi. Can you fool me? Towards automatically checking protocol gullibility. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, 2008.

[4] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding protocol manipulation attacks. In *Proceedings of the SIGCOMM Conference (SIGCOMM)*, 2011.

[5] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[6] Lorenzo Leonini, Étienne Rivière, and Pascal Felber. Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[7] Matt Welsh, David E. Culler, and Eric A. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2001.

[8] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using random subsets to build scalable network services. In *Proceedings of the Symposium on Internet Technologies and Systems (USITS)*, 2003.

[9] Adolfo Rodriguez, Dejan Kostić, and Amin Vahdat. Scalability in adaptive multi-metric overlays. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[10] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, , and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2003.

[11] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the Middleware conference (Middleware)*, 2001.

[12] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2004.

[13] Dejan Kostić, Alex C. Snoeren, Amin Vahdat, Ryan Braud, Charles Killian, James W. Anderson, Jeannie Albrecht, Adolfo Rodriguez, and Erik Vandekieft. High-bandwidth data dissemination for large-scale distributed systems. *Transactions on Computer Systems (TOCS)*, 26(1), 2008.

[14] Shiding Lin, Aimin Pan, Zheng Zhang, Rui Guo, and Zhenyu Guo. WiDS: An integrated toolkit for distributed systems development. In *Proceedings of the Conference on Hot topics in Operating Systems (HOTOS)*, 2005.

[15] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, USA, 1996.

[16] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the SIGCOMM Conference (SIGCOMM)*, 2001.

[17] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2007.

[18] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2005.

[19] Network simulator 3. http://www.nsnam.org/.

[20] Georgia tech network simulator.
http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS/.

[21] p2psim: A simulator for peer-to-peer protocols, 2013.
http://pdos.csail.mit.edu/p2psim/.

[22] Emulab-network emulation. http://www.emulab.net/.

[23] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2002.

[24] cyber-defense technology experimental research laboratory testbed.
http://www.isi.edu/deter/.

[25] PlanetLab. Planet lab. http://www.planet-lab.org, August 2002.

[26] Resilient overlay networks. http://nms.csail.mit.edu/ron/.

[27] Global environment for network innovation. http://www.geni.net.

[28] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[29] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI)*, OSDI'12, Berkeley, CA, USA, 2012. USENIX Association.

[30] Xuezheng Lui, Wei Lin, Aimin Pan, and Zheng Zhang. Wids checker: Combating bugs in distributed systems. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, Massachusetts, April 2007.

[31] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[32] Gerard J. Holzmann. The model checker spin. *Transactions on Software Engineering*, 23, 1997.

[33] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[34] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. Fate and destini: A framework for cloud recovery testing. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[35] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[36] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI)*, 2002.

[37] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[38] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystal-Ball: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[39] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 2012. ACM.

[40] Chia Yuan Cho, Domagoj Babi, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the Usenix Security Symposium*, 2011.

[41] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.

[42] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[43] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2008.

[44] Wenhua Wang, Yu Lei, Donggang Liu, David Kung, Christoph Csallner, Dazhi Zhang, Raghu Kacker, and Rick Kuhn. A combinatorial approach to detecting buffer overflow vulnerabilities. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2011.

[45] Radu Banabic, George Candea, and Rachid Guerraoui. Automated vulnerability discovery in distributed systems. In *Proceedings of the Workshop on Hot Topics in Dependable Systems (HotDep)*, 2011.

[46] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *Proceedings of the SIGCOMM Conference (SIGCOMM)*, 2004.

[47] Yang-Hua Chu, Aditya Ganjam, T. S. Eugene Ng, Sanjay Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang. Early experience with an internet broadcast system based on overlay multicast. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2004.

[48] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the Workshop on Networked Group Communication (NGC)*, 2001.

[49] David John Zage and Cristina Nita-Rotaru. On the accuracy of decentralized virtual coordinate systems in adversarial networks. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2007.

[50] Jeffrey Seibert, Sheila Becker, Cristina Nita-Rotaru, and Radu State. Securing virtual coordinates by enforcing physical laws. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, Washington, DC, USA, 2012. Computer Society.

[51] Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, Berkeley, CA, USA, 2005. USENIX Association.

[52] P. Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 1997.

[53] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[54] Charles Killian, Karthik Nagarak, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2010.

[55] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the International Conference on Peer-to-Peer Computing Economics*, 2003.

[56] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the SIGCOMM Conference (SIGCOMM)*, SIGCOMM '01, New York, NY, USA, 2001. ACM.

[57] Miguel Castro, Peter Drushel, Ayalvadi Ganesh, Antony Rowstron, and Dan Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI)*, 2002.

[58] AAron Walters, David Zage, and Cristina Nita-Rotaru. A Framework for Mitigating Attacks Against Measurement-Based Adaptation Mechanisms in Unstructured Multicast Overlay Networks. *Transactions on Networking*, 16, 2008.

[59] Jonathan Ledlie, Paul Gardner, and Margo Seltzer. Network coordinates in the wild. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[60] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the SIGCOMM Conference Internet Measurement Workshop (SIGCOMM-IMW)*, 2002.

[61] Eric Chan-Tin, Victor Heorhiadi, Nicholas Hopper, and Yongdae Kim. The frog-boiling attack: Limitations of secure network coordinate systems. *Transactions on Information and System Security*, 14(3), November 2011.

[62] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Byzantine replication under attack. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2008.

[63] Hyojeong Lee, Jeff Seibert, Charles Killian, and Cristina Nita-Rotaru. Gatling: Automatic attack discovery in large-scale distributed systems. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[64] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2013.

[65] Md. Endadul Hoque, Hyojeong Lee, Rahul Potharaju, Charles E. Killian, and Cristina Nita-Rotaru. Adversarial testing of wireless routing implementations. In *Proceedings of the Conference on Security and Privacy in Wireless and Mobile Networksonference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2013.

[66] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *Transactions on Dependable and Secure Computerng*, 7(1), January 2010.

[67] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2007.

[68] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *Transactions on Dependable and Secure Computerng*, 8(4), July 2011.

[69] Allen Clement, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[70] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2006.

[71] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *Transactions on Computer Systems (TOCS)*, 3(1), 1985.

[72] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Elsevier JPDC*, 18(4), 1993.

[73] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[74] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium (OLS)*, July 2009.

[75] PBFT. http://www.pmg.lcs.mit.edu/bft/.

[76] G. Santos Veronese, M. Correia, AN. Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the Symposium of Reliable Distributed Systems (SRDS)*, 2009.

[77] Sonya Johnson Wierman. Vajra: Distributed fault injection for dependability evaluation. Master's thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, May 2006.

[78] Leslie Lamport. The part-time parliament. *Transactions on Computer Systems*, 16(2), May 1998.

[79] Victor Shoup. Practical threshold signatures. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2000.

[80] Steward. http://dsn.jhu.edu/download/download_steward.cgi.

[81] Jonathan Kirsch. *Intrusion-tolerant replication under attack*. PhD thesis, The Johns Hopkins University, Baltimore, Maryland, USA, February 2010.

[82] Jonathan Kirsch and Yair Amir. Paxos for system builders. Technical report, 2008.

[83] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *Transactions on Programming Languages and Systems (TOPLAS)*, 4(3), 1982.

[84] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *Transactions on Computer Systems (TOCS)*, 5(1), 1987.

[85] Raimondas Sasnauskas, Philipp Kaiser, Russ Lucas Jukic, and Klaus Wehrle. Integration testing of protocol implementations using symbolic distributed execution. In *Proceedings of the International Conference on Network Protocols (ICNP)*, ICNP '12. Computer Society, 2012.

[86] Stefan Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2011.

[87] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, NSDI'12, Berkeley, CA, USA, 2012. USENIX Association.

[88] Alessandro Armando and Luca Compagna. SAT-based model-checking for security protocols analysis. *International Journal of Information Security*, 7, January 2008.

[89] Bruno Blanchet. From secrecy to authenticity in security protocols. In *Proceedings of the International Static Analysis Symposium*. Springer, 2002.

[90] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hankes Drielsma, Pierre-Cyrille Heám, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, Davud Von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proceedings of Computer Aided Verification*, 2005.

[91] Joao Antunes, Nuno Neves, Miguel Correia, Paulo Verissimo, and Rui Neves. Vulnerability Discovery with Attack Injection. *Transactions on Software Engineering*, 36, 2010.

[92] Paul Marinescu and George Candea. Efficient testing of recovery code using fault injection. *Transactions on Computer Systems (TOCS)*, December 2011.

[93] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the Usenix Security Symposium*, 2013.

[94] Humberto J. Abdelnur, Radu State, and Olivier Festor. KiF: A stateful SIP fuzzer. In *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications (IPTComm)*, pages 47–56. ACM, 2007.

VITA

VITA

Hyojeong Lee was born in Seoul, Korea. She received her bachelors and masters degrees from the department of Computer Engineering in Hongik University, Seoul, Korea. She received her Ph.D. in Computer Science in 2014 from Purdue University. During her time at Purdue, she was a member of the Dependable and Secure Distributed Systems Lab and was affiliated with the Center for Education and Research in Information Assurance and Security (CERIAS). Her research focused on finding performance attacks in distributed system implementations and other systems. She also conducted research in finding concurrency bugs in web systems automatically, applying automatic attack finding in other areas such as wireless routing protocol implementations and network transport layer protocol implementations, and developing a programming model that simplifies distributed system implementation.