Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations

Hyojeong Lee, Jeff Seibert, Endadul Hoque, Charles Killian, and Cristina Nita-Rotaru
The Department of Computer Science, Purdue University, West Lafayette, Indiana
Email: {hyojlee, jcseiber, mhoque, ckillian, crisn}@cs.purdue.edu

***Abstract*—Security and performance are critical goals for distributed systems. The increased design complexity, incomplete expertise of developers, and limited functionality of existing testing tools often result in bugs and vulnerabilities that prevent implementations from achieving their design goals in practice. Many of these bugs, vulnerabilities, and misconfigurations manifest after the code has already been deployed making the debugging process difficult and costly.**

In this paper, we present Turret, a platform for automatically finding performance attacks in unmodified implementations of distributed systems. Turret does not require the user to provide any information about vulnerabilities and runs the implementation in the same operating system setup as the deployment, with an emulated network. Turret uses a new attack finding algorithm and several optimizations that allow it to find attacks in a matter of minutes. We ran Turret on 5 different distributed system implementations specifically designed to tolerate insider attacks, and found 30 performance attacks, 24 of which were not previously reported to the best of our knowledge.

I. Introduction

Most distributed systems are designed to meet 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[1]. Many of these vulnerabilities manifest after the code has already been deployed, making the debugging process difficult and costly.

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.* Checking just the design is not sufficient because implementations: (1) include many optimizations and adaptive mechanisms not formally specified and proved, (2) consist of many subprotocols that have unexpected interactions and sometimes opposite goals, or (3) do not always fully follow the design. One class of attacks that are especially difficult to find are attacks that slow down the system, preventing it from achieving its potential performance given the network's condition, referred to as performance attacks.

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 [1]–[5] 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 [1]–[3], require the target implementation to be written in a specific language for a specific programming model [4], or require a particular target operating system and the implementation to be instrumented [1].

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 [6]–[8]. The work in [6] 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 [7] 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 a vulnerability of the system. This approach is fitted for corner cases, but does not attempt to systematically find attacks. Finally, our previous work, Gatling [8] does not require a priori knowledge about specific vulnerabilities in the system, however, it relies on the Mace [9] framework which requires the tested systems to be written in the Mace language.

In this paper, 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 [10]. We do not restrict libraries or operating systems because we want our solution to be applicable to already existing implementations

[1]For example, Amazon is estimated to earn $9,823 every five seconds.

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 of this paper 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 [8, 11] by reducing the time required to find an attack to minutes. Low time overhead is critical for platforms that run systems in realistic conditions, as 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 5 to 15 virtual machines. 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 [12], Steward [13], Zyzzyva [14], Prime [15], and Aardvark [16]) 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 the paper is organized as follows. We present the system and attack model in Section II and our design in Section III. We describe implementation issues in Section IV and the results we obtained running Turret on five systems in Section V. We overview related work in Section VI. Finally, we present our conclusion in Section VII.

II. 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.

A. Distributed System Model

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*. This model is representative for many distributed systems [9, 17]–[25] that are designed following event-based state machines that communicate through messages. Also several other systems using RPCs [26, 27], continuations [28], or data flow [29] are *compatible* with the approach where participants communicate via messages instead of shared memory.

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. Many distributed systems make their performance metrics clearly measurable.

B. 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 performance metric that gives an indication of the progress the system has in completing its goals. We define:

Definition 1 - Performance attack: *We define a performance attack as a set of actions that deviate from the protocol, taken by a group of malicious nodes and resulting in a performance that is worse by some Δ than in benign scenarios.*

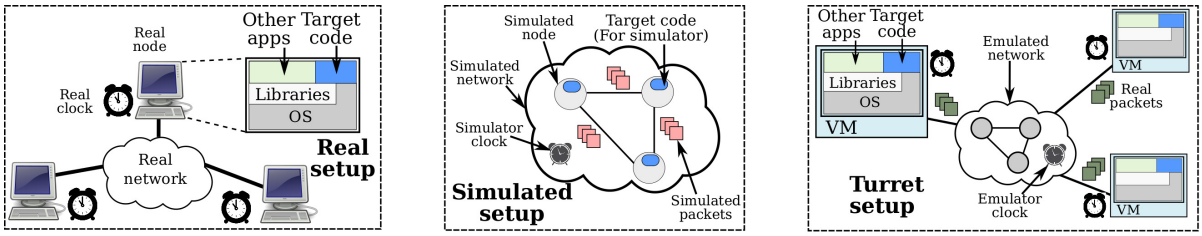
We focus on attacks specific to message-passing distributed systems, but general enough for any of such systems. The attacker does not know the state machine of the system, but it knows the type and the format of the messages and uses this information to have a global impact on the system performance. 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 the 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) 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

Fig. 1. Approaches for finding performance attacks in distributed systems implementations.

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 which 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.

III. 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.

A. Design Overview

Fig. 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. 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 [8].

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. 1(c)). In an emulated approach, the 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 automatically create malicious message actions 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.

B. 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*.

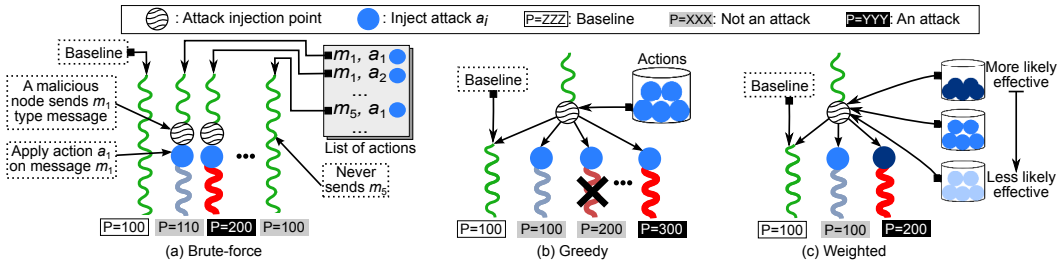


Fig. 2. Attack finding algorithms

The simplest approach is a *brute-force search algorithm* as the one showed in Fig.2 (a). The algorithm relies on a list of all possible attack scenarios (malicious actions for each message type). [2] It first obtains a baseline, then for each attack scenario in the list, 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* like the one we previously used in a simulated approach [8] and showed in Fig.2 (b). When the algorithm encounters an attack injection point, it branches the execution and obtains a baseline as well as 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 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 to be added, overhead amplified also by the fact that experiments run in real time. We discuss execution branching support in Section III-C.

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.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.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 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.

C. 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 application level, which requires modifications of the application, library or operating system, or at the hypervisor level. 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 [30, 31]. A snapshot of a distributed system includes the local states of all individual nodes and the state of the messages in transit [32, 33]. 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

[2]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.

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 [32], 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 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.

D. 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.

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

A. Overview

We choose KVM as our virtualization technique and the NS3 emulator for the network emulation. Fig. 3 depicts Turret

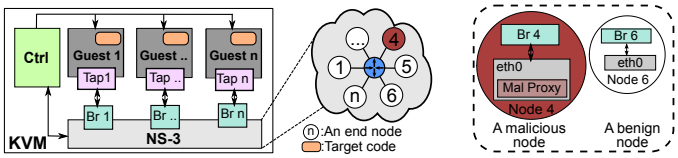


Fig. 3. The Turret Platform, illustrated for n nodes with node 4 designated as malicious

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 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.

B. Malicious Proxy

The malicious proxy has two components: one that intercepts the packets and one that processes them according to a malicious action. To ensure that all traffic is available to the malicious proxy while not restricting the emulated network topology, the proxy intercepts packets along the ingress path inside NS3. Port numbers associated with malicious nodes and specified in the NS3 configuration file ensure that only packets corresponding to malicious nodes are intercepted. For TCP, we modified the emulated network card to intercept packets and forward them to the application layer in the NS3 end node where the malicious action is applied. For UDP, the malicious proxy intercepts packets in the NS3 Bridge module, it strips off the UDP headers and passes the data on to the malicious proxy component that will perform the malicious action without incurring the overhead of passing the packet up multiple layers.

As malicious actions are based on message types, the malicious proxy needs to determine the type of a message. 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.

C. 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 [34, 35]. 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 [35] 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 create also 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 a 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.

V. Experimental Results

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

We used an 8 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.

A. Performance

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, 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 the overhead was similar to the benign case.

Load/save for VM and NS3 snapshots. We show the reduced overhead of our shared page optimization in Table II. 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

TABLE I. 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	[36]
	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	[16]
	Dup *Prepare* 50	Duplicating *Prepare* messages 50 times degrades the performance	[16]
	Dup *Commit* 50	Duplicating *Commit* messages 50 times degrades the performance	[16]
	Dup *Status* 50	Duplicating *Status* messages 50 times degrades the performance	[16]
	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	[36]
	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	
	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	

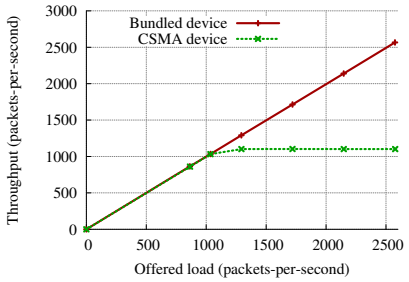


Fig. 4. Throughput for bundled and CSMA devices

TABLE II. 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

shared snapshots, we reduce the 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 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 III, weighted greedy algorithm found identical attacks 84.5 – 99.1% faster than the greedy algorithm.

TABLE III. 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

B. Case study: PBFT

We demonstrate Turret on the implementation of the most well known byzantine-resilient state machine replication system, PBFT [12]. 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 participate in several protocols.

Normal Case: 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.

View Change: This protocol evicts a primary when it is faulty, thus allowing the system 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 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 uses these to bring its state up to date, 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 [12] available at [37]. The code uses UDP as its communication protocol.

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 [12].

Attacks Limiting Progress.
Delay Pre-Prepare *Attack*: This attack has been found previously by Amir et al. [36] 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. 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. 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 158.3 updates/sec to 1.08 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 4.95 updates/sec for the drop 50% attack, as seen in Fig. 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 131 updates/sec, as seen in Fig. 5(b).

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 37.9, 36.8, and 43.1 updates/sec, respectively. However, in case of *Status* messages, the throughput decreases to 126.3 updates/sec. We show these results in Fig. 5(c). 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.

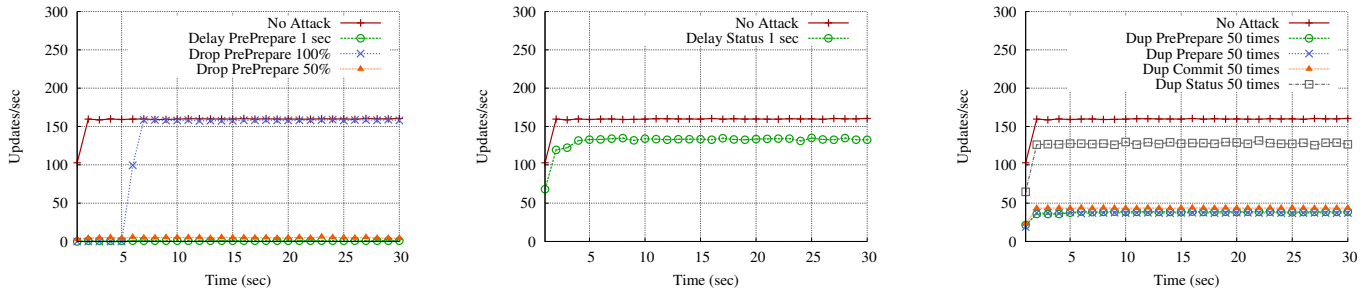
Attacks Causing Crashing.
These are attacks that crash nodes. We found that there are a few integer fields in *Pre-Prepare* and *Status*, that if any of them are replaced with negative values, all non-faulty replicas will experience a segmentation fault. Investigating deeper, these fields deal with sizes 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.

C. Results on Other Systems

In addition to PBFT, we used Turret on four other systems, Steward [13], Zyzzyva [14], Prime [15], and Aardvark [16]). We selected these systems because they are representative: they scale to wide area networks (Steward), they tolerate some performance attacks (Prime and Aardvark). Due to the lack of space, we only briefly describe the discovered attacks and we summarized all discovered attacks in Table I.

Steward. 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



(a) Attacks that limit progress (b) Delay attacks that cause DoS (c) Duplication attacks that cause DoS

Fig. 5. Updates per second for PBFT for the attacks found.

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

Zyzzyva. 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.

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

Aardvark We found a total of 4 attacks: 3 lying attacks and 1 delivery attack. All the lying attacks cause benign nodes to crash because message fields are not validated properly. Delaying *Status*, the message delivery attack, which is similar to the one discovered against PBFT, causes the system to slow down. However, Aardvark's flooding protection mechanism can mute the attack when the delay becomes too big.

D. 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 the testing platform for the programming assignments given in this course. The projects were designed based on Paxos [38], Byzantine Generals Problem [39], and Total Order Multicast [40]. 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.

VI. Related Work

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.* [41] 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.* [42] 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.

Fuzzing is another technique used to find vulnerabilities in implementations [43, 44]. While traditional fuzzing has been a black-box approach, new fuzzing techniques leverage code analysis. Notably, Sage [43] combines fuzzing with symbolic execution to extend code coverage(test wider). Dowser [44] 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 which is important for performance problems or distributed systems.

Banabic *et al.* [45] use a fault injection technique to discover vulnerabilities in distributed systems. They try to find all possible inputs that cause the system to reach a vulnerability. They apply their technique on PBFT [12] and find MAC corruptions that cause nodes to crash. Their work is different from ours that they require the user to configure the fault injector to execute attacks that a user expects and they can not lie based on message fields.

Kothari *et al.* [7] focus on automatically finding attacks that manipulate control flow through message modification. They employ static analysis to search values to modify and generate attack actions. They require a C language implementation and a user-provided location of a potentially vulnerable statement in the code, which may not be easy to obtain.

Stanojevic *et al.* [6] develop a technique that automatically searches for attacks in two-party protocols by using a brute force search on attack actions including packet dropping and modification of fields in the packet headers. Our previous work Gatling [8] automatically discovers performance attacks caused by insiders in distributed systems by deviating the protocol in a short time window and searches the biggest performance change using a greedy algorithm. Both [6] and [8] require the implementation to be written in Mace [9]. Turret is more general as it does not pose any limitation on the language or the environment of the target application.

VII. Conclusion

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 [12], Steward [13], Zyzzyva [14], Prime [15], and Aardvark [16]. 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.

Acknowledgement

This material is based in part upon work supported by the NSF under Grant Number CNS-1223834.

References

[1] 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 *NSDI*, 2009.

[2] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill, "CMC: A pragmatic approach to model checking real code," in *OSDI*, 2002.

[3] G. J. Holzmann, "The model checker spin," *IEEE TSE*, vol. 23, 1997.

[4] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Detecting liveness bugs in systems code," in *NSDI*, 2007.

[5] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak, "Crystalball: Predicting and preventing inconsistencies in deployed distributed systems," in *NSDI*, 2009.

[6] M. Stanojevic, R. Mahajan, T. Millstein, and M. Musuvathi, "Can you fool me? Towards automatically checking protocol gullibility," in *HotNets*, 2008.

[7] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," in *SIGCOMM*, 2011.

[8] H. Lee, J. Seibert, C. Killian, and C. Nita-Rotaru, "Gatling: Automatic attack discovery in large-scale distributed systems," in *NDSS*, 2012.

[9] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: language support for building distributed systems," in *PLDI*, 2007.

[10] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H. J. Chu, A. Terzis, and T. Herbert, "packetdrill: Scriptable network stack testing, from sockets to packets," in *ATC*, 2013.

[11] M. E. Hoque, H. Lee, R. Potharaju, C. E. Killian, and C. Nita-Rotaru, "Adversarial testing of wireless routing implementations," in *WiSec*, 2013.

[12] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, 1999.

[13] 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," *IEEE TDSC*, vol. 7, no. 1, 2010.

[14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *SOSP*, 2007.

[15] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE TDSC*, vol. 8, no. 4, 2011.

[16] A. Clement, E. Wong, L. Alvisi, and M. Dahlin, "Making byzantine fault tolerant systems tolerate byzantine faults," in *NSDI*, 2009.

[17] M. Welsh, D. E. Culler, and E. A. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *SOSP*, 2001.

[18] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat, "Using random subsets to build scalable network services," in *USITS*, 2003.

[19] A. Rodriguez, D. Kostić, Dejan, and A. Vahdat, "Scalability in adaptive multi-metric overlays," in *ICDCS '04*, 2004.

[20] D. Kostic, A. Rodriguez, J. Albrecht, , and A. Vahdat, "Bullet: High bandwidth data dissemination using an overlay mesh," in *SOSP*, 2003.

[21] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM Middleware*, 2001.

[22] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," 2004. [Online]. Available: citeseer.ist.psu.edu/rhea03handling.html

[23] 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 TOCS*, vol. 26, no. 1, 2008.

[24] S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo, "WiDS: an integrated toolkit for distributed systems deveopment," in *HOTOS*, 2005.

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

[26] 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 *NSDI*, 2009.

[27] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM*, 2001.

[28] M. Krohn, E. Kohler, and M. F. Kaashoek, "Events can make sense," in *ATC*, 2007.

[29] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *SOSP*, 2005.

[30] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," in *ATC*, 2006.

[31] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global comprehension for distributed replay." in *NSDI*, 2007.

[32] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM TOCS*, vol. 3, no. 1, 1985.

[33] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *Elsevier JPDC*, vol. 18, no. 4, 1993.

[34] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: harnessing memory redundancy in virtual machines," in *OSDI*, 2008.

[35] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *OLS*, 2009.

[36] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *DSN*, 2008.

[37] "PBFT," http://www.pmg.lcs.mit.edu/bft/.

[38] J. Kirsch and Y. Amir, "Paxos for system builders," *Dept. of CS, Johns Hopkins University, Tech. Rep*, 2008.

[39] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM TOPLAS*, vol. 4, no. 3, 1982.

[40] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM TOCS*, vol. 5, no. 1, 1987.

[41] P. Marinescu and G. Candea, "Efficient testing of recovery code using fault injection," *ACM TOCS*, December 2011.

[42] 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 *NSDI*, 2011.

[43] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[44] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *SEC*, 2013.

[45] R. Banabic, G. Candea, and R. Guerraoui, "Automated vulnerability discovery in distributed systems," in *HotDep*, 2011.