MAZE: A Secure Cloud Storage Service Using Moving Target Defense and Secure Shell Protocol (SSH) Tunneling

by

Vasco Xu

B.Phil. Computer Science, University of Pittsburgh, 2020

Submitted to the Faculty of
the Department of Computer Science and the Honors College in
partial fulfillment of the requirements for the degree of

Bachelor's of Philosophy

University of Pittsburgh

2020

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This thesis was presented

by

Vasco Xu

It was defended on

November 2, 2020

and approved by

Sherif Khattab, Lecturer, Department of Computer Science

Ehab Al-Shaer, Professor, Information Networking Institute and CyLab, CMU

Amy Babay, Assistant Professor, Department of Computer Science

Daniel Mossé, Professor, Department of Computer Science

Thesis Advisor: Sherif Khattab, Lecturer, Department of Computer Science

MAZE: A Secure Cloud Storage Service Using Moving Target Defense and Secure Shell Protocol (SSH) Tunneling

Vasco Xu, B.Phil

University of Pittsburgh, 2020

Cloud storage services have emerged as a popular destination for businesses and individuals to securely store documents due in part to being virtually accessible anywhere, anytime. However, cloud storage systems are static attack targets enabling attackers to thoroughly study the system without fear that their conclusions about the system would be rendered inaccurate. As such, computer security researchers began exploring techniques, known as Moving Target Defense (MTD), to turn distributed systems into moving targets. Whereas traditional defense mechanisms attempt to identify and cover system vulnerabilities, the underlying philosophy of MTD is that it is impossible to build perfectly secure systems. Instead, MTD techniques attempt to constantly change the attack surface in order to increase the cost (in terms of time and resources) and difficulty of executing successful attacks, in the first place. Current research in MTD, however, is lacking in implementations of MTD techniques on real systems (rather than just simulations).

This work presents MAZE, a secure cloud storage system in which the files to be protected (e.g., security keys, account numbers or passwords) are split into pieces and pseudo-randomly dispersed within a large, continuously-changing maze of computers. Hopping from one computer to another within MAZE is only possible by following timely created doors, which are implemented using Secure Shell Protocol (SSH) tunnels. At any computer, there can be many open doors, each leading to a different computer. In order to retrieve a file, the user has to follow a schedule that is provided by the MAZE service to authorized users only. The schedule informs the client of which doors to traverse through to retrieve all the pieces of the file. In addition, computers within MAZE have two refresh periods: the first restarts the computer and reloads the system software from a clean copy in order to thwart potentially ongoing attacks, and the second modifies the file pieces to become incompatible with the file pieces before modification. In order for attackers to successfully retrieve a file, they must retrieve all file pieces within the second refresh period. We implemented MAZE and performed a series of experiments that demonstrated the potential of an MTD-based cloud storage system in protecting against attackers while providing reasonable response time.

Table of Contents

List of Tables

List of Figures

Preface

This thesis, unfortunately, marks the end of my undergraduate career. It has not been the smoothest of journeys, but I was fortunate enough to meet an incredible group of friends and Professors along the way who have helped me stay on track.

To Ge, Brandon, Shay, Caitlin, Abbas, Stan and Andrew, the trips we shared together are some of the happiest moments of life and I would not trade them for anything else. Thank You.

To Dr. Khattab, this thesis would not have been possible without you. I faced so many roadblocks but you always helped me push through them. You have taught me, inspired me, and motivated me to become a better computer scientist and person. You always managed to schedule time to review my work and discuss research ideas, and have shown me nothing but kindness. Thank You.

To Dr. Mossé, thank you for all the support that you have given me as a researcher. Your combination of humor and passion for research has inspired me to work harder everyday. Thank you for all the help in applying to graduate school, it means so much to me. Muito Obrigado.

To Dr. Babay, thank you for inviting me to attend your class (some of what I learned is even used in this thesis!). You have helped greatly in developing my research interests and have always been so understanding and nice. Thank You.

If I end up pursuing a career in academia, I hope to one day treat my students with the same kindness, patience and love that you have all shown me. From the bottom of my heart, Thank You.

Finally, to my parents and brother, your unconditional support means everything to me. Thank You.

1.0 Introduction

In light of the recent COVID-19 pandemic, people have started exploring online alternatives to what used to be done in-person. Online collaboration has increased, resulting in numerous documents (e.g., recorded Zoom meetings, medical data, account numbers, account passwords) to be stored in cloud storage platforms (e.g., Box, OneDrive, Google Drive, Dropbox). As a result, public cloud storage providers have become high-profile targets for attackers during and after the pandemic. For example, as recent as January 2020, a company called Data Deposit Box, which is a "top rated secure cloud backup storage service for small businesses" was victim of a data breach which exposed personal information of about 148 customers [6]. The breach was due to a vulnerability in the system configuration, which was immediately identified and fixed within a day. Gartner Inc., a leading research and advisory company, estimates that 95% of cloud breaches are due to human errors such as configuration mistakes [9]. Configuration mistakes are caused not just by the complexity of the system but also by sheer laziness. This is worrisome, as cloud storages are becoming increasingly popular for both business and personal use. Therefore, it is easy to see the importance of cloud security particularly in cloud storage systems.

Independent of the pandemic, cloud storage services have become a desired destination for people and businesses to store important documents online [8]. Deloitte reports that 58% percent of a total of 500 IT leaders moved to the cloud because of security and data protection [16]. One of the main reasons to use a cloud storage service is because they can be accessed anywhere, anytime. The downside of cloud storage systems is that they are static attack targets. In other words, the software underlying the cloud storage system is, in most cases, the same in all the machines that run it since most machines serve the same purpose of storing data while ensuring consistency, availability, and security. Moreover, defense applications tend to be static; defense mechanisms either attempt to discover and fix all possible vulnerabilities (e.g., suspicious inputs), by simulating possible attacks, or try to mitigate an attack once it has already happened. Clever attackers can analyze local copies of the application and exploit weaknesses therein. Since the same software is run on multiple machines, a vulnerability in one copy of the software is likely to also exist in all other copies. Therefore, exploiting one machine, in a static system, can possibly lead to many other machines being exploited. In public cloud storages, attackers have the luxury of time to prepare for an attack by trying to find vulnerabilities within the system. On the other hand, a defender must be prepared for any possible attack that comes their way, even before those attacks are launched. In a business setting, one data breach could be enough to lose all customers. Although many tools and techniques have already been proposed to find and fix vulnerabilities within cloud systems [25, 46, 48], these tools and techniques are not enough to protect against cyber-attacks. This limitation exists because (1) as cloud storage systems become more complex, it becomes harder to find every possible vulnerability, (2) as cloud storage systems become more intelligent so do attackers and (3) attackers have the advantage of time to understand and explore the system to its fullest because the system is, in most cases, open to everyone.

In this thesis we propose MAZE, a secure cloud storage system, which employs a technique called Moving Target Defense (MTD) to transform a static cloud storage system into a moving target. In MAZE, important files (encrypted or unencrypted) to be protected are split into pieces and pseudo-randomly dispersed within a large, continuously-changing maze of computers. Hopping from one computer to another within MAZE is only possible by following timely created doors, which are implemented using Secure Shell Protocol (SSH) tunnels. These doors open and close at pseudo-randomly generated times. At any computer, there can be many open doors, each leading to a different computer. In order to retrieve a file, the user has to follow a schedule, provided by the MAZE service to authorized users only. The schedule informs the user of which doors to traverse through in order to retrieve all the pieces of the file. At a periodic time epoch, nodes are restarted and system software is copied over from a secure read-only medium. A small subset of computers in MAZE are reachable from the Internet (known as the gateways); the gateways are constantly changing adding a further layer of difficulty for attackers. *Without the schedule, an attacker would blindly follow doors and end up getting lost in the maze.*

1.1 System Model

The cloud storage system that we consider in this work consists of nodes and files. A *node* is a physical or virtual machine. There are three types of nodes: file nodes, gateway nodes, and client nodes. A *file node* is used to store files and does not accept connections from the Internet. A *gateway node* is a special type of node, within the network, that is reachable from the Internet (i.e., accepts connections from the Internet). When a user wants to access the cloud storage system they must first contact a gateway node; they are the entry points to the system. A *client node* is also a special type of node that is not part of the cloud storage system itself. It is the machine that users utilize to access the cloud storage. Each client node has a public IP address and can accept incoming network connections on that address (e.g., incoming connections from file nodes). We consider client authentication to be assumed and orthogonal to this work [18, 21, 47]. We call a collection of nodes a *cluster*. A cluster can be either owned by a company or an individual who has access to a cluster personally or through a cloud service provider, such as Amazon AWS or Microsoft Azure. Lastly, *files* are documents (e.g., Zoom recordings, photos, account numbers) that a user wants to store within the cloud storage system. Files may or may not be encrypted for further protection and must be able to be split into file pieces [28], each containing different parts of the file.

The cloud storage system configuration can be thought of as a graph of nodes with edges connecting them as seen in Figure 1. The cloud represents the public network that the nodes are under. Each vertex (circle) in the graph represents a file node (non-shaded), a gateway (in green) or a client node (in orange). We note that client nodes access MAZE through a gateway node, since gateways are the entry points to the system. The dashed line stemming from the client node to the gateway node represents the secure two-way connection between those two nodes. In Figure 1, we only display one client node, but the cloud storage system can support many client nodes. The dashed arrow stemming from file nodes to the client node represents a secure one-way connection between the two nodes for transferring of files. Each edge (non-dashed line) inside the graph represents a connection (i.e., networking link) between two vertices. Networking links are predefined by the system administrator and do not change very often. Networking links could either be physical links or virtual links. A physical link exists between nodes that are directly connected to each other through physical wires. A virtual link, on the other hand, exists between nodes that are connected to each other in the network graph/topology, but not necessarily directly connected in the underlying network of physical links. We assume that the network is reliable and permanent. The file pieces are shown as blue squares within the file nodes.

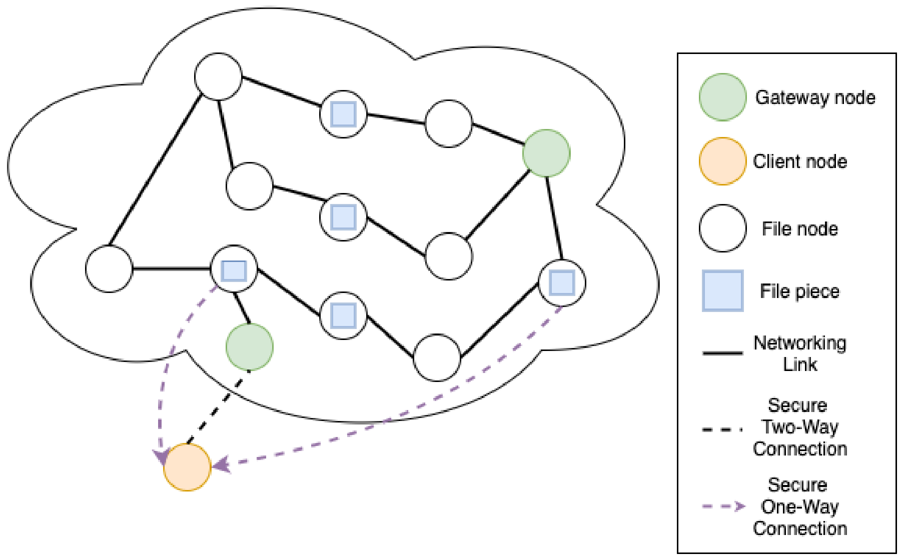

Figure 1: The MAZE cloud storage system consists of gateways nodes, file nodes, file pieces and client nodes. *Files nodes* are used to store file pieces and are not directly reachable from the Internet. *Gateway nodes* are directly reachable from the Internet and are therefore, the entry-points to the system. The *client node* is the machine of the user that wants to access the MAZE system, which must have a public IP address.

1.2 Threat Model and Assumptions

We assume that the goal of an attacker is to retrieve a *specific* file inside the cloud storage system as opposed to wanting to retrieve *every* possible file or *any* file. At any point in time, an attacker could have multiple agents (i.e., a process that runs a script to perform an automated task on a single node). An attack is considered successful when an attacker retrieves all file pieces before a node refreshes. On a node refresh, attacker agents are evicted from the system, and file pieces are modified in such a way that they become incompatible (i.e., cannot be glued back together) with file pieces before modification. The modification of file pieces can be achieved through a technique known as *proactive secret sharing* [31, 50, 42], which is orthogonal to this work. Node refreshes are discussed in detail in Chapter 3.

Our attack model considers attackers who are able to retrieve any file by taking control of a growing subset of nodes. In other words, attackers can take control of nodes incrementally, one at a time, but not all of them at once. Because we model the MAZE cloud storage system configuration as a graph, an attacker can perform a breadth-first search or a depth-first search to visit all the nodes. Also, we assume that an attacker does not know how many pieces a file was split into and which nodes those pieces are stored in and thus would have to perform a full traversal in order to retrieve all possible file pieces. Attackers can recognize file pieces, either from their names or using content analysis, but do not know how many of these pieces there are.

We consider attackers to be unauthorized users and thus do not have direct access to the MAZE system. We assume that attackers cannot break authentication or steal client credentials or access keys. Therefore, when we say that an attacker compromises a node, we assume they did so without access to any of the client's credentials. We define compromising a node as the act of gaining control and access to a node by other unauthorized means, such as exploiting a vulnerability in the system configuration. Since only the gateway node is accepting connections from the Internet, attacker agents must initiate their attack from a gateway node. In addition, we assume that attacker agents cannot escalate their privileges to administrator or root privileges.

Fault tolerance is important in distributed systems, such as cloud storage systems, as data is expected to be accessible anywhere, anytime. Data stored in a cloud storage system should be replicated to ensure that if one node is down, other nodes should be able to serve the desired data. In this work, fault tolerance is assumed and considered to be orthogonal to this work [34, 24, 27]. We do not consider network compromises, that is, attackers cannot open and close networking links between nodes. In other words, attackers cannot control (add, edit, or remove) edges of the graph nor add or remove nodes from the system.

1.3 Problem and Thesis Statements

The key problem addressed in this work is how to build a secure cloud storage under certain constraints from the perspective of the defender while providing reasonable performance. From a defense perspective, we want to build a system such that, given an attacker with a certain number of malicious agents, and a specific time limit (i.e., before a node refresh) to retrieve files, the probability of retrieving all file pieces within the time limit is very low. From a performance perspective, given a certain number of nodes, users, and files pieces, we aim to build a cloud storage system that provides adequate response time in addition to security.

This thesis explores the possibility of emulating the structure and difficulty of a physical maze to secure computer systems while maintaining reasonable response time. We achieve the aforementioned by using Moving Target Defense and tunneling, which dynamically reconfigures the system to confuse attackers, similar to how some paths in a physical maze are meant to confuse the people within it. Overall, this thesis presents MAZE, a secure cloud storage system using Moving Target Defense and tunneling, details its implementation, and evaluates the system in terms of performance and security.

1.4 Outline

This chapter introduced the importance of secure cloud storage services and how this work plans to improve upon it, along with the assumptions that we made while designing and building the system.

Chapter 2 provides background information of the methods and techniques used in building the proposed system and a comprehensive review of literature related to this work.

Chapter 3 presents MAZE, a secure cloud storage using MTD and SSH Tunneling, detailing two design approaches and tradeoffs between performance and security.

Chapter 4 details the implementation of MAZE, the experiment setup, the testbed implementation, and evaluates MAZE in terms of performance and security.

Chapter 5 discusses limitations of the MAZE cloud storage system and proposes possible solutions and future work.

1.5 Definitions

In this thesis, there are a few keywords that are worth noting because they are used to describe concepts prevalent throughout this work. The keywords and their definitions are provided below:

- **Node:** A machine (or virtual-machine) that is part of the MAZE cloud storage system.
- **Tunneling:** Secure movement of data over an unsecure network. Tunneling typically involves secure communication between two private networks to be made over a public network, such as the Internet. All the communication between the two private networks goes through a forwarding point (i.e., a separate machine responsible for the actual forwarding of data between the other machines).
- **Tunnel:** An abstract representation of the secure communication channel used in tunneling. It is represented as a source end-point, destination end-point, and a forwarding point. An end-point consists of an IP address and a port number. The goal of tunnels in MAZE is such that in order for a program to traverse (i.e., connect) to another node, it connects to a local server socket (listening at a specified port on the local machine), which in turn securely forwards the network traffic to the destination end-point through the forwarding point.
- **Schedule:** A series of tunnels that connect a list of nodes together.
- **Gateway:** A machine (i.e., node) that accepts connections from the Internet. They are the entry-points to the MAZE system.
- **Black-hole:** A node that does not store any file pieces and is not used as a source end-point of any tunnels (only used as a destination end-point). As such, it has no outgoing tunnels, only incoming ones.
- **MAZE software package:** A collection of software programs that are installed on nodes for MAZE to work.

- **Service-side:** Service-side programs are responsible for computing the schedule and establishing a series of tunnels based on the schedule.
- **Client-side:** Client-side programs are responsible for traversing a series of tunnels to store or retrieve a file.

2.0 Background and Related Work

Moving Target Defense (MTD) is a relatively new sub-field of cybersecurity that attempts to balance the unfair advantage attackers have due to their unpredictability. MTD techniques continuously change a system's attack surface with the aim of increasing the costs (time and resources) of an attack while simultaneously decreasing the success of an attack. In cybersecurity, defenders try to build defense mechanisms against possible vulnerabilities and attacks. However, attackers have the advantage of time to try and find possible ways around the defense mechanisms and deploy their attacks. In MTD, a defender becomes an unpredictable target, constantly changing its properties to avoid being attacked.

In this section, we review cloud storage systems, describe MTD in detail, provide an overview of SSH tunneling, and provide a literature review of recent MTD techniques.

2.1 Cloud Storage Services

Cloud storage is a service for saving data in an offsite location that can be accessed from the Internet or a private network. Data saved on a cloud storage is managed by a third-party who is responsible for keeping the data safe, consistent and available. Cloud storages are built as an alternative to physical hard drives, which have the downsides of only being able to store finite amounts of data and not being highly accessible. The advantage of cloud storages are that they can be accessed anywhere, anytime. Customers do not worry about managing the data and only have to pay for what they need. There are typically three types of cloud storages: public, private, and hybrid. A public cloud storage is one that can be accessed through the Internet. Therefore, any device with Internet connectivity (e.g., mobile phone, tablet or computer) can access a public cloud storage. Examples include Box, Dropbox and Google Drive. A private cloud storage is one that resides within a company's network. Therefore, they can only be accessed when connected to the company's network either physically or through a VPN. The primary difference between a public and a private cloud storage is that the servers in a public cloud storage are shared across customers whereas in a private cloud storage,customers have servers dedicated for their use only. Therefore, private cloud storages are considered to be safer and deliver better performance (in terms of speed). In addition, it is common for servers powering private cloud storages to be physically located inside the company, giving the company greater control. A hybrid cloud storage is the use of both a public and a private cloud storage. For instance, highly sensitive data is stored on the private cloud and less-sensitive data is stored on the public cloud [4, 14].

Cloud storages can be further categorized into file, block or object storages which present and organize data in different ways, each to serve a different purpose. File storage systems organize data similar to how file systems organize data in modern operating systems. Files are stored inside folders and the full path to a file is required when trying to access that file. Block storage systems splits data into singular blocks (i.e., pieces) of data, each with its own address (a unique ID for the block). Blocks can be distributed to other nodes and be configured to work on different operating systems that better suit its needs. The underlying system is responsible for assembling the blocks and presenting the data to the user. Since blocks do not have a path, they can be retrieved quickly through its unique ID (i.e., their address). Block storages are commonly used for databases, which require low-latency and are suitable for RAID systems (since the data is already in blocks) [38]. Object storage systems organize data in a flat hierarchy and are best suited for data that is written once but read many times (e.g., video). Data is split into pieces, known as objects, where each object is comprised of three components: the data (i.e., content) of the object, a unique identifier, and metadata. The unique identifier allows for fast retrieval of the data; the metadata contains contextual information such as size, date, and may even contain detailed information about the contents of the data itself (e.g. location, or name of actors for a video file). Modification of the data results in the creation of a new object. Therefore, object storages are useful for storing static data (i.e., data that is not regularly modified), such as video and music. Amazon S3 is a well-known example of an object storage [15, 7, 2, 3].

MAZE can be implemented as a standalone cloud storage system or be integrated within existing cloud storages to enhance their security. Although in this work we present MAZE as a public file storage system, the techniques used in designing and building MAZE can be applied to any type of cloud storage. MAZE just needs access to a cluster of machines (i.e., servers), which could reside in a public or private cloud.

2.2 Moving Target Defense

MTD is an emerging security paradigm that aims to balance the asymmetric advantage of attackers in that, in non-MTD systems, they have time to study and execute well thought out attacks that exploit unknown vulnerabilities. Meanwhile, defenders have to discover or predict all possible vulnerabilities and either fix them or create defense mechanisms against them. The contributing reason for such an imbalance in the cybersecurity field, is the static nature of computer systems. MTD aims to increase the difficulty and cost of executing attacks by constantly changing and randomizing the properties of a computer system such that attackers are faced with a greater deal of uncertainty, thus shifting the advantage over to the defenders.

MTD is a scalable cybersecurity solution. A system that employs MTD reduces the need to have advanced threat detection. Threat detection is hard, time-consuming, and sometimes unreliable because it may result in false-positives (i.e, detecting an attack when there is none). MTD lessens the burden on threat detection software and cybersecurity teams because it shifts the power to the defenders. Since the attack surface constantly changes, the attacker has a harder time to attack because they have to constantly locate the target. In addition, as computer systems and their infrastructure grow over time, attackers typically gain an advantage because there are more possible entry points to attack. For example, adding new servers means that attackers have more servers to try and infiltrate. With static infrastructure, defense teams work hard to ensure everything is safe by having to either find issues themselves or wait until they happen in order to correct them, at which point, it may be too late. Therefore, static infrastructure is not scalable because with increased infrastructure, defense teams have to work even harder to find every possible bug or vulnerability and fix them or create defense mechanisms for.

The work in [37] categorizes MTD techniques by the different layers of a computer system that MTD can randomize. In this section, we cover MTD techniques for the following layers: platform layer (i.e., the hardware components and the operating system), networking layer, runtime environment, and software application. We can think of these layers as levels, such that if one level does not employ MTD, the level underneath can employ MTD and still provide adequate security. For example, application developers can embed MTD techniques directly into their software. If application developers do not embed MTD, then the runtime system can transform a non MTD-aware software into a secure system. If the runtime system does not employ MTD, then the operating system or the networking layer can embed MTD techniques to provide security and protection against attacks targeted at the layers above.

2.2.1 Dynamic Platform Techniques

Dynamic platform techniques involve changing properties of the underlying platform, such as building applications that can run on top of multiple operating systems and hardware architectures. The aforementioned can be achieved by compiling for different architectures, such as compiling the Linux kernel for both x86 and ARM. The major benefit of dynamic platforms are that they help prevent platform-dependent attacks. Attackers often develop attacks by exploiting bugs in the underlying platform such as in Spectre [33] and Meltdown [35]. Applications that run on different platforms result in attackers requiring more time to develop attacks for multiple platforms. However, this technique comes with several noteworthy drawbacks. First, it cannot defend against attacks targeted at higher-level applications. For example, an application that runs on multiple architectures cannot prevent SQL injections [30]. In addition, a dynamic platform typically results in a more complex system, which may be even harder to maintain and lead to more bugs. The main weakness of a dynamic platform is that if a user has an exploit for Windows, they can just wait until the software migrates to Windows and run their exploit. Therefore, dynamic platform techniques are useful only when the attacker must comprise all platforms to succeed.

2.2.2 Dynamic Network Techniques

Dynamic network techniques change the properties of the network to complicate network-based attacks. Popular techniques in this category revolve around dynamically changing a host's IP address in order to increase the difficulty of locating the host. An example of a network-based MTD technique is to dynamically mutate the IP address and port number of a sender and receiver. The aforementioned can be achieved using a cryptographic hash function, in such a way that only the two communicating hosts can obtain each other's next addresses [43]. A real-life analog is the military's use of frequency-hopping, in which frequencies of radio transmissions are repeatedly switched in order to reduce interference and avoid interception.

2.2.3 Dynamic Runtime Techniques

The runtime environment of an application refers to the layer of abstraction provided by the operating system to higher-level applications. The most important abstraction is the use of virtual memory addresses, which are translated by the operating system to physical addresses in memory. This abstraction creates the illusion that computers have much greater main memory than they actually have. Dynamic runtime techniques change the abstraction provided by the operating system to the application layer. The most well-known example of a dynamic runtime technique is Address Space Layout Randomization (ASLR) [45], which randomizes virtual addresses of the many components of a running program. ASLR is currently implemented in many versions of Windows OS, Mac OS and Linux. In ASLR, the address location of the base executable, stack, heap, and libraries are randomized such that an attacker does not know where to place their malicious code. ASLR is designed to guard against memory corruption bugs such as buffer-overflow attacks [26].

2.2.4 Dynamic Software Techniques

Dynamic software techniques aim to diversify the internals of a software application such as a multicompiler, which can produce different versions of a software from the same source code. Multicompilers use several methods to achieve the aforementioned, for instance, the use of padding to make the size of memory regions unpredictable. They can also use non-operation (NOP) instructions that do not perform any action but change the location of other instructions. The hope with dynamic software techniques is that if an attack works on a variant of the application it is unlikely to work on another variant. For example, if multiple machines run the same software and an attacker successfully develops malware that compromises a machine with the target software, then the attacker can just use the same malware to compromise other machines. Dynamic software lessens the chance that the same malware will work on other machines effectively preventing large-scale attacks. In addition, dynamic software can be used as a measure against code reuse attacks [20]. Some attackers reuse the target software's code against itself because the target software may have defenses against foreign code. However, dynamic software increases the difficulty and cost of code reuse attacks because the location of code and instructions are different on different versions of the software. A drawback of dynamic software techniques is that software applications are typically compiled with special optimization flags. Therefore, introducing randomness in the memory layout through dynamic software techniques may no longer provide the same performance speedup from the optimization flags.

We categorize MAZE as a dynamic networking technique due to its use of tunneling to redirect connections to different machines.

2.3 Secure Shell (SSH) Protocol

Secure Shell (SSH) is a protocol that provides a cryptographically secure connection between two hosts over an unsecure network [49]. SSH can be used for remote command-line access, file transfer and tunneling. SSH was designed as a replacement to Telnet [39], which transferred packets unencrypted over the internet. In the Telnet protocol, anyone with a packet sniffer could eavesdrop on the contents of the packet, which becomes problematic when those packets contain personal or secret information.

MAZE uses the Secure Shell (SSH) protocol in its implementation because it allows for secure remote command-line access and supports tunneling, both of which are fundamental to the design of MAZE as will be described in Chapter 3.

2.3.1 SSH Protocol Overview

SSH splits data into a series of packets containing the following fields: packet length, padding amount, payload, additional padding and Message Authentication Code (MAC). The entire packet, except for the packet length and the MAC, are encrypted. To secure the connection between client and server, SSH uses three data manipulation techniques: symmetric encryption, asymmetric encryption and hashing. Symmetric encryption is used by SSH to encrypt the entire connection. The secret key is generated using the *Diffie Hellman Key Exchange* [22]. Data sent through SSH is encrypted and decrypted using this secret key. Asymmetric encryption is used in SSH by the server to authenticate the client. This process is commonly called *SSH-key based authentication.* The client creates a key-pair and uploads the public key to any server it wishes to access. Once a secure connection has been established between client and server through symmetric encryption, the server authenticates the client by sending a challenge message, encrypted with the client's public key, to the client. If the client is able to decrypt it, then it has proven that it has the associated private key and is therefore, authenticated. Hashing is used in SSH to calculate the MAC, which ensures a received message was not corrupted or altered. The MAC is calculated as the hash of the symmetric key, sequence number and the data. When the packet arrives at its destination, the receiver calculates the same hash of the data and compares it against the MAC in the received packet to ensure its integrity.

2.3.2 SSH Tunneling

SSH tunneling is a method for transporting arbitrary unencrypted data over a secure SSH connection. SSH supports several types of tunneling, but in this section, we focus primarily on two, as they are directly related to this work: `direct port forwarding` and `reverse port forwarding` [13]. In direct port forwarding, the client machine, running an SSH client opens a tunnel and forwards any data that it is sent to an SSH server. The SSH server can live on the end-point (where the data ultimately wants to arrive) or be separate from the end-point, in which case the SSH server connects to the end-point and sends it the encrypted data. For example, it is common for schools to install proxy filters to prevent students from visiting social media websites while connected to school WiFi. Direct port forwarding can be used to bypass such a restriction. We consider a school computer called `school` connected to the school WiFi, and a home computer called `home` connected to home WiFi. The client running on the `school` computer can use the following command to setup a tunnel from the `school` computer to the `home` computer in order to access *www.facebook.com*:

```
ssh -L 5000:www.facebook.com:80 home (executed by ‘school’)
```

The above command binds port 5000 on the client machine to listen for local requests. When the port is connected to by an application, the SSH client proceeds to connect to an SSH server running on `home`. The SSH server on `home` is responsible for accessing *www.facebook.com* and tunnels (i.e., sends) that data back to `school`, effectively bypassing the firewall. It is important to note that the tunnel (i.e, the connection between `home` and `school`) is encrypted while the connection between `home` and *www.facebook.com* is unencrypted. In essence, `home` is acting as a forwarding point to perform requests on `school’s` behalf. We also note that web traffic data as well as any other type of data can be transferred through tunnels. The generic command for direct port forwarding is as follows:

```
ssh -L <local-listening-port>:<remote-host>:<remote-port> <forwarding-point>
```

In reverse port forwarding, the client machine establishes a tunnel to the server but once the tunnel is established, the server listens on a port. Whenever a connection is made to that port on the server, the data is sent back to the client. Consider, we are at home and want to access an internal university resource from our home. We again use the names `home` and `school` to represent the home and school computer respectively. Reverse port forwarding enables us to access internal university resources from home by executing the following command from the `school` machine:

```
ssh -R 5000:www.internal-site.com:80 home (executed by ‘school’)
```

Upon execution, the SSH client at `school` connects to the SSH server running at `home` creating an SSH channel to transfer data. Then, the SSH server at `home` binds to port 5000, listening for local requests. Incoming local requests are forwarded through the newly created SSH channel to `school`, which performs the requests and forwards the data back to `home`. The generic command for reverse port forwarding is as follows:

```
ssh -R <local-listening-port>:<remote-host>:<remote-port> <forwarding-point>
```

2.3.3 libssh

libssh is a C library that implements the SSHv1 and SSHv2 protocols for both client and server applications [10]. It provides an API for developers to write programs that use the SSH protocol. In this section we give a brief introduction on how to use *libssh*, focusing mainly on features that were used to build MAZE.

2.3.3.1 SSH Session To establish an SSH connection with *libssh*, we allocate a new SSH session object by calling the function `ssh_new()`, which returns an `ssh_session` object (i.e., an abstract representation of an SSH session). Similar to standard functions in C, *libssh* follows the allocate-deallocate pattern. An object allocated with `xxxxx_new()` must be deallocated using `xxxxx_free()`. Then, we use the function `ssh_options_set(...)` to set options for the SSH session. The important SSH options are:

- `SSH_OPTIONS_HOST:` name or IP address of host to connect to.
- `SSH_OPTIONS_PORT:` port number of host to connect to (default is 22).
- `SSH_OPTIONS_USER:` the user under which to connect to.
- `SSH_OPTIONS_LOG_VERBOSITY:` log additional details.

Once a session is established, we can connect to an SSH server using `ssh_connect(...)`, passing it the `ssh_session` object we created.

2.3.3.2 Authentication After connecting to an SSH server, we authenticate both the server and the user. The server is authenticated to ensure that it is known and safe to connect to. Server authentication can be performed using the `verify_knownhost(...)` function, which takes in a `ssh_session` object as a parameter. `verify_knownhost(...)` is not part of the standard *libssh* API but its implementation can be found on the *libssh* website [10]. The user is authenticated so the server can identify a user and verify their identity. *libssh* supports several methods of user authentication. The most common user authentication method is by using a password. A password is sent to the server, and it either accepts it or not. The second method is key-based authentication described in Section 2.3.1. Once the user is authenticated, the server grants it access to many resources such as port forwarding. In this work, key-based authentication is used.

2.3.3.3 SSH Tunnels *libssh* supports two types of tunneling:
`direct port forwarding` and `reverse port forwarding`.

- **Direct Port Forwarding** In direct port forwarding, the client machine opens a tunnel to the server at a specific port. When an application at the client machine connects to localhost at that port, their data is forwarded to the server. In order to perform direct port forwarding using *libssh*, we create a separate channel to be used for tunneling, since SSH channels process only a single service. SSH channels are created using the `ssh_channel_new(...)`, which takes in an `ssh_session` object as a parameter. Then, we open a forwarding channel with the `ssh_channel_open_forward(...)` function. To perform the actual forwarding, we use the `ssh_channel_write(...)` function to write incoming data to the server.
- **Reverse Port Forwarding** In reverse port forwarding, the server listen on a port; whenever a connection is made to that port on the server, the incoming data is forwarded back to the client. In order to perform reverse port forwarding using *libssh*, we ask server to listen for incoming connections at specific port using the `ssh_channel_listen_forward(...)` function. In order to accept incoming TCP/IP connections we use the `ssh_channel_accept_forward(...)` function. Once a TCP/IP connection is accepted, a `ssh_channel` object is returned. Then, we read data from the SSH channel using the `ssh_channel_read_nonblocking(...)` function and send that data to the client using the `ssh_channel_write(...)` function.

2.4 Related Work

The use of MTD in securing cloud storage systems is not unique to this work. Many companies have already seen much success creating data storage technologies that implement different strategies of MTD, such as CryptoMove [5] and Nexitech [11]. However, we have noticed a lack in working implementations of MTD-based systems (as opposed to simulations or emulations) in MTD research [43]. Therefore, the main contribution of this research is the design, implementation, and experimental evaluation of a working MTD-based secure cloud storage system. MAZE employs techniques (e.g., tunneling) that have been explored in the field of MTD but not directly applied to securing cloud storage systems. For example, the work in [19] uses tunneling to protect against packet sniffers from determining the end-points (i.e., sender and receiver) of transferring packets. The client's port is constantly changing over time but its traffic is redirected the correct server through tunnels.

OpenFlow Random Host Mutation (OF-RHM) [32] randomly mutates IP addresses of end-hosts such that attackers have greater difficulty locating the target. OF-RHM employs Software Defined Networking (SDN) and OpenFlow for managing the mutation of IP addresses. Each host is given a virtual IP address, which is translated to their real IP address by OpenFlow tables. Simulation results show that OF-RHM can reduce the accuracy of information gathering via scanning by 99% and save 90% of network hosts from scanning worms. The drawback of OF-RHM is that hosts can still be reached through DNS.

Packet Header Randomization (PHEAR) [44] proposes an MTD technique that leverages SDN to securely route traffic in enterprise networks. PHEAR dynamically and transparently removes network identifiers (e.g., MAC and IP address) from packet headers while still correctly routing packets to the correct destination through SDNs. PHEAR attempts to improve upon the drawback of OF-RHM, which was that it distributed virtual IP addresses through DNS, allowing any host (or attacker) to perform a DNS lookup to obtain the target's virtual IP address. Instead, PHEAR consists of end-host proxies which replace packet identifiers with short-lived pseudonyms. An SDN controller is responsible for routing the modified packets based on the pseudonyms. The experimental results of PHEAR demonstrate that it provides low-latency and high throughput, enough for interactive applications such as web browsing.

Catch Me If You Can [29] is an MTD framework that combines address mutation, network stack scrambling and decoy deployment. Address mutation provides randomization of network addresses (e.g., IP addresses) resulting in a target machine harder to locate. Addresses are determined such that they are unpredictable by a third-party but deterministically computable by authorized users. This is achieved through one-way hash functions. The paper explores the possibility of IP address collisions and provides a few possible solutions for it, such as changing the input to the hash function or by simply incrementing the IP address (which is less secure). Network stack randomization prevents host identification based on the communication stack. Decoy deployment employs methods for detecting and stalling an attacker once they are inside the system. For example, traps are placed within the system, which triggers defense mechanisms to trap the attacker.

Mayflies [17] is a fault-tolerant MTD framework for distributed systems. Nodes (i.e., servers and replicas) exist for a short period of time to perform some task or computation and then are destroyed and reincarnated on a different platform with different characteristics (e.g., different OS). Mayflies attempts to defend against progressive attacks or limit the duration of a successful undetected attack. An ongoing attack becomes ineffective in a reincarnated node due to its different characteristics. The paper on Mayflies focuses heavily on evaluating the performance of Mayflies but does not analyze or evaluate its security.

3.0 MAZE Design

The MAZE cloud storage system has two variations, namely reactive MAZE and proactive MAZE, each with its pros and cons. Both approaches are divided into two parts: the service-side and the client-side. The service-side is responsible for preparing the nodes (e.g., node refreshes), generating the schedule, and following the schedule to establish the appropriate tunnels. The client-side is responsible for receiving parameters from a user (e.g., path to a file and password) and following the generated schedule to store or retrieve a file based on the user's parameters. Figure 2 provides a general outline of the responsibilities of the client and service-side, where detailed descriptions of each specific responsibility is described in Section 3.1. The rectangles in red represent features not currently implemented in MAZE.

In this chapter, we describe the underlying architecture of the MAZE cloud storage system. In this work, we focus on the development and implementation of the reactive approach. Therefore, when we refer to MAZE in this work, we are referring to the reactive approach. Both approaches share common building blocks (e.g., establishing of tunnels, storing and retrieving of files, and a command-line program to interact with MAZE) therefore, the proactive approach can be built by weaving in components from the reactive approach. In Section 3.1, we outline the design of reactive MAZE in detail, and in Section 3.3, we give a general overview of proactive MAZE.

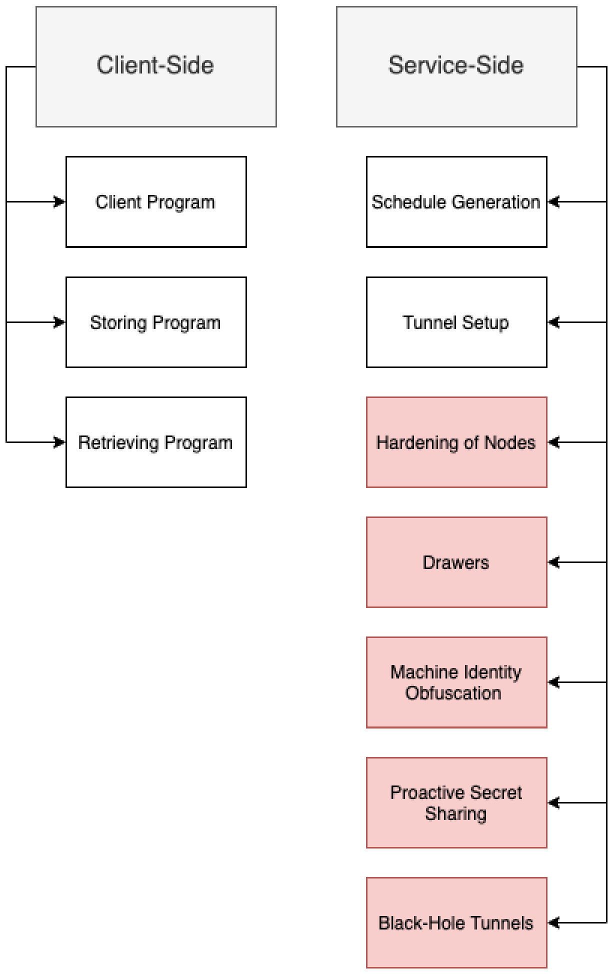

Figure 2: We divide the design of MAZE into two parts: service-side and client-side. The rectangles in red represent features not implemented in the prototype described in Chapter 4

.

3.1 Reactive Approach

The reactive MAZE approach involves establishing tunnels on-demand. In other words, tunnels are only established once a user wants to store or retrieve a file. The client program (a command-line program) runs on the user's machine and is responsible for accepting parameters from the user and sending them to the gateway. The client program accepts three parameters: whether to store or retrieve a file, the path to the file, and a password. The gateway runs a socket server, listening for requests from client programs. Once a request is received, the gateway parses the user's parameters and uses them to initiate the tunnel setup process. The tunnel setup process involves splitting the file into pieces (where the number of pieces is equal to the number of tunnels), and then using the password to create and build the schedule (i.e., establishing the tunnels). The password is used to deterministically determine which nodes and ports to establish tunnels between. At a high-level the schedule generation works as follows: we hash the password and use the first two bytes of the hash output to determine the port of the source end-point of the tunnel, the middle two bytes to determine the destination end-point's IP address, and the last two bytes to determine the destination end-point's port number. We repeat the aforementioned process for the number of desired tunnels. Ideally, we would want to hash the concatenation of the password with a secret client token to lessen the chances of hash collisions; however, client authentication is out of the scope of this work.

We limit the connections to the tunnel end-points such that they can only accept connections from `localhost`. Therefore, in order to traverse to a different node, a program must connect to `localhost` at a specific port on the current node. In other words, programs cannot directly connect to a different node through SSH. They must follow the tunnels, which redirects their connections. At any given time, there could be multiple tunnels stemming from the same node but at different source port numbers and to different destinations.

The file pieces are split in such a way that each piece is numbered. We can therefore, define a file as $F = \{f_1, ..., f_n\}$ where:

- F: represents a file.
- n: represents the number of pieces.
- f_i: represents a file piece.

3.1.1 Service-Side

The MAZE service-side is responsible for preparing the nodes (hardening of nodes and node refreshes), generating a schedule based on the password, and iterating through the schedule to establish the appropriate tunnels.

In MAZE, nodes are *hardened* such that the cost of traversing a regular network link is much greater than the cost of traversing a tunnel. Hardening a node involves limiting the connections to the tunnel end-points such that they can only accept connections from `localhost`. Traversing a regular networking link by an attacker requires exploiting a vulnerability in a networking server on the target node. Due to the difficulty of traversing a networking link, we assume that attacker agents use tunnels to hop between nodes. Client programs also rely on tunnels to access hardened nodes. Tunnels are one-use only, once a program traverses through a tunnel, no other program or attacker can use that tunnel.

Each node has two refresh periods: (1) nodes are restarted and the system software is copied over from a secure read-only medium; and (2) file pieces are modified to become incompatible (i.e., unable to be glued back together) with file pieces before modification. In the first refresh period, tunnels are torn down and re-established, and any attack agents are evicted from the node and have to regain control over the node. In the second refresh period, file pieces are modified using *proactive secret sharing*. In order for an attack to succeed, an attacker must capture all file pieces, in one pass, before the attack agents are killed by the periodic refresh. If an attacker retrieves a subset of the file pieces and then is killed and reattempts to connect to retrieve more file pieces, the newly retrieved file pieces become incompatible with the previously retrieved ones. In other words, file pieces collected over different refresh periods can no longer be glued back together to form a single coherent file. Having more tunnels make it harder for an attacker to retrieve all file pieces, in time, before the refresh.

To further increase difficulty for an attacker, we setup *black-hole tunnels*, in which the end-point of the tunnel is a node that never stores any files and does not have any outgoing tunnels. Therefore, when an attack agent traverses through black-hole tunnels, they arrive at a node in which there are no tunnels setup and an attacker agent is effectively stuck. While in black-hole nodes, attack agents are unable to hop to other nodes and there are no file pieces to retrieve. Attack agents in black-hole nodes are eventually killed during refresh.

File storage space in the MAZE system is organized as *drawers* of data. Each file node contains a large number of drawers where each drawer contains the content of a file piece or gibberish data, meant to confuse attackers and increase the cost of being able to retrieve every file piece. The name of each drawer is pseudo-randomly generated and all drawers are of the same size, such that the name or the size of a drawer reveals no information about the data. In order for an attacker to retrieve a file piece they must know which drawer it is in, using content analysis for example.

3.1.1.1 Schedule Generation

The schedule generation algorithm is responsible for generating an array of consecutive tunnels for the client to traverse in order to store or retrieve a file. Consecutive tunnels refers to when tunnels are established from a source to a destination, and the source of the next tunnel is the destination of the previous tunnel.

The schedule generation algorithm uses hashing to determine source and destination end-points of tunnels from the user's password. Given a password, we build a schedule using certain bytes of the hashed password to determine the source port number, the destination node (uniquely identified by its IP address), and the destination port number. The source port number refers to the port number at the current node at which to connect to in order to traverse through the tunnel. The destination node and the destination port number are the end-points of the tunnel where file pieces are ultimately stored. *Tunnels are established between two machines in such a way that connecting to localhost on the source port number redirects the connection to the destination node at the destination port number, effectively traversing that tunnel.*

The first node the client program connects to, is the gateway. From the gateway, there are three pieces of information it needs to know in order to establish a tunnel: (1) the source port number, (2) the destination node and (3) the destination port number. The three aforementioned pieces of information can be computed from the user's password. First, the schedule generation algorithm hashes the password and uses the first two bytes of the hash output to determine the source port number, the middle two bytes to determine the destination node, and the last two bytes to determine the destination port number. We convert those bytes into integer values, and for the port numbers ensure that the integer values fall between the range of valid TCP server port numbers (1,024 and 65,535). In order to determine the destination node (i.e., its IP address), we use the converted integer value as an index into an array of MAZE nodes and their associated IP addresses.

Figure 3 provides an example of how a string of 256 bytes can be used to determine the source port number, destination node, and destination port number. The first two bytes (bytes 0 and 1) are used for calculating the source port number, the middle two bytes (bytes 127 and 128) are used for calculating the destination node and the last two bytes (bytes 254 and 255) are used for calculating the destination port number.

Formally, we define a schedule as $S =< Tunnel_i >$ where:

- $Tunnel_i$ represents a single tunnel.
- $Tunnel_0 = h(password)$ and $Tunnel_i = h(Tunnel_{i-1})$
- $h(...)$ represents a hash function.

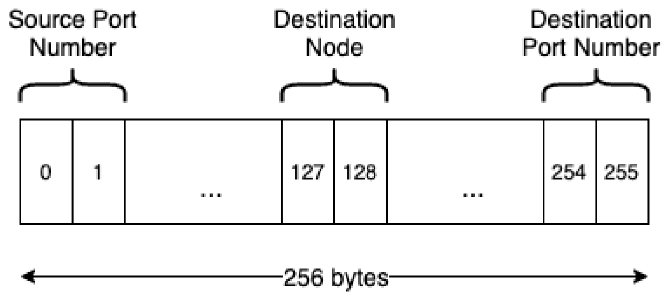

Figure 3: Deriving tunnel end-points from a 256-byte password hash.

Figure 4 demonstrates the schedule generation process. The schedule is an array of tunnels, where the previously hashed output is hashed once again in order to obtain a new hash output of which we can follow the above formula to determine the tunnel end-points. In Section 4.2, we experiment with a varied number of tunnels and evaluate the performance. We could also pseudo-randomly determine the number of tunnels by using certain bytes from the hashed password as previously done to determine the port numbers.

Due to the limited number of available TCP port numbers (`1,024-65,535`), and depending on the number of available MAZE nodes, the schedule generation algorithm could compute a tunnel from a source end-point of an existing tunnel (IP address and port number) to a different destination end-point, causing a collision. We cannot have two tunnels from the same source to two different destinations. In Section 5.1, we discuss possible solutions to handling tunnel collisions.

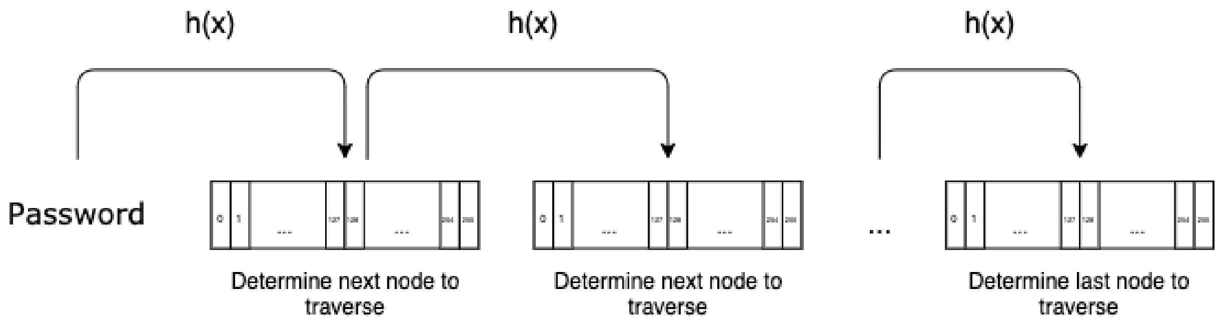
h(x)
h(x)
h(x)
Password
Determine next node to traverse
Determine next node to traverse
...
Determine last node to traverse

Figure 4: Generate a schedule by continuously hashing the previous hashes of the password, and using the hash outputs to determine the tunnel end-points.

3.1.1.2 Tunnel Setup The actual setup and building of tunnels is performed by a `forwarding program`, run on the gateway node. The forwarding program is responsible for establishing tunnels specified by the schedule; each tunnel is between two nodes (source and destination end-point) but the tunneling (i.e., forwarding of the data) is performed by the gateway. The tunnels are used by the client program to hop from one node to the other, retrieving file pieces at each node. To hop from node to node, the client program connects to the current machine (i.e., `localhost`) at a specific port and the tunnel redirects the connection to a different node at a different port. The idea is that the client program should not know which node they are connecting to, until they are connected to it. The forwarding program first runs the schedule generation algorithm to get an array of tunnels. Then, it iterates through the array and establishes the tunnels accordingly. In Section 4.1, we discuss, in detail, the implementation of tunnels using SSH tunneling.

MAZE uses tunnels for two main reasons: (1) a tunnel provides a secure connection over an untrusted network link and (2) it is single-use and one-way. A tunnel, as defined in MAZE, can only be used once. After a program uses a tunnel to connect to a different node, that same tunnel cannot be used again unless it is re-established. Single-use tunnels are important to decreases the chance that an attacker can find and use an available tunnel before an authorized client program uses it. Authorized client programs know the exact ports they need to connect to in order to traverse a tunnel, to store or retrieve a file. Since they know the exact port, they can connect to it directly and have their connection forwarded to the next node. Meanwhile, attackers are just guessing for available ports in a brute-force manner, which is slower.

It is important to note that since the MAZE system can be thought of as a graph, the starting node of an attacker and an authorized client program are the same. They must both start at a gateway node, which is an internet-reachable machine. Other nodes are not publicly reachable through the Internet. Therefore, since attack agents and client programs both start at the same node but the client program knows exactly which ports to connect to in order to arrive at the subsequent node, a client program has a higher chance to use the tunnel first, rendering it unusable by the attack agent.

In order to prevent an attack agent from just non-stop trying to connect to all ports, the MAZE system employs a periodic refresh, in which all non-root processes, including the attack agent, are killed . We can add an additional layer of protection by introducing a delay once a node wants to connect to `localhost`. Because an attacker has to try all possible ports, the delay causes an attacker to require increasingly more time to find the correct port while a client program can just connect directly to the port. Even though an authorized client will also have a delay when connecting, it is just one added delay per tunnel.

3.1.2 Client-Side

The MAZE client-side is responsible for (1) accepting parameters from the user (e.g., whether to store or retrieve a file, the path to the file, and a password) and sending them to the gateway; (2) splitting the desired file into pieces; and (3) executing a program that follows the tunnels and either stores or retrieves the file pieces.

3.1.2.1 Client Program The client program is command-line program, that runs on a user's machine, used to access and interact with the MAZE cloud storage system. It is responsible for accepting a list of arguments and sending the arguments to the gateway. The client program requires the following parameters: (1) `-s`, storing of a file, or `-r`, retrieving of a file, followed by the path to that file; and (2) `-p`, the password with which to compute the schedule. A file stored with a password `p` must also be retrieved with the same password `p`. For example, the command below stores a file called *book.txt* with a password of *pittsburgh*:

```
$ maze -s book.txt -p pittsburgh
```

Similarly, the command below retrieves a file called *book.txt* with a password of *pittsburgh*:

```
$ maze -r book.txt -p pittsburgh
```

3.1.2.2 Storing and Retrieving The storing program is part of the MAZE software package that is installed on all MAZE nodes. It is executed on the gateway after establishing the appropriate tunnels. The storing program follows the schedule, connecting to the respective ports (source port numbers) to arrive at subsequent nodes and copying the appropriate file pieces from the client to the current file node.

The retrieving program is also part of the MAZE software package. It is almost identical to the storing program; it follows the schedule of tunnels, but instead of copying file pieces from the client to the current node, it copies them from the current node back to the client.

3.2 Tradeoffs

A significant bottleneck of the MAZE design is the use of tunnels. When a client wants to store or retrieve a file, a set of tunnels are established for client programs to traverse. In the current implementation of MAZE, the number of tunnels is passed in as a user-specified argument for performing experiments with different numbers of tunnels. Establishing and traversing a tunnel requires more time than directly connecting to a node through SSH. Therefore, having more file pieces results in more tunnels to traverse and slower performance. However, at the cost of slower storing and retrieving speeds, an increase in tunnels results in greater security. An attacker needs a certain amount of time to find each tunnel, and they are assumed to find each tunnel sequentially [1]. In other words, only after they find the first

[1]Traffic analysis techniques can enable the attacker to discover the tunnels in parallel. We leave this attack model for future work.

tunnel can they try and find the second tunnel in the schedule. Therefore, the more tunnels, the greater the total time an attacker needs to find all the tunnels. If the total time is greater than the node refresh period, then the attack fails. This tradeoff between performance and security is analogous to that of Onion Routing [41], in which the larger the number of layers or hops, the slower the performance. However, this slow-down in performance allows for increased security for anonymous browsing of the web.

3.3 Proactive Approach

The proactive MAZE approach involves a periodically and constantly changing re-configuration of tunnels. In proactive MAZE, tunnels are already established (i.e., do not wait for client programs to connect to setup tunnels) and tunnels are pseudo-randomly torn down and new ones are established. In this section, we provide an overview of the proactive MAZE approach, although we do not implement it in this work.

In proactive MAZE, the cloud storage system configuration is represented as a graph of nodes connected by edges (i.e., tunnels). It involves pseudo-randomly changing the system configuration by constantly tearing down tunnels and establishing new ones. The system configuration is publicly known, therefore anyone can know which tunnels are currently established. In the reactive MAZE approach, a password is used to determine the source and destination end-points of tunnels. However, in the proactive MAZE approach, a password is used to determine the nodes in which to store or retrieve the file pieces. The schedule generation algorithm in the proactive approach is responsible for using the password to determine a list of nodes and determine which file pieces are to be contained in which node. The aforementioned is collectively known as the schedule. Based on the schedule, we compute the shortest path (through the tunnels) that visits all the nodes in the schedule, before the system reconfigures, to store or retrieve a file. On a timely basis, the system reconfigures and file pieces at each node are modified through *proactive secret sharing* techniques.

The proactive approach introduces the idea of *node shuffling*, in which the content of the nodes (i.e., file drawers) are shuffled amongst other nodes. The node shuffling algorithm is publicly known, in other words, anyone knows which node's contents moved to which other node. Recall that file pieces are structured as drawers. Since authorized client programs have the password, they can determine which nodes currently contain the drawers in which their file pieces are stored in. Given those nodes (that contain the appropriate drawers), the client program computes the shortest path that visits those nodes and accesses the respective drawers to store or retrieve file pieces. An attack agent does not know which drawers contain their desired file pieces. Therefore, they resort to copying all the contents of a node, most of which are fake files with gibberish data. In order for an attack to succeed, it must retrieve all the file pieces before the system reconfigures because after reconfiguration the file pieces are modified such that they become incompatible with the previous versions. The advantage of the proactive approach is that response time is much faster since the tunnels are already established once a user wants to store or retrieve a file.

4.0 MAZE Evaluation

In this chapter, we evaluate the performance and security of the MAZE cloud storage system by implementing the proposed design, and building an experimental platform to perform experiments on MAZE. In Section 4.1, we describe in detail our implementation of MAZE and in Section 4.2 and 4.3 we present and analyze the experiment results.

4.1 MAZE implementation

In this work, we focus on the implementation of the reactive MAZE approach, accomplished through a series of C programs and Bash scripts. We divide the implementation into two parts: the service-side and the client-side, which correspond to the service-side and client-side in the MAZE design.

4.1.1 Service-Side

The service-side programs are responsible for listening for incoming connections from client programs, and parsing the user's arguments to generate the schedule. Once the schedule is generated we can iterate through it and establish tunnels accordingly. In our implementation we use the *libssh* library build tunnels using SSH.

As mentioned previously in Figure 2, the hardening of nodes, drawers, machine identity obfuscation and proactive secret sharing are currently not implemented in MAZE and therefore not discussed.

The service-side consists of the following programs (all of which are run on gateway nodes):

- `Gateway Program:` Listens for incoming connections from client programs.
- `Master Forwarding Program:` Generates and iterates through the schedule establishing tunnels at the respective end-points.

- `Master SSH Server Process:` Generates and iterates through the schedule starting SSH servers at the tunnel destination end-points.

4.1.1.1 Gateway Program The gateway nodes run a socket server, implemented using the POSIX sockets API, that listens for incoming connections from client programs. Once the server receives a connection from an authorized client, it reads the incoming the data (i.e., the user's parameters) and parses the data using the `strtok` C library function, storing the parameters in an array. We did not implement client authentication as it is orthogonal to this work. After receiving parameters from the user, the gateway begins the tunnel setup process. We fork a new process to run the *master SSH server process* and fork another process to run the *master forwarding program*, passing the password as an argument to both programs. Once the SSH servers and tunnels are established, the gateway either runs a program to store a file or retrieve a file, based on the user's parameters.

4.1.1.2 Schedule Generation Algorithm We represent a schedule as an array of tunnels, where a tunnel is defined as the following C struct:

```c
struct Tunnel
{
    char *src;
    char *src_port;
    char *dest;
    char *dest_port;
};
```

A tunnel consists of: (1) a source node (char *src), represented by an IP address; (2) a source port number (char *src_port), which is the port number at the source node that leads to the tunnel; (3) a destination node (char *dest), which is the node that the tunnel leads to, represented also by its IP address; and (4) the destination port number, which is the port number at the destination node (char *dest_port) that the tunnel connects to.

The schedule generation algorithm uses SHA256 to continuously hash a password in order to determine the source and destination end-points. We use the OpenSSL [12] C library, in the implementation, since it has built-in functions to generate SHA256 hashes from a given string. The first source node in the schedule is the gateway node. In other words, the source node of the first tunnel in the array of tunnels is the gateway node. To determine the remaining members of the tunnel (source port number, destination node and destination port number) we use SHA256 to create an arbitrary string of 256-bytes from the password. The first two bytes of the hash are converted to an integer to determine the 16-bit TCP source port number. The same is done to the last two bytes of the hash to determine the 16-bit TCP destination port number. The middle two bytes are converted into an integer to determine the destination node. The integer value of the middle two bytes represent an index into an array of available file nodes. If the index is greater than the length of the array, we mod it by the length of the array. The aforementioned steps determine the first tunnel in the schedule. To determine the second tunnel in the schedule, we use the previous destination node as the source node of the next tunnel, thus creating a connected series of tunnels. Then, we perform the same process as before, using SHA256 to hash the previous hash output and using it to calculate the other members of the tunnel. Depending on the number of available file nodes, it is possible that file nodes are used more than once.

MAZE uses SSH to establish tunnels and to connect to other nodes. SSH listens on a default port of 22. Other than port 22, SSH allows port numbers between 1,024 and 65,535. Therefore, we must ensure that when calculating the source port number and the destination port number, that it falls between the range of 1,024 and 65,535. We achieve the aforementioned using the following formula:

```
port number = integer_hash_value % (65535 - 1024 + 1)) + 1024
```

where *integer_hash_value* is either the source or destination port number determined by converting the first or last two bytes of the hash output into an integer.

In the current MAZE implementation, the number of tunnels is passed in as a user-specified parameter because, in the experiment section, we want to test how the number of tunnels affect performance. We do not handle tunnel collisions in the current implementation, but discuss possible solutions to collisions in Section 5.1. The password that we used in the experiment happened to have no collisions for up to 80 tunnels.

4.1.1.3 Forwarding Program Tunnels in MAZE are implemented using SSH through the *libssh* C library. To establish the tunnels, we use a combination of reverse port forwarding and direct port forwarding, both of which are described thoroughly in Section 2.3. The *forwarding program* is run on the gateway, which performs the actual forwarding of data. In Figure 5 we illustrate an example of how the tunneling is implemented in MAZE.

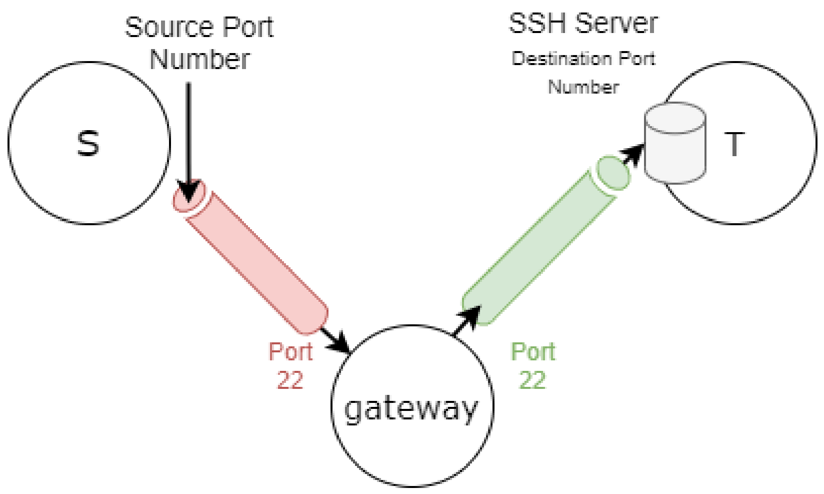

Figure 5: Implementation of a tunnel in MAZE. Both S and T are file nodes. Each node runs an SSH server listening at port 22 and only accepts connections that use the access-protected administrator key for authentication. The destination file node T additionally runs an SSH server that listens at the destination port number and accepts connections only from `localhost` using the client key for authentication. When the client program starts an SSH connection to the source port on node S, its connection is forwarded through the tunnel to the SSH server at the destination port on node T.

In the Figure 5 we have three nodes: node S (the source), node T (the destination) and the gateway node. The red tube represents `reverse port forwarding` and the blue tube represents `direct port forwarding`. We refer back to this Figure and their nodes when explaining the implementation of the forwarding program.

The *forwarding program* takes as input four arguments: (1) the source node's IP address, (2) the port number at which the source machine listens for requests to be forwarded, (3) the destination node's IP address, and (4) the port number at which the destination node listens for incoming requests.

In the code snippet below, we first ask node S (from Figure 5) to listen for incoming SSH connections. Then, the *forwarding program* blocks and wait for incoming requests; once it receives a request, the incoming SSH channel (connection) is saved as an `ssh_channel` object.

```
/* Listen for incoming connections */
rc = ssh_channel_listen_forward(session, hostserver, hostport, NULL);
...
/* Accept incoming TCP/IP forwarding channel */
channel = ssh_channel_accept_forward(session, TIMEOUT_MS, &port);

/* TIMEOUT_MS refers to the time (in milliseconds) that
 the tunnel stays open for.
*/
...
```

Once data are received from S, we can begin the forwarding process to node T as demonstrated in the code snippet below. We open an SSH channel dedicated for forwarding the data using the `ssh_channel_open_forward(...)` function and pass it the required parameters.

```
// Create a forwarding channel
forwarding_channel = ssh_channel_new(session);
...
rc = ssh_channel_open_forward(forwarding_channel, destserver, destport,
                              hostserver, hostport);
```

Afterwards, we read data from node S through its SSH channel and write that data to node T, through the newly created forwarding channel, effectively forwarding the data. In addition, we also do the opposite, where we read data coming from node T and write it back to node S. A code snippet of the aforementioned is provided below:

```
  while (1) {
    // Read bytes from incoming channel
    nbytes = ssh_channel_read_nonblocking(channel, buffer,
                                          sizeof(buffer), 0);
    if (nbytes > 0) {
      // Write bytes to outgoing channel
      nwritten = ssh_channel_write(forwarding_channel,
                                   buffer,
                                   nbytes);
    ...
    }
    ...
    // Read bytes from incoming channel
    nbytes = ssh_channel_read_nonblocking(forwarding_channel,
                                          buffer,
                                          sizeof(buffer),
                                          0);

    if(nbytes > 0) {
      // Write data to SSH channel
      nwritten = ssh_channel_write(channel, buffer, nbytes);
      ...
    }
  }
```

The *forwarding program* establishes a single tunnel between two nodes. However, the goal is to build all the tunnels specified in the schedule. Therefore, we built a *master forwarding program*, run on the gateway node, that is responsible for iterating through the schedule, forking new processes to call the *forwarding program* (described above) and giving it the necessary parameters in order to build all the necessary SSH tunnels.

4.1.1.4 SSH Server Program The *SSH server program* is responsible for starting SSH servers at a node. When the *forwarding program* establishes a tunnel from a source end-point to a destination end-point, we ask the source node to listen at a specific port and when a program connects to that port, they are redirected to the destination node at a specific port. In order to connect to the destination node a specific port, the destination node must have an SSH server listening for requests at that port. The MAZE software package includes scripts for both starting and stopping SSH servers. We illustrate the opening of an SSH server in the code snippet (Bash script) below:

```
# $1 = port number of SSH server to start

#!/bin/bash
if [ ! -f ~/.ssh/id_rsa_$1 ]; then
```

```
  # Generate SSH key file
  ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa_$1
fi

# Replace port number in config file
sed -i '5s/.*/Port '"$1"'/' sshd_config

# Restart SSH
/usr/sbin/sshd -h ~/.ssh/id_rsa_$1 -f ./sshd_config
```

The `ssh-keygen` command generates a host key file, if one does not already exist, which is required when starting an SSH server. The `sed` command looks for the port number in the SSH config file and replaces it with the desired port number. Then, we run the `sshd` command and specify (1) the newly generated host key, and (2) the modified sshd config file, which opens an SSH server at the desired port.

To stop an SSH server, we locate the process ID of the SSH server and kill it. We illustrate the termination of an SSH server in the code snippet (Bash script) below:

```
# $1 = port number of SSH server to terminate

# Determine address of SSH server
addr="0.0.0.0:"
addr+=$1

# Find process ID of the SSH server to terminate
pid=$(netstat -ant -p tcp | grep ssh | grep -m 1 $addr | \
awk "{print \$7}" | cut -f1 -d"/" | sed 's/\s.*$//')

# Terminate SSH server
kill -9 $pid
```

The expression in the `pid=$(...)` line finds the process ID of the SSH server at the desired port. Once found, we stop it using the `kill` command.

The *SSH server program*, described above, starts an SSH server at a single node. Therefore, similar to the *forwarding program*, we built a *master SSH server program* that is responsible for iterating through the schedule and forking new processes to start SSH servers at all the tunnel destination end-points.

4.1.2 Client-Side

At the client-side, a command-line program is responsible for accepting arguments from a user (e.g., whether to store or retrieve a file, the filename, and a password) and sending those to the gateway. Then, at the gateway, after the service-side programs are executed, client-side programs are run, which traverse through the tunnels and store or retrieve file pieces.

The client-side is consisted of the following programs:

- `Client Program:` A command-line program on the user's machine that accepts and sends arguments to the gateway.
- `Storing Program:` Traverses through the tunnels, storing file pieces at each node.
- `Retrieving Program:` Traverses through the tunnels, retrieving file pieces at each node.

4.1.2.1 Client Program The *client program* is a command-line program, that runs on the user's machine (i.e., the client node), and is responsible for accepting parameters from the user and sending those to the gateway. We implemented the *client program* in C, which uses the `getopt` C library function to parse command-line arguments. Before sending the arguments to the gateway, the client program splits the desired file into pieces using `split` command, available in Unix-like systems. The `split` command accepts many optional arguments, of which, we use the `-n` option to specify the number of pieces to split the file into, and the `-d` option to use numeric suffixes (i.e., the pieces are numbered).

The user parameters are concatenated into a single string, separated by spaces, and sent to the gateway. We implement sockets using the POSIX sockets API, however, data transferred through plain sockets is not encrypted and could thus be visible by packet sniffing software. We encrypt the socket connection with SSH direct port forwarding, which we discussed thoroughly in Section 2.3. In summary, a tunnel is setup from the user's machine to the gateway, which transports the data (the user's parameters) through a secure SSH connection.

4.1.2.2 Storing Program The *storing program* copies file pieces from the user's machine to file nodes; determines the source port number to connect to in order to arrive at the subsequent node in the schedule; and then runs the same *storing program* at the newly arrived node. The aforementioned approach effectively works in a recursive manner; at each traversal, the storing program checks if it has already traversed through all the nodes, if not, calculate the source port number, connect to it, and run the *storing program* at that newly connected node.

The implementation of the *storing program* is separated into two components. The first component computes the source port number but does not connect to it; and the second component copies the respective file piece from the user's machine to current node, and then connects to `localhost` at the source port number, which traverses through the tunnel to arrive at the subsequent node.

The schedule contains all the source and destination end-points that the *storing program* needs. However, we do not directly use the schedule in the *storing program* because it would have to be either passed in as a parameter (which could become costly as the schedule becomes larger) or it would have to be recreated at each storing program. The source port number is computed by converting the first two bytes of the password hash into an integer value and ensuring the value falls between the valid range for TCP port numbers. Therefore, in the *storing program* we just need to hash the previous password hash output to compute the source port number in order to traverse the tunnel. As such, we pass in the previous password hash output as an argument to the storing program, such that it can be used to compute the source port number.

The first component is implemented as a C program, which requires the following arguments: the user's IP address, the filename, the file number, and the previous password hash. When the *storing program* is first executed by the gateway, the file number is the number of pieces the file was split into. The C program begins by hashing the previous password hash using SHA256 to get a new 256-byte hash and using that new hash to compute the source port number. Then, the C program decrements the file number and checks if the file number is equal to zero (which means all pieces have been stored); if it is not then it executes the second component, implemented as a Bash script, and passing it the following arguments:

the user's IP address, the filename, the decremented file number, the source port number, and the new password hash; if the file number is equal to zero, then we do nothing else (do not execute the Bash script) because we have already copied over all file pieces from the user's machine to the respective MAZE nodes.

The second component, implemented as a Bash script, is responsible for determining the name of the file piece to be copied, copying over the file piece from the user's machine, and then connecting to `localhost` at the source port number to arrive at the subsequent node and run the storing program from that node.

A code snippet of the Bash script (the second component) is provided below:

```
#!/bin/bash

# $1 = user's IP address
# $2 = filename
# $3 = file number
# $4 = source port number
# $5 = previous hash

# Determine filename
filename="${2}${3}"

# Copy file piece from user's machine to the current node
scp -i ~/.ssh/maze.pem $1/$filename ./

err=1
while [ $err -ne 0 ]
do
    # Connect to localhost at the sourcce port number
    ssh localhost -i ~/.ssh/maze.pem -p $4 -t \
    "./store␣\$1␣\$2␣\$3␣\$5"

    err=$?
done
```

The Bash script begins by concatenating the filename and the file number to determine the name of the file piece to copy over from the user's machine. Then, the script uses the `scp` command to copy the file from the user's machine to the current node. We traverse to the next node in the schedule by connecting to `localhost` at the source port number using SSH, which traverses the tunnel. In the SSH command, we use `-t` option, which executes a command at the destination node once the SSH has succeeded. We use the `-t` option to run the *storing program* (the first component) at the next node giving it the required parameters. By executing the *storing program* at the next node, we effectively create a recursive process. The while loop is used to try and traverse through the tunnel until it succeeds. It is impossible that the client program traverses through the tunnels faster than they are established. Therefore, we try to connect until it succeeds, which is slightly faster than waiting for all the tunnels to be established and then executing the storing program.

4.1.2.3 Retrieving Program The implementation of the *retrieving program* is almost identical to that of the *storing program*. The *retrieving program* is also separated into the same two components: a C program and a Bash script. The C program is identical since the process of computing the source port number is the same. The difference lies in the Bash script. Instead of using `scp` to copy file pieces from the user's machine to the current node, it does the opposite. It copies file pieces from the current node back to the user's machine. We illustrate this in the code snippet below:

```
#!/bin/bash

# $1 = user's IP address
# $2 = filename
# $3 = file number
# $4 = source port number
# $5 = previous hash

# Determine filename
filename="${2}${3}"

# Copy file piece from user's machine to the current node
scp -i ~/.ssh/maze.pem $filename $1

err=1
while [ $err -ne 0 ]
do
    # Connect to localhost at the source port number
    ssh localhost -i ~/.ssh/maze.pem -p $4 -t \
    "./retrieve $1 $2 $3 $5"

    err=$?
done
```

We notice that the Bash script is almost identical to that of the *storing program* except for the `scp` command, which copies file pieces from the current node to the user's machine. In addition, after it connects to the tunnel and arrives at the next node, it executes the *retrieving program* at that node, once again creating a recursive process.

4.2 Experiment Setup

Our experimental platform used 12 t2.micro Amazon EC2 instances [1] with 8GB of memory running Ubuntu 16.04. All Amazon EC2 instances were located in the us-east-1 region (North Virginia). We setup one instance as the gateway node, one instance as the client node, and the remaining ten as file nodes. The setup process involved downloading the MAZE software package on all instances.

We evaluated the performance of MAZE by performing two experiments. The first experiment tests the overhead of using MAZE to store and retrieve files with varying file sizes (50KB, 100KB, 1MB, 100MB, 500MB and 1GB) with a constant number of tunnels (10 tunnels). We generated test files of varying sizes using the `mkfile` command in MacOS. The second experiment tests the overhead of using MAZE to store and retrieve a file of size 100 MB with a varying number of tunnels (10, 20, 30, 40, 50 tunnels). The overhead of MAZE was calculated as follows:

$$\frac{\text{average store/retrieve time with MAZE - average store/retrieve time without MAZE}}{\text{average store/retrieve time without MAZE}}$$

where *average time* is the average time (in seconds) of ten executions of storing or retrieving; and *without MAZE* refers to transferring file pieces through `scp`. For storing and retrieving *without MAZE*, we generated schedules of varying tunnels (10, 20, 30, 40, 50) and output the destination node's IP addresses to a text file. Then, we wrote a script that reads that text file, line-by-line, and splits the desired file into the appropriate number of pieces (where the number of pieces is equal to the number of tunnels) and transfers them to the destination node through `scp`. For all the experiments, we only use one client and do not take into account the time to split or assemble a file as they are independent of the tunnels. In addition, for all experiments, we use the same password, *pittsburgh*, which does not have tunnel collisions for up to 80 tunnels. In order to compute the time elapsed for storing and retrieving a file with MAZE, we modified the *storing* and *retrieving program* to record the time at which the tunnel setup process started (before the establishing of tunnels) and the time at which the last piece of a file was stored or retrieved. Since we are using Bash scripts to recursively traverse the tunnels, we could not directly compute the elapsed time (end time - start time) within the Bash script since it does not support floating point subtraction. Therefore, we would compute the start and end time using Bash and output those times into a file, which we transferred back to the client node. Then, we used Python to read those files and calculate the elapsed time.

4.3 Results and Discussion

Figure 6 displays the overhead of storing and retrieving a file with sizes: 50KB, 100KB, 1MB, 100MB, 500MB and 1GB with 10 tunnels. We observe that, in Figure 6, the overhead of both storing and retrieving a file decreases drastically from about 300% to about 50%. In addition, we notice that the overhead for small files (50KB, 100KB, 1MB) was very similar, around 300%, but the overhead decreased as file sizes became larger. The reason for the aforementioned is that, for small files, the primary bottleneck was the establishment and the traversal of tunnels, while for large files, the primary bottleneck was the size of the file. Recall that the establishment and forwarding of a connection by a tunnel is performed by the gateway. Therefore, for each tunnel, the connection is first sent to the gateway and only then forwarded to the destination (a triangulation). As such, a tunnel is inherently slower than a direct connection to the destination. However, as the file size increases, the overhead of a tunnel is overshadowed by the overhead of transferring a large file piece. We expect that with even larger files, the overall overhead of MAZE decreases even further. Therefore, we can conclude that MAZE is best suited for storing large files. If a user wants to store multiple files with MAZE, it would be more efficient to package the individual files into a single large file and store the large file.

It is important to note that we used t2.micro Amazon EC2 instances, which are low-tier

VMs offered by Amazon AWS. They are described to have *Low to Moderate* networking performance. The networking performance of the VM would often cause traversing through tunnels to be slower than usual. In addition, we only had ten Amazon EC2 machines for testing, therefore, when the number of tunnels was greater than ten, VMs would be reused.

In Figure 7, we evaluate the overhead of storing and retrieving files with MAZE with varying number of file pieces. Recall that the number of file pieces corresponds to the number of tunnels. Therefore, ten file pieces with MAZE, represents splitting a file into ten pieces and generating a schedule of ten tunnels. In terms of *without MAZE*, we also split the file into the same number of pieces as with MAZE (e.g., ten pieces) but instead of traversing through the tunnels, we generated the schedule and simply used `scp` to transfer the pieces to the destination nodes. We observed that the overhead of storing and retrieving increased as the number of tunnels increased. As mentioned previously, the cost (in terms of response time) of traversing a tunnel is greater than that of directly using `scp` to transfer files. Therefore, it is natural to expect the overhead of MAZE to increase as the number of tunnels increases.

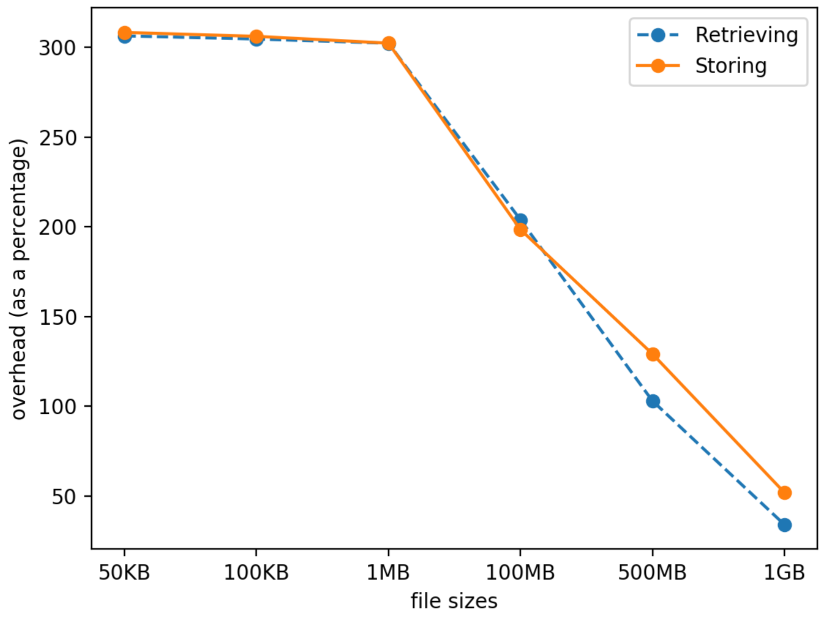

Figure 6: Overhead of Storing and Retrieving File Pieces with MAZE (varying file sizes)

Average Storing Time (secs)		
file size	with MAZE	without MAZE
50KB	12.179	2.982
100KB	12.146	2.990
1MB	12.186	3.028
100MB	12.358	4.139
500MB	19.046	8.308
1GB	26.483	17.424

Average Retrieval Time (secs)		
file size	with MAZE	without MAZE
50KB	12.170	3.004
100KB	12.158	2.994
1MB	12.213	3.035
100MB	12.309	4.049
500MB	16.579	8.164
1GB	23.445	17.481

Table 1: Average time (rounded to the nearest thousandths) to store and retrieve files with and without MAZE (varying the file sizes)

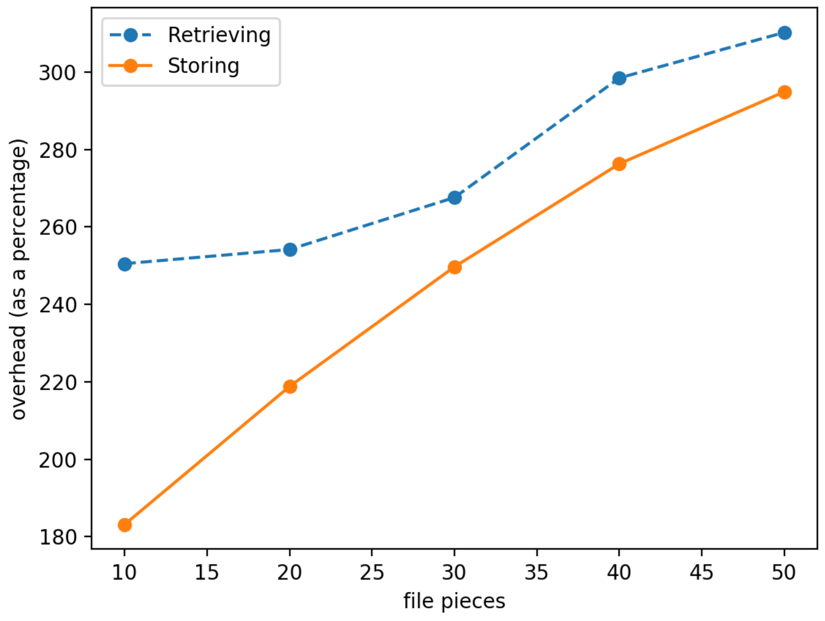

Figure 7: Overhead of Storing and Retrieving File Pieces with MAZE (varying file pieces/ tunnels)

Average Storing Time (secs)		
file pieces	with MAZE	without MAZE
10	12.420	4.387
20	24.852	7.796
30	39.212	11.213
40	54.933	14.598
50	72.13	18.265

Average Retrieval Time (secs)		
file pieces	with MAZE	without MAZE
10	12.345	3.171
20	24.891	7.028
30	38.925	10.589
40	56.163	14.094
50	71.941	17.536

Table 2: Average time (rounded to the nearest thousandths) to store and retrieve files with and without MAZE (varying the number of file pieces / tunnels)

4.4 Security Evaluation

In the system model, we discussed that attacker agents can gain control over a node in the MAZE system. From that node, however, since the attacker agent does not have the schedule, it does not know what port it needs to connect to and therefore resorts to brute-force connections to every possible port. Moreover, because the tunnels are one-use only, once a *client program* traverses through a tunnel, the attacker can no longer use that tunnel. As such, since the *client program* has the schedule, they are able traverse through the tunnel first, rendering it unusable for an attacker. Once an attacker reaches a node it is only a matter of time until the attacker agent is killed by the periodic refresh.

To evaluate the security of MAZE we developed an attacker agent that attempts to connect to every possible unreserved TCP port (1,024 to 65,535). The average time of ten executions was 343.937 seconds or 5.732285 minutes. Therefore, if we configure the periodic refresh to be less than half of 343.937 seconds, an attacker agent should be killed before successfully finding an available unused tunnel.

5.0 Conclusion

Cloud storage systems are rising in popularity for both business and personal use due to the increase in online collaboration and cloud storage's promise to be accessible anytime, anywhere. However, the static nature of cloud systems enables attacker to thoroughly study the system and its vulnerabilities without fear of the system configuration periodically changing. To address the aforementioned, we propose MAZE, a cloud storage system that employs Moving Target Defense and uses tunneling to prevent attackers from freely hopping between machines and retrieving desired files. The experiment results and analysis demonstrate that MAZE is suitable for transferring large files because as the file size increases, the relative overhead of MAZE decreases. In Section 5.1, we discuss limitations of MAZE and possible solutions thereof. Then, in Section 5.2, we detail future directions for MAZE.

5.1 Limitations

Legitimate unreserved TCP port numbers range from 1,024 to 65,535. Due to the limited amount of port numbers, it is possible that collisions may occur. A collision occurs when MAZE attempts to establish a tunnel from the same source end-point to a different destination end-point. In other words, if a server is already listening at a certain source port number and forwarding connections to a certain destination end-point, it cannot be asked to listen at the same source port number but forward the connection to a different destination end-point. Attempting to do the aforementioned will result in failure to establish the tunnel. Collision can occur in two situations: in schedule generation for one client or due to concurrent clients. Collisions occur in schedule generation for one client when, in the same schedule, two tunnels to have the same source port number due to the first two bytes of the password hash output being the same. Collisions occur in concurrent clients when the schedule of multiple clients contain the same source port numbers but different destinations. To address this shortcoming, we can keep a list of open port numbers (i.e., port numbers not used as tunnel end-points) and once there is a collision we could use linear probing. However, if all ports are being used, we can keep a queue of tunnels that want to established at a certain source port number. Once a tunnel in the queue is used, we can establish the next tunnel in the queue. The drawback of this approach, is that a user may experience performance slowdown because they have to wait for previous tunnels to be used and for new ones to be established.

The CAP Theorem [23], proposed by Eric Brewer, states that a distributed system can only guarantee two of the three characteristics: consistency (all nodes have the same data in the same state), availability (every request receives a response), and partition tolerance (system does not fail with some malfunctioning nodes). MAZE provides consistency because tunnels are one-use only. For example, if a user is storing a file with their password and at the same time attempts to read the file, the reading of the file has to wait for all the tunnels from storing to complete before trying to read. In addition, we can support partition tolerance in MAZE by employing a technique called Information Dispersal Algorithm (IDA) [40]. IDA splits data into f different pieces such that only k of the pieces have to be retrieved in order to regenerate the original data, where $k \leq f$. Therefore, if we split a file into pieces using IDA, and distribute them to the appropriate nodes, even if some nodes are down we can still regenerate the entire file. However, MAZE cannot guarantee availability because of tunnel collisions. Storing and retrieving a file may require more time if other users opened tunnels at the same source end-points.

5.2 Future Work

In this work we focused on designing, implementing and testing the reactive MAZE approach. However, due to the limited number of Amazon EC2 machines used in the evaluation, some features of MAZE went untested. In our implementation, we only had one gateway machine. The drawback of having only one gateway node is that it is highly susceptible to DDoS attacks, which can significantly slow down the system. In future work, we would like to explore techniques to protect gateway nodes from possible attacks. For instance, we could use IP-hopping to constantly randomize the IP-address of the gateway such that it is only accessible to authorized users. In Figure 2, many features of the service-side were not implemented, such as the black-hole nodes, hardening of nodes, machine identity obfuscation, and proactive secret sharing. In future work, we would like to build out these features and evaluate their performance and security.

In addition, we want to build an attack simulator for performing security evaluation experiments on the MAZE system. For example, more intelligent attackers would attempt to determine the end-points of tunnels through traffic analysis. Furthermore, we would like to explore sophisticated open-source network scanning tools, such as Nmap [36], that can scan ports in parallel, and examine whether a client can traverse through the tunnels before even Nmap can find the tunnels. We also plan to implement the proactive MAZE approach and compare its performance and security against the reactive approach. The proactive MAZE approach can be implemented by weaving in components from the reactive approach.

Bibliography

[1] Amazon ec2 instance types. `https://aws.amazon.com/ec2/instance-types`. Accessed: 2020-10-15.

[2] Block storage. `https://www.ibm.com/cloud/learn/block-storage`. Accessed: 2020-10-15.

[3] Block storage. `https://aws.amazon.com/s3`. Accessed: 2020-10-15.

[4] Cloud storage. `https://www.ibm.com/cloud/learn/cloud-storage`. Accessed: 2020-10-15.

[5] Cryptomove. `https://www.cryptomove.com`. Accessed: 2020-10-15.

[6] Data deposit box reports on data security incident. `https://finance.yahoo.com/news/data-deposit-box-reports-data-200000550.html`. Accessed: 2020-10-15.

[7] File storage, block storage, or object storage? `https://www.redhat.com/en/topics/data-storage/file-block-object-storage`. Accessed: 2020-10-15.

[8] Forecast number of personal cloud storage consumers/users worldwide from 2014 to 2020. `https://www.statista.com/statistics/499558/worldwide-personal-cloud-storage-users/`. Accessed: 2020-10-15.

[9] Human error often the culprit in cloud data breaches. `https://www.wsj.com/articles/human-error-often-the-culprit-in-cloud-data-breaches-11566898203`. Accessed: 2020-10-15.

[10] libssh. `https://www.libssh.org`. Accessed: 2020-10-15.

[11] Nexitech. `https://nexitech.com`. Accessed: 2020-10-15.

[12] Openssl. `https://www.openssl.org`. Accessed: 2020-10-15.

[13] Ssh tunneling explained. `https://chamibuddhika.wordpress.com/2012/03/21/ssh-tunnelling-explained/`. Accessed: 2020-10-15.

[14] What is cloud storage? `https://aws.amazon.com/what-is-cloud-storage`. Accessed: 2020-10-15.

[15] What is object storage? `https://www.netapp.com/data-storage/storagegrid/what-is-object-storage`. Accessed: 2020-10-15.

[16] Why organizations are moving to the cloud. `https://www2.deloitte.com/us/en/insights/industry/technology/why-organizations-are-moving-to-the-cloud.html`. Accessed: 2020-10-15.

[17] Noor O Ahmed and Bharat Bhargava. Mayflies: A moving target defense framework for distributed systems. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, pages 59–64, 2016.

[18] Fadi Aloul, Syed Zahidi, and Wassim El-Hajj. Two factor authentication using mobile phones. In *2009 IEEE/ACS International Conference on Computer Systems and Applications*, pages 641–644. IEEE, 2009.

[19] Michael Atighetchi, Partha Pal, Franklin Webber, and Christopher Jones. Adaptive use of network-centric mechanisms in cyber-defense. In *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003.*, pages 183–192. IEEE, 2003.

[20] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[21] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567. IEEE, 2012.

[22] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Provably secure authenticated group diffie-hellman key exchange. *ACM Transactions on Information and System Security (TISSEC)*, 10(3):10–es, 2007.

[23] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[24] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[25] Te-Shun Chou. Security threats on cloud computing vulnerabilities. *International Journal of Computer Science & Information Technology*, 5(3):79, 2013.

[26] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[28] Qi Duan, Yongge Wang, Fadi Mohsen, and Ehab Al-Shaer. Private and anonymous data storage and distribution in cloud. In *2013 IEEE International Conference on Services Computing*, pages 264–271. IEEE, 2013.

[29] Daniel Fraunholz, Daniel Krohmer, Simon Duque Anton, and Hans Dieter Schotten. Catch me if you can: Dynamic concealment of network entities. In *Proceedings of the 5th ACM Workshop on Moving Target Defense*, pages 31–39, 2018.

[30] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, volume 1, pages 13–15. IEEE, 2006.

[31] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Annual International Cryptology Conference*, pages 339–352. Springer, 1995.

[32] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132, 2012.

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[34] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[36] Gordon Fyodor Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning.* Insecure, 2009.

[37] Hamed Okhravi, William W Streilein, and Kevin S Bauer. Moving target techniques: Leveraging uncertainty for cyberdefense. Technical report, MIT Lincoln Laboratory Lexington United States, 2015.

[38] David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, 1988.

[39] Jon Postel. Telnet protocol specification. In *RFC 854, ISI.* Citeseer, 1983.

[40] Michael O Rabin. The information dispersal algorithm and its applications. In *Sequences*, pages 406–419. Springer, 1990.

[41] Michael G Reed, Paul F Syverson, and David M Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected areas in Communications*, 16(4):482–494, 1998.

[42] David A Schultz, Barbara Liskov, and Moses Liskov. Mobile proactive secret sharing. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 458–458, 2008.

[43] Sailik Sengupta, Ankur Chowdhary, Abdulhakim Sabur, Adel Alshamrani, Dijiang Huang, and Subbarao Kambhampati. A survey of moving target defenses for network security. *IEEE Communications Surveys & Tutorials*, 2020.

[44] Richard Skowyra, Kevin Bauer, Veer Dedhia, and Hamed Okhravi. Have no phear: Networks without identifiers. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, pages 3–14, 2016.

[45] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[46] Salvatore J Stolfo, Malek Ben Salem, and Angelos D Keromytis. Fog computing: Mitigating insider data theft attacks in the cloud. In *2012 IEEE symposium on security and privacy workshops*, pages 125–128. IEEE, 2012.

[47] Ding Wang and Ping Wang. On the anonymity of two-factor authentication schemes for wireless sensor networks: Attacks, principle and solutions. *Computer Networks*, 73:41–57, 2014.

[48] Tianyi Xing, Dijiang Huang, Le Xu, Chun-Jen Chung, and Pankaj Khatkar. Snortflow: A openflow-based intrusion prevention system in cloud environment. In *2013 second GENI research and educational experiment workshop*, pages 89–92. IEEE, 2013.

[49] Tatu Ylonen. Ssh–secure login connections over the internet. In *Proceedings of the 6th USENIX Security Symposium*, volume 37, 1996.

[50] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. Apss: Proactive secret sharing in asynchronous systems. *ACM transactions on information and system security (TISSEC)*, 8(3):259–286, 2005.