Gatling: Automatic Attack Discovery in Large-Scale Distributed Systems

Hyojeong Lee, Jeff Seibert, Charles Killian and Cristina Nita-Rotaru
Department of Computer Science
Purdue University
{hyojlee, jcseiber, ckillian, crisn}@purdue.edu

Abstract

In this paper, we propose Gatling, a framework that automatically finds performance attacks caused by insider attackers in large-scale message-passing distributed systems. In performance attacks, malicious nodes deviate from the protocol when sending or creating messages, with the goal of degrading system performance. We identify a representative set of basic malicious message delivery and lying actions and design a greedy search algorithm that finds effective attacks consisting of a subset of these actions. While lying malicious actions are protocol dependent, requiring the format and meaning of messages, Gatling captures them without needing to modify the target system, by using a type-aware compiler. We have implemented and used Gatling on six systems, a virtual coordinate system, a distributed hash table lookup service and application, two multicast systems and one file sharing application, and found a total of 41 attacks, ranging from a few minutes to a few hours to find each attack.

1 Introduction

Building robust, high-performance, large scale distributed systems is a challenging task given the complexity of designing, testing, and debugging such systems. Programming environments [27, 33, 35, 36, 38], execution logging and replay tools [20, 39], test case generation tools [13], and a variety of testbeds [4, 7, 44], emulations [1, 2, 52], and simulation platforms [3, 5, 6] were created to ease code development and testing.

Given the difficulty of finding bugs manually, several techniques have been applied to find them automatically. Model checking using static analysis [25, 34] 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. 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 systems implementation has been done with the use of symbolic execution, fault injection, and model checking. Symbolic execution [13] has been used to generate test cases that are capable of covering many control flow branches. This technique also suffers from a state-space explosion when applied to more complex implementations. Fault injection [24] 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 an implementation in a systematic manner. Finally, model checking using a systematic state-space exploration [25,26,41,56,57] has been used on system implementations to find bugs that violate specified invariants. To mitigate the effect of state-space explosion, the state exploration uses an iterative search, bounding some aspect of the execution. These heuristics do not prove bug absence, but rather help pinpoint where bugs do exist.

More recently, debugging techniques have been applied to automatically find attacks. 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 [16, 22, 37, 43, 54]. Dynamic taint analysis [37, 43, 54] 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 [11] 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, com-

promised 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 works in [50] which considers lying in headers of packets in two-party protocols, and [32] which assumes the user supplies a suspect line of code, indicating that it should not be executed many times. The method in [50] explores all states and does not scale for a distributed system. The method used in [32] has 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 work 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, requiring the user to provide only one or several metrics measuring system progress. 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 [27] toolkit. 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 the 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 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 five additional systems having different application goals and designs: the Vivaldi [19] virtual coordinate system, the Chord lookup service and distributed hash table (DHT) [51], and two multicast systems: ESM [17] and Scribe [49]. Gatling found a total of 41 performance attacks across the systems tested, 17 attacks based on message lying actions and 24 attacks based on message delivery actions. Finding each attack took a few minutes to a few hours.

Roadmap. Sections 2 and 3 describe the design and implementation of Gatling. Section 4 provides an example of how to use Gatling to find attacks in a well-known distributed file sharing system, BulletPrime [28]. Section 5 presents results on using our tool on five representative distributed systems: Vivaldi [19], Chord, DHT [51], ESM [17], and Scribe [49]. Section 6 presents related work and Section 7 concludes our paper.

2 Gatling Design

The design goal of Gatling is to automatically find insider attacks in distributed systems. 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. Such attacks require that the system has a performance metric, that when evaluated gives an indication of the progress it has in completing its goals. For example, for an overlay multicast system throughput is an indication of the performance of the system. Specifically, we define:

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

Next, we describe the system model we consider, identify malicious actions which represent building blocks for an attack, and describe our algorithm that searches and discovers attacks consisting of multiple malicious actions.

2.1 Design Overview

Model checking event-driven state machines. Many distributed systems [27, 29–31, 36, 40, 46–48, 55] are designed following event-based state machines that communicate through messages. Also several other systems use RPCs [35, 51], continuations [33], or data flow [38], which are *compatible* with this approach. 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*.

A well-known approach to find bugs in distributed systems is to use model checking 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 distributed nodes.

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. 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).

Several prior model checker designs [21, 26, 41, 42] have explored the capabilities of event-driven state machines to find bugs in systems implementations. Each of these designs provide mechanisms to explore complex interactions between nodes which would normally be infeasible for a user to exhaustively explore. However, 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 only in an execution with benign participants.

Our approach. 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, a 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.

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 find in a practical way such attacks. The resulting architecture is illustrated in Fig. 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 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.

2.2 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.
- *Delaying*: A malicious node does not immediately send a message and injects a delay.

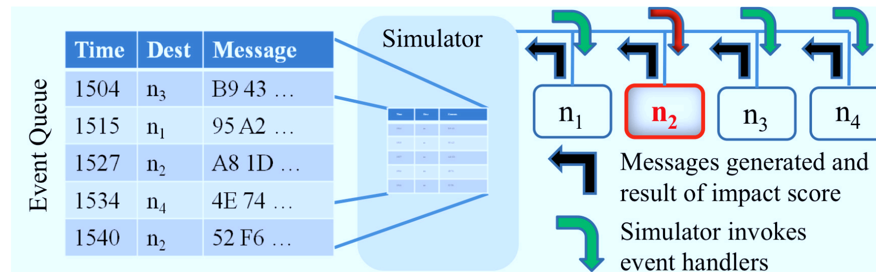


Figure 1. Gatling simulator model

- *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.

- *Duplicating*: A malicious node sends a message twice instead of sending only one copy, applying delay to the second copy. 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.

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. Gatling makes data-type-specific changes to message contents by being 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.

- *Zero*: For signed types, a malicious node can additionally change the value of the field to be the number 0.

- *Scaling*: A malicious node could increase or decrease the numeric value by a percentage.

- *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.

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 of 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.

2.3 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.

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 it has found the maximally effective single-behavior attack, it determines the contribution of each malicious action of the attack, and then can be repeated to find additional attacks.

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. 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 2.2) 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.

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 suc-

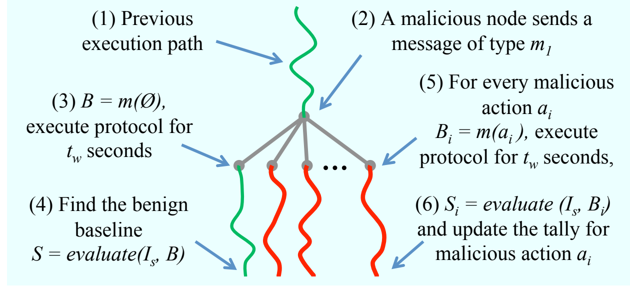


Figure 2. Greedy action selection procedure for one instance of sending message type m_1

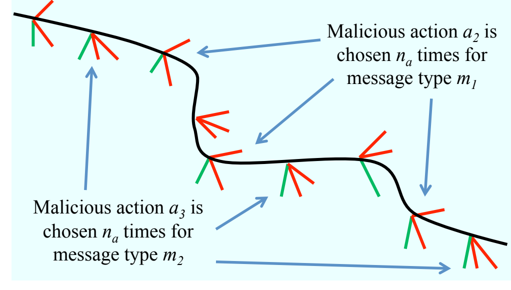


Figure 3. Greedy procedure applied for several instances of message types m_1 and m_2

cessful in finding single-behavior, 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 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 impact is greater than Δ. 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

variables:
  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
while $IneffectiveTally < HaltingThreshold$ **do**
  Continue simulating system until malicious node sends a message m;
  $msgType \leftarrow typeof(m)$;
  $MostEffectiveAction \leftarrow Attack[msgType]$;
  if $MostEffectiveAction = \emptyset$ **then**
    Find behavior $MostEffectiveAction \in \{\emptyset, A(msgType)\}$ according to selection procedure (Fig. 2)
    if $MostEffectiveAction \neq \emptyset$ **then**
      $AttackTally[msgType][MostEffectiveAction]{+}{+}$;
      $IneffectiveTally \leftarrow 0$;
      if $AttackTally[msgType][MostEffectiveAction] = n_a$ **then**
        $Attack[msgType] \leftarrow MostEffectiveAction$;
      end
    else
      $IneffectiveTally{+}{+}$;
    end
  end
  execute behavior $m(MostEffectiveAction)$;
end
$AttackAndContribution, \delta$ = computeActionContribution($Attack$);
if $\delta > \Delta$ **then**
  output $AttackAndContribution$;
  Repeat, ignoring prior Attack actions
end

Algorithm 1: Attack discovery algorithm

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 algorithm in details in Algorithm 1.

Impact score and parameter selection. The user must specify an impact score. As stated, the impact score must be able to be executed 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 two parameters, t_w and n_a. 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.

3 Implementation

We created a concrete implementation of Gatling for the Mace [27] 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, but just by defining specific lying actions for different types.

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.

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. 4 shows the Mace+Gatling architectural design 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 that layer defines. 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 applica-

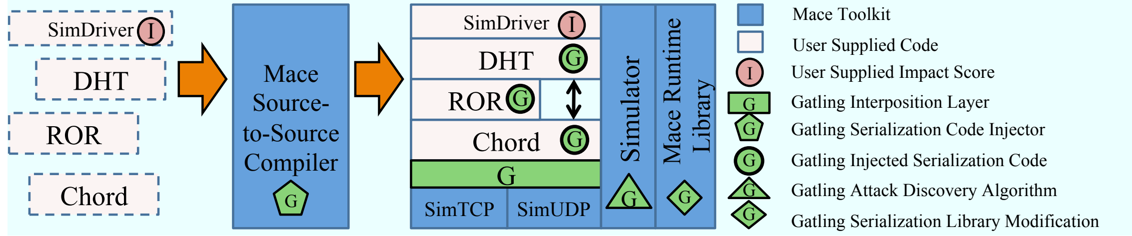


Figure 4. Gatling implementation for one node: DHT example

tion in lieu of *SimDriver*.

4 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 [28, 31] 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.

BulletPrime is a file distribution system similar to BitTorrent [18]. 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. 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 [30] 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.

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. 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.

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

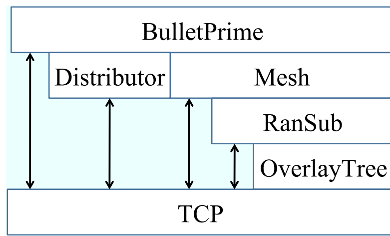


Figure 5. BulletPrime design

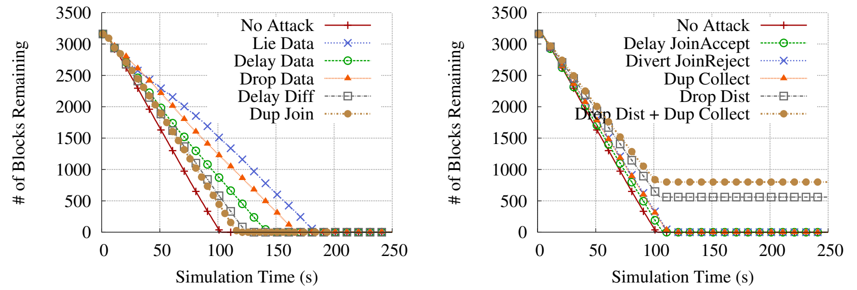


Figure 6. Remaining blocks for the attacks found on BulletPrime

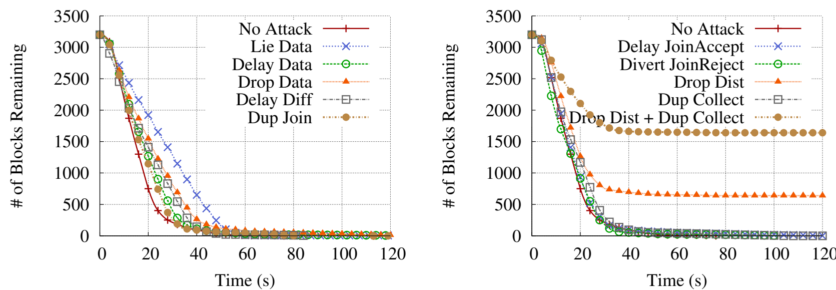


Figure 7. Remaining blocks for the attacks found on BulletPrime on PlanetLab

further Gatling simulations.

Fig. 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. 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. 7.

5 Results

We further validate the Gatling design by applying it to five systems with different application goals and designs. Specifically, we evaluate the Vivaldi [19] virtual coordinate system, the Chord lookup service and distributed hash table (DHT) [51], and two multicast systems: ESM [17] and Scribe [49]. Chord, DHT, and Scribe were previously im-

plemented for Mace; we implemented Vivaldi and ESM according to published papers. We set the number of malicious nodes to be 20% and we select malicious nodes randomly.

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 41 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 1, we summarize all the attacks.

5.1 Vivaldi

System description. Vivaldi [19] 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* [19] which describes how well the system predicts the actual RTT between nodes. 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.

Experimental setup. We simulated 400 nodes and randomly assign RTT values for each node from the KING data set [23] 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. 8 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). As a baseline we also present Vivaldi when there are no attacks, which we find converges 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 [58]. First, known as inflation is a lying attack where malicious nodes lie about their coordinates, providing larger than normal values from the spanning set without causing overflow. Second, known as deflation attack, occurs when where malicious nodes drop outgoing probes, thereby never updating their own coordinates. 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.

5.2 Chord

System description. Chord [51] 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. i then contacts these nodes and tells them to update their information. Later, a stabilization procedure will update global information to make sure i is known by others 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. 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. 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.

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. 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

System	Metric Used	Attack Name	Attack Description	Known Attack
BulletPrime	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	
		Inflation	Lie about coordinates, setting them to large values	[58]
		Oscillation	Lie about coordinates, setting them to random values	[58]
		Delay	Delay probe reply messages 2s	[58]
		Deflation	Do not initiate request (Drop probes)	[58]
Chord	Number of Reachable Nodes	Drop Find Pred	Drop query to find predecessor	[45]
		Drop Get Pred	Drop query to get predecessor and successor	[45]
		Drop Get Pred Reply	Drop the answer to find predecssor	[14]
		Lie Find Pred	Lie about key that is in query while forwarding queries	[14]
		Lie Predecessor	Lie about predecessor in response while forwarding	[14]
		Lie Successor	Lie about successor candidates in response while forwarding	[14]
DHT	Lookup Latency	Drop Msg	Drop recursive route messages	[14]
		Delay Msg	Delay recursive route messages	
		Dup Msg	Delay recursive route messages and divert second message	[14]
		Lie Msg Src	Lie about the source of recursive route messages	[14]
		Lie Msg Dest	Lie about the destination of recursive route messages	[14]
		Lie SetKeyRange	Lie about what keys are stored	
		Lie Reply Key	Lie about the key in get reply messages	[14]
		Lie Reply Found	Lie about finding the value in get reply messages	[14]
		Lie Reply Get	Lie about the request wanting the value in get reply messages	[14]
ESM	Throughput	Drop Data	Drop data messages	[53]
		Dup Parent	Duplicate parent reply messages, drop data	
		Lie Latency	Lie about measured latency, duplicate probe messages, drop data	[53]
		Lie Bandwidth	Lie about received bandwidth, duplicate probe messages, drop data	[53]
	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	
		Lie GroupId Join	Lie about the group identifier in a join message	

Table 1. Attacks found using Gatling: 41 attacks in total (17 lie, 12 drop, 6 delay, 5 duplicate, 1 divert)

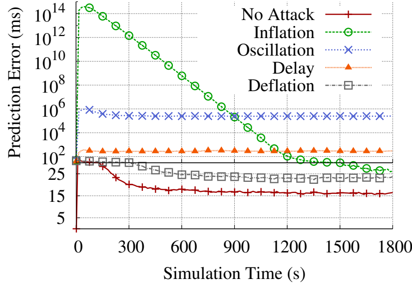
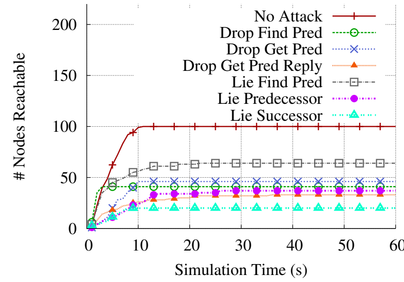
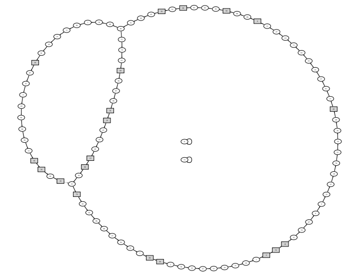


Figure 8. Prediction error for the attacks found on Vivaldi



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



(b) *Chord ring under Lie Predecessor attack*

Figure 9. Attack impact on Chord

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. 9(a). The effect of *Lie Predecessor* on the ring can be seen visually in Fig. 9(b), where some nodes failed to join, and others are confused about their relationships to adjacent nodes.

5.3 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 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 [51], structured as the example described in Figure 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 one 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. 10. 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*).

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.

5.4 ESM

System description. ESM [17] 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 [17]: *throughput* and *latency*. Throughput as described in [17], is the amount of data received over the last 5 sec, so the impact score is 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. 11(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. First 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 they are accepted as a child, sending the duplicate message to a random node. With this amplification, the throughput is usually 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 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. 11(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.

5.5 Scribe

System description. Scribe [49] 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 [48], 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, except that nodes whose out-degree is too high will push children down in the tree.

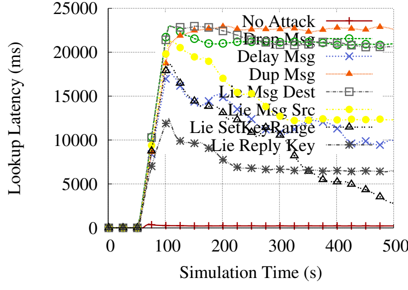
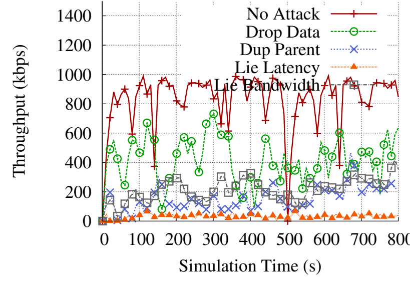
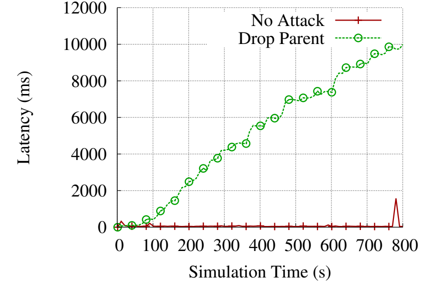


Figure 10. Lookup latency for the attacks found on DHT

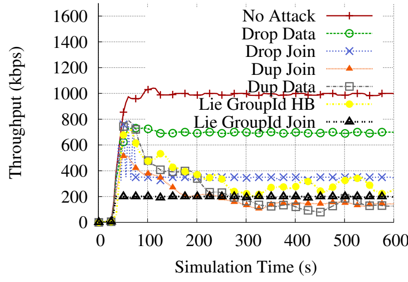


(a) Throughput impact score

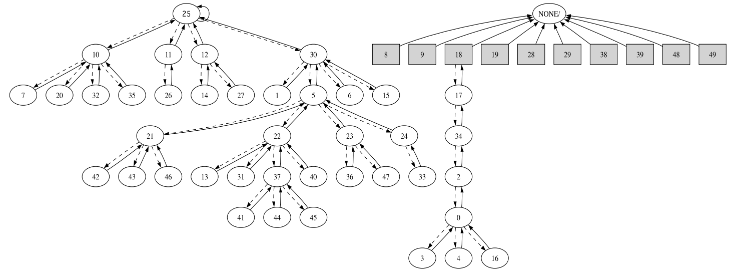


(b) Latency impact score

Figure 11. Throughtput and latency for the attacks found on ESM



(a) Throughput for the attacks found on Scribe



(b) Scribe tree under Lie GroupId Join attack

Figure 12. Attack impact on Scribe

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. 12(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. 12(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.

6 Related Work

Automated debugging techniques, such as model checking, have been in use for many years. Most similar to our work is CrystalBall [56], 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.

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 have also been explored in the context of limited models and environments [9, 10, 12]. These debugging techniques include static and dynamic analysis [16, 22, 32, 37, 43, 54], modelchecking [9, 10, 12], and fault injection [8, 11, 50]. 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.* [11]. They employ a fault injection technique on PBFT [15] to find combinations of MAC corruptions that cause nodes to crash. Our work is more general, as Gatling does not focus on finding all possible inputs that cause a single kind of vulnerability, but rather searches on basic malicious actions to find new attacks.

Stanojevic *et al.* [50] develop a fault injection 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.* [32] 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 causes 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. Gatling, on the other hand, utilizes 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.

7 Conclusion

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 presented Gatling, a 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 six distributed systems that have a diverse set of system goals. We were able to discover 41 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.

References

[1] Cyber-DEfense Technology Experimental Research laboratory Testbed. http://www.isi.edu/deter/.

[2] Emulab - Network Emulation. http://www.emulab.net/.

[3] Georgia Tech Network Simulator. http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS/.

[4] Global Environment for Network Innovation. http://www.geni.net.

[5] Network Simulator 3. http://www.nsnam.org/.

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

[7] Resilient Overlay Networks. http://nms.csail.mit.edu/ron/.

[8] J. Antunes, N. Neves, M. Correia, P. Verissimo, and R. Neves. Vulnerability Discovery with Attack Injection. *IEEE Transactions on Software Engineering*, 36:357–370, 2010.

[9] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P. Hem, O. Kouchnarenko, J. Mantovani, S. Mdersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Vigan, and L. Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Proceedings of Computer Aided Verification*, 2005.

[10] A. Armando and L. Compagna. SAT-based model-checking for security protocols analysis. *International Journal of Information Security*, 7:3–32, January 2008.

[11] R. Banabic, G. Candea, and R. Guerraoui. Automated Vulnerability Discovery in Distributed Systems. In *Proceedings of HotDep*, 2011.

[12] B. Blanchet. From Secrecy to Authenticity in Security Protocols. In *Proceedings of International Static Analysis Symposium*. Springer, 2002.

[13] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI*, 2008.

[14] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of OSDI*, 2002.

[15] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of OSDI*, 1999.

[16] C. Y. Cho, D. Babi, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of USENIX Security*, 2011.

[17] Y.-H. Chu, A. Ganjam, T. S. E. Ng, S. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang. Early Experience with an Internet Broadcast System Based on Overlay Multicast. In *Proceedings of USENIX ATC*, 2004.

[18] B. Cohen. Incentives build robustness in BitTorrent. In *Proceedings of P2P Economics*, 2003.

[19] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a decentralized network coordinate system. In *Proceedings of SIGCOMM*, 2004.

[20] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *Proceedings of NSDI*, 2007.

[21] P. Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of POPL*, 1997.

[22] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS*, 2008.

[23] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. In *Proceedings of ACM SIGCOMM-IMW*, 2002.

[24] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: a framework for cloud recovery testing. In *Proceedings of NSDI*, 2011.

[25] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, May 1997.

[26] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Detecting Liveness Bugs in Systems Code. In *Proceedings of NSDI*, 2007.

[27] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *Proceedings of PLDI*, 2007.

[28] D. Kostić, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of USENIX ATC*, 2005.

[29] D. Kostic, A. Rodriguez, J. Albrecht, , and A. Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *Proceedings of SOSP*, 2003.

[30] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *Proceedings of USENIX-USITS*, 2003.

[31] D. Kostić, A. C. Snoeren, A. Vahdat, R. Braud, C. Killian, J. W. Anderson, J. Albrecht, A. Rodriguez, and E. Vandekieft. High-bandwidth data dissemination for large-scale distributed systems. *ACM Transactions on Computer Systems*, 26(1):1–61, 2008.

[32] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding Protocol Manipulation Attacks. In *Proceedings of SIGCOMM*, 2011.

[33] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proceedings of USENIX ATC*, 2007.

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

[35] L. Leonini, E. Rivière, and P. Felber. SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proceedings of NSDI*, 2009.

[36] S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo. WiDS: an Integrated Toolkit for Distributed Systems Deveopment. In *Proceedigs of HotOS*, 2005.

[37] Z. Lin, X. Zhang, and D. Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *Proceedings of DSN*, 2008.

[38] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proceedings of SOSP*, Brighton, United Kingdom, October 2005.

[39] X. Lui, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs In Distributed Systems. In *Proceedings of NSDI*, Cambridge, Massachusetts, April 2007.

[40] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[41] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of OSDI*, 2002.

[42] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of OSDI*, 2008.

[43] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of NDSS*, 2005.

[44] PlanetLab. http://www.planetlab.org.

[45] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*. ACM, 2001.

[46] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of USENIX ATC*, 2004.

[47] A. Rodriguez, D. Kostić, Dejan, and A. Vahdat. Scalability in Adaptive Multi-Metric Overlays. In *Proceedings of IEEE ICDCS*, 2004.

[48] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of IFIP/ACM Middleware*, 2001.

[49] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of NGC*, 2001.

[50] M. Stanojevic, R. Mahajan, T. Millstein, and M. Musuvathi. Can You Fool Me? Towards Automatically Checking Protocol Gullibility. In *Proceedings of HotNets*, 2008.

[51] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM*, 2001.

[52] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of OSDI*, 2002.

[53] A. Walters, D. Zage, and C. Nita-Rotaru. A Framework for Mitigating Attacks Against Measurement-Based Adaptation Mechanisms in Unstructured Multicast Overlay Networks. *IEEE/ACM Transactions on Networking*, 16:1434–1446, 2008.

[54] W. Wang, Y. Lei, D. Liu, D. Kung, C. Csallner, D. Zhang, R. Kacker, and R. Kuhn. A Combinatorial Approach to Detecting Buffer Overflow Vulnerabilities. In *Proceedings of DSN*, 2011.

[55] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture For Well-conditioned, Scalable Internet Services. In *Proceedings of SOSP*, 2001.

[56] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of NSDI*, 2009.

[57] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of NSDI*, 2009.

[58] D. J. Zage and C. Nita-Rotaru. On the accuracy of decentralized virtual coordinate systems in adversarial networks. In *Proceedings of CCS*, 2007.