Remote Filesystem Event Notification and Processing for Distributed Systems

Kushal Thapa[†‡], Vinay Lokesh*[#], Stan McClellan[†§]
[†]*Ingram School of Engineering*
**Dept. of Computer Science*
Texas State University
San Marcos, TX, USA
e-mail: [‡]k_t260@txstate.edu, [#]v_v183@txstate.edu, [§]stan.mcclellan@txstate.edu

***Abstract*— Monitoring and safeguarding the integrity of files in local filesystems is imperative to computer systems for many purposes, including system security, data acquisition, and other processing requirements. However, distributed systems may have difficulty in monitoring remote filesystem events even though asynchronous notification of filesystem events on a remote, resource-constrained device can be very useful. This paper discusses several aspects of monitoring remote filesystem events in a loosely-coupled and distributed architecture. This paper investigates a simple and scalable technique to enable secure remote file system monitoring using existing Operating System resident tools with minimum overhead.**

Keywords*— *Secure Remote Filesystem Monitoring; Firewall; Distributed Architecture; Secure Network Communication; SSH; Secure Shell Protocol; Filesystem.

I. INTRODUCTION

Most modern computer systems incorporate local storage containing files associated with user data, application data, and other important data, such as trade secrets and passwords [1]. The vast majority of attacks to such classified files are well known issues in day-to-day operations so it is vital to ensure the integrity of the file system. There are two general approaches to monitoring filesystems for unauthorized access: (a) Hash-based file integrity (b) Real-time file integrity. The hash-based approach is to scan critical files on systems on a regular schedule, detecting changes by comparing the current file hash to the previous version. In a real-time approach, the information is provided on not just file changes, but also on all the file read, write, and create events so determining a violation becomes much simpler [2].

Although most modern computer systems have several tools that are capable of tracking local file system events, it becomes much more complex to monitor file system events remotely from a central location [3].

Each system in distributed architecture is typically capable of monitoring filesystem events, such as creation, deletion, and changes in local files. This can be performed by tools like inotify [4], kqueue [5], FSEvent [6], direvent [7] etc. However, these tools inherently lack the ability to monitor remote filesystems. Tools like Secure Shell Protocol Filesystem (SSHFS) [8][9] allow a user to mount remote directories into the local system; however, file monitoring is not possible with SSHFS. Asynchronous notification of filesystem events on a remote, resource-constrained devices can be very useful, particularly in distributed acquisition architectures and other scenarios where data is processed asynchronously.

In distributed architecture, one of the complexities introduced by Internet-based (IP) networking is a firewall [10][11]. Firewalls are vital for network security; however, the presence of an intervening firewall can make communication between distributed systems much more complex. Many networking solutions and architectures allow the users to circumvent certain firewall restrictions, thus increasing complexity while introducing security risks. Here, we leverage the well-known network architecture where an Internet-reachable system acts as a middleman to establish a secure, bidirectional network connection between firewalled devices. This approach is not new, however, comprehensive analysis of various parameters is difficult to obtain, so we provide some results and discussion regarding the various configuration options and performance of this architecture.

In Section II of this paper, we describe various tools that are generally used to monitor local filesystem events. We also briefly discuss about Secure Shell Protocol Filesystem (SSHFS) [9] and Secure Shell Protocol (SSH) [12]. Section III presents our approach to an experiment, which presents different network architectures and usage of those architectures with SSHFS and SSH. In Section IV we evaluate local filesystem monitoring and network communications using metrics for (a) complexity, (b) portability, and (c) efficiency/speed. We conclude the paper in Section V describing the overall summary of our experiment.

II. BACKGROUND AND RELATED WORK

Inotify [4] is a filesystem event notification tool for Linux operating systems. This tool allows user to add an automated watch to a file or directory which can monitor certain filesystem event(s) (for example: open, write, modify, close, etc.). When those events occur on the file or the folders being watched, this tool provides asynchronous, event-driven notification to the user for user interaction. Inotify-tools provides command-line interface to inotify [13][14]. Inotifywait is a shell utility included in inotify-tools that waits for changes to files or folders, and outputs the description of these changes when made. There are many options available for this command [15] through which the user can specify the target for the watch, the nature of the watch and the format of the output. These options, along with the fact that multiple watches can be made simultaneously, makes this tool very easy-to-configure, user-friendly and scalable. However, inotify is a kernel feature, which only monitors local file system events, and thus, remote filesystem events, not implemented in the local kernel, are not registered by inotify [13][15].

iWatch [16] is a kernel feature written as a Perl wrapper for inotify to monitor changes in specific directories or files, sending alarms to system administrator in real-time. iWatch can run as a daemon, as well as via the command-line. The daemon mode uses an Extensible Markup Language (XML) configuration file to register a list of directories and files to monitor. The command line mode will run without a configuration file. In the XML configuration file, each target can have an individual email contact point. This contact

 ISBN: 978-1-61208-835-8

point allows an email notification for any modification in the monitored targets.

kqueue [3][5] is an event notification interface in FreeBSD, supported by other operating systems such as NetBSD, OpenBSD, DragonflyBSD, and macOS. kqueue monitor demands a file descriptor to be opened for every file which is being watched hence restricting its application to very large file systems [5]. kqueue does not provide direct support for generic events such as 'create' for files and its Application Programming Interface (API) is designed with higher dependency on the local kernel, limiting the ability to work asynchronously with remote file system monitoring.

Filesystem Notification Events (**FSEvents)** [6] is an event notification API designed for macOS. FSEvents is a kernel feature and has a device file called /dev/fsevents. It follows a simple process where all the primal event notifications are passed to the userspace through this device file. The event stream it is then filtered by a daemon to publish notifications. The macOS version 10.7(lion) added the capability to watch filesystem [6]. The FSEvents monitor is not constrained by requiring unique watchers and thus scales well for large systems with huge number of directories. Although FSEvents can monitor a directory that is within a remote mounted volume and provides a callback for local changes, it cannot detect changes made by users on other machines.

FileSystemWatcher [17] is a specific class in the System.IO namespace, which is used to monitor and detect file system changes in Windows. It triggers events for every change that appear in file or directory which is being watched. It generates a new instance for FileSystemWatcher with arguments required to specify the directory and type of files which needs to be monitored, and a buffer into which all the file changes are written. The kernel then reports file changes by writing to that buffer. This suffers event loss when large number of changes are pushed into the buffer. One of the drawbacks of filesystem watcher is that it can only establish a watch to monitor directories, not files. To monitor a file, its parent directory must be watched in order to receive change events for all of the directory's children.

Tripwire is one of well-known file integrity program[18][19]. Tripwire was essentially built as a strong file integrity checker for Unix systems. The original Tripwire was termed as Academic Source Release (ASR) which has features such as strong set of supported hash functions, the power to examine file attributes, and a good configuration. It was a freely available program with reporting capabilities limited to results displayed only on the terminal screen. It also lacks database protection and verification capability.

The **Python Watchdog** module [20] is used to monitor file system events. Python Watchdog has a standard API for developers to select and deploy a monitor. Facebook's Watchman [21] is similar to Python's Watchdog module which also provides a similar interface for initiating different monitors. However, both these tools are operating system dependent making it infeasible for remote file monitoring and processing.

Direvent, like inotify, is a filesystem monitoring tool. However, direvent works in GNU/Linux, BSD and Darwin (Mac OS X) systems [7]. This allows for uniformity, and possibly integration of file monitoring processes, across diverse systems in a network. The files and directories to be watched, along with their corresponding target event, are specified in the direvent configuration file with 'watcher' statements. Filesystem events can be divided into two major groups. (a) system-dependent events that are specific for each kernel interface (b) generic events that do not depend on the underlying system. They provide a higher level of abstraction and make it possible to port configurations between various systems and architectures. However, direvent relies on the local event monitoring Application Programming Interface (API) provided by kernel [22]. As a result, it is not natively compatible with remote file system monitoring. When compiling with Berkeley Software Distribution (BSD) systems direvent uses kqueue – another kernel event notification mechanism.

Secure Shell Protocol Filesystem (**SSHFS)** [9] is a file system in user space (FUSE) that uses the SSH File Transfer Protocol (SFTP) to locally mount a remote file system. The mounted file systems can be accessed and used the same way a local file system is, both from the command line or using other tools. Unfortunately, inotify is not aware of filesystem changes on an SSHFS mount which are initiated from the remote end of the link.

Secure Shell Protocol (SSH) [12][23] is a secure network communication paradigm that operates at the Open System Interconnection (OSI) session level. All session data transferred through an SSH connection is transparently encrypted. This encryption is transparent, which means it gets decrypted by SSH client daemon at the specified destination, and thus, users do not have to deal with decrypted data, because of its utility and security features, SSH is widely used for remote system management tasks and can incorporate multiple use-cases, including forwarding graphical sessions, automating "jump" behavior to access systems behind firewalls, and so on.

a) Multiplexing

SSH has the ability to carry multiple sessions over single TCP connection via "multiplexing". One of the benefits of multiplexing is that it speeds up certain operations that utilize an SSH session.

b) Reverse Port Forwarding

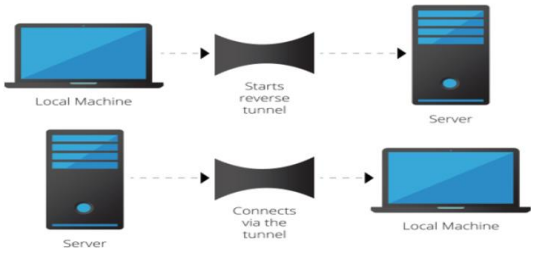

Figure 1. Working of SSH Reverse Port Forwarding [24].

Figure 1 shows the working of Reverse SSH port forwarding. It is a technique through which systems that are behind a firewall can be accessed from the outside world. With this technique, a port on a remote machine can be forwarded to the local machine while still initiating the tunnel from the local machine. This works by listening to the port on the remote side, and whenever a connection is made to this port, the connection is forwarded over the secure channel to the host port from the local machine.

III. APPROACH

Our evaluation of these several alternatives consists of two parts: (a) building a simple and secure network architecture to communicate between devices, and (b) using that architecture to test remote, asynchronous filesystem

monitoring. The network architecture as well as file monitoring test setup that is designed and described in this paper is simplified to only two devices, however, this system is scalable to include any number of devices.

A. Network Architecture

Network communication is one of the primary challenges when creating a system where interconnectivity between devices is required. For this research, we have chosen an SSH based network architecture which leverages an IP reachable system acting as a "jump server" to establish communication between devices behind their individual firewall. Figure 2 shows the basic components of this three-prong architecture.

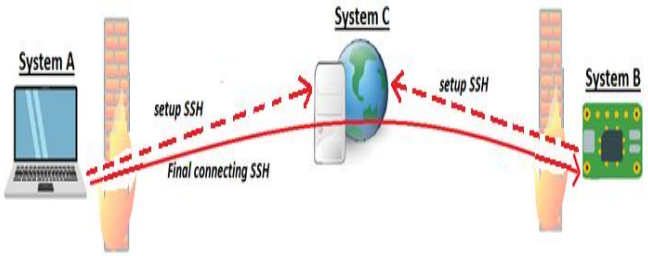

Figure 2. Representation of three-prong architecture.

As shown in Figure 2, system A is used to control remote devices behind a firewall, represented by System B. Both A and B are behind network firewalls, so a direct SSH connection cannot be made from A to B or vice versa. Thus, the need of system C, which is an open IP reachable server. Both A and B can communicate with C because firewalls allow outgoing connections. Using port forwarding, an SSH tunnel can be made from A to B via C. In this architecture, system C acts merely as SSH reflector, and no special configuration is necessary.

One of the main advantages of this architecture over other architectures and network solutions that circumvent the firewall restrictions is its simple configuration. The configuration for establishing this network begins when the first host (A) which creates an SSH connection to C. Similarly another host (B) also creates an SSH connection to C. Finally, to create connection between A and B ports from A and B are forwarded to a common port in C, thereby creating a continuous tunnel from A to B. Two other simpler network architectures (referenced as Arch-0 and Arch-2 in this paper) were built using the components of our primary architecture (Arch-1) to compare the results. The design of these two architectures are shown in Figure 3 and Figure 4, respectively.

i. Arch-0

Figure 3. Representation of systems and network connection behind a common firewall.

As shown in Figure 3, there is a common architecture between systems in the same network where there is no firewall between hosts. System A can SSH directly into system B. This architecture is used as a control and as a baseline in this study.

ii. Arch-2

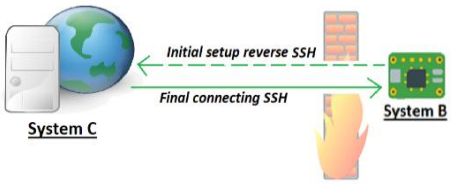

Figure 4. Representation of components, security zones and connections in a client-server architecture.

Figure 4 shows the network architecture that is used more commonly as an alternative for our three-prong architecture. Here, the peripheral devices indicated by system B are controlled directly by an open IP reachable server. The system C cannot directly form an SSH connection to B because of the presence of a firewall. Thus, B needs to initiate a special SSH tunnel which is used by C to form a reverse SSH channel back to B.

In this architecture, C acts as the control center. Thus, the main difference between our three-prong architecture and this architecture lies in the role of the server. In this architecture, the server's resources are heavily utilized. This can be advantageous if the server is powerful. However, since the server is internet reachable, it can pose security risks if its configuration is too complex. The main advantage of the three-prong architecture is in "hidden complexity" because the control unit is protected by firewall, so the exposed attack surface is minimized.

To evaluate these architectures, we used multiplexed SSH combined with a simple timing test which is shown using a vertical line diagram in Figure 5. This test program sends a simple UNIX command via regular and multiplexed SSH tunnels, and records the time taken to send, execute and receive the output of this command. To account for the variability of network speed at different times, this program was run every hour of every day for about a month. The results of this experiment are compiled in Section IV.

Figure 5 shows the three network architectures of interest, Arch-0, Arch-1 and Arch-2. In this case, Arch-1 indicated by red is a "three prong" network architecture, whereas Arch-0 shown in Figure 3 and Arch-2 shown in Figure 4. indicated blue and green respectively, are two other simpler architectures built using the primary Arch-1 architecture. The bold lines represent multiplexed connections of these three architectures while the thin lines represent non-multiplexed connections. The dashed lines represent connections necessary for their corresponding architecture. For our timing test, both system A and B are on the same network. In case of Arch-1 the connections goes through System C hence the firewalls can be separate.

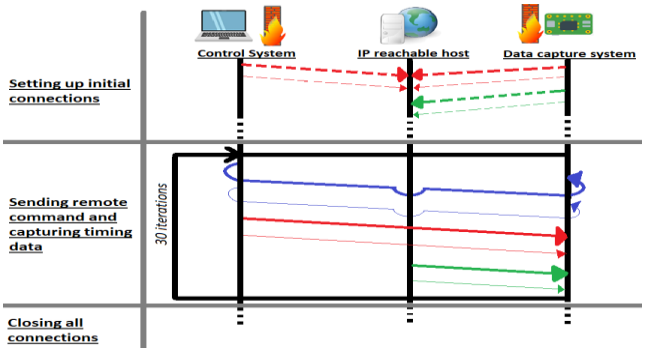

Figure 5. Vertical line diagram to represent three network architecture, Arch-0, Arch-1 and Arch 2.

 ISBN: 978-1-61208-835-8

B. Remote Filesystem Monitoring

Application specific file monitoring is useful in detecting certain changes and responding to those changes. However, tools such as inotify and direvent lack capability to monitor files and directories of other devices in the network. To overcome this challenge, we took two approaches:

1) Using SSHFS

Secure Shell Protocol Filesystem SSHFS [11] enables the user to mount remote filesystem in the local filesystem. The user can access and monitor the mounted files and directories manually. To automate this monitoring process, we tried coupling SSHFS with inotify by mounting a remote filesystem and monitoring changes on it.

In general terms, inotify is not aware of filesystem changes on an SSHFS mount which are initiated from the remote end of the link. This is because SSHFS is built on top of SFTP; hence, it is a client view of the remote filesystem and does not export filesystem events from the remote system.

2) Using SSH

Using Secure Shell Protocol (SSH) [16], we devised a two-step method for remote filesystem monitoring which is simple, intuitive and scalable, and utilizes the light pre-existing OS-resident tools. The target file or directory in the local system is monitored using a tool such as inotifywait.

Then, using the event registration of the monitored target as a trigger, a command is sent to the other end of the channel using SSH. Such SSH appended commands can be a simple OS command or another trigger to a script, and thus can be easily modified according to application requirements.

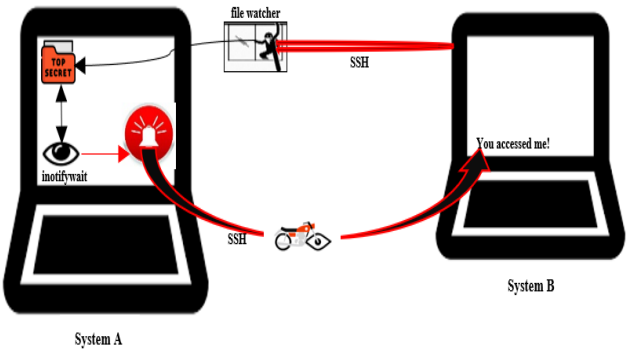

Figure 6. Remote Filesystem monitoring using SSH and inotify tools.

As shown in Figure 6, an inotifywait is issued on a top secret directory in System A, so whenever a filesystem event occurs in that watched section of the filesystem, the event is transmitted to the remote monitoring configuration on System B along with a timestamp of the event.

IV. RESULTS

We evaluate different network architectures mentioned in Section III along with a timing data as shown in Figure 7 and Figure 8. Multiplexed SSH connections significantly reduce connection time because the TCP handshake and keying interaction to set up the SSH session is already performed, and is being re-used and efficiently. From Figure 7 and Figure 8, it is clear that Arch-0 exhibits the fastest communication time in both multiplexed and non-multiplexed architectures, which is reasonable since both systems are on the same network, with no firewall.

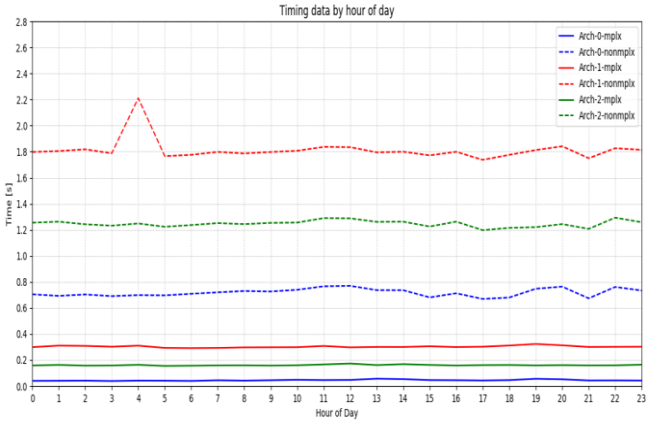

Figure 7. Timing data by hour of the day.

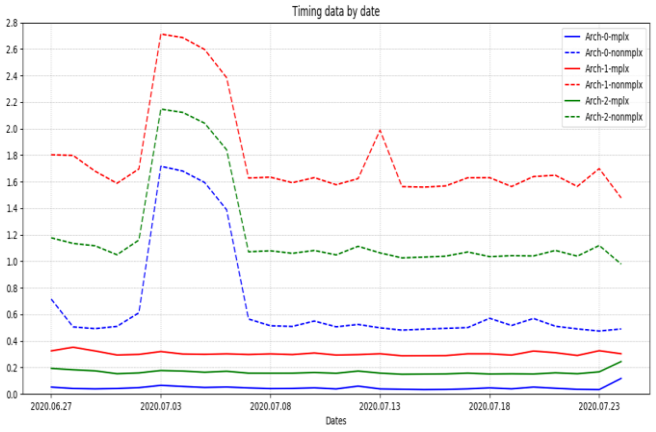

Figure 8. Timing data by date.

Also from Figure 7 and Figure 8, the multiplexed connection of Arch-1 recorded lower time than non-multiplexed version of Arch-0, which is interesting because in Arch-1 a firewall exists between systems, so TCP/keying lags are substantial, even for systems on the same network. The non-multiplexed connections exhibit substantial random latencies due to multiple network transits of TCP handshakes and keying interchanges, whereas corresponding points in the multiplexed configuration do not exhibit the same issues due to the more efficient re-use of multiplexed connections. The performance of multiplexed connections is much more consistent.

V. CONCLUSION

This paper discusses the use of various filesystem monitoring tools which support local file system monitoring, but inherently lack ability to monitor remote filesystems. In distributed and loosely coupled architectures, monitoring of filesystem events on remote systems, possibly behind firewalls, can have important application-layer benefits and utility. To examine the performance of various system configurations, we evaluate network architectures with both multiplexing and non-multiplexing techniques, concluding that a simple and scalable technique using multiplexed SSH connections and inotify tools enables secure remote file system monitoring with minimum overhead. By recording timing of filesystem events on each of these network architectures, we note that multiplexed SSH connections are consistent, and much more efficient than other methods, even with complex distributed architectures involving exposed systems, multiple firewalls, and “three prong” SSH tunnels.

 ISBN: 978-1-61208-835-8

REFERENCES

[1] K. P. Suresh, U. Himanshu, and L. Leonel, “File integrity monitoring tools: issues, challenges, and solutions,” [Online]. Available from: https://onlinelibrary.wiley.com/doi/epdf/10.1002/cpe.5825 [retrieved April 2021]

[2] S. Evangelou, “How to verify File Integrity using hash algorithms in Powershell,” [Online]. Available from: https://stefanos.cloud/blog/kb/how-to-verify-file-integrity-using-hash-algorithms-in-powershell/ [retrieved April 2021]

[3] A. K. Paul et al. "FSMonitor: Scalable File System Monitoring for Arbitary Storage Systems," in IEEE International Conference on Cluster Computing (CLUSTER) Cluster Computing (CLUSTER), Albuquerque, 2019, pp. 1-11.

[4] R. Love, "Kernel Korner - Intro to Inotify," Linux Journal, 2005.

[5] J. Lemon, "Kqueue – A generic and scalable event notification facility," in Proceedings of the FREENIX Track: USENIX Annual Technical Conference, Boston, 2001, pp. 1-14

[6] Apple, "File System Events Programming Guide," 13 December 2012.[Online]. Available from: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40005289-CH1-SW1. [retrieved April 2021].

[7] S. Poznyakoff, "GNU Direvent," 13 July 2019. [Online]. Available from: https://www.gnu.org.ua/software/direvent/manual/direvent.html. [retrieved 06 August 2020].

[8] N. Rath, "libfuse/ sshfs," [Online]. Available from: https://github.com/libfuse/sshfs. [retrieved March 2021].

[9] M. E. Hoskins, "SSHFS: Super Easy File Access over SSH," Linux Journal, 2006, pp. 1-6.

[10] J. R. Vacca and S. Ellis, Firewalls : jumpstart for network and systems administrators, Elsevier Digital, 2005.

[11] W. Noonan and I. Dubrawsky, Firewall fundamentals, Indianapolis, Cisco, 2006.

[12] D. J. Barrett, R. G. Byrnes, and R. E. Silverman, "Introduction to SSH," in SSH, The Secure Shell: The Definitive Guide : The Definitive Guide, O’Reilly Media, Inc., 2011, pp. 1-15.

[13] A. Schwab, "inotify-tools," [Online]. Available from: https://github.com/inotify-tools/inotify-tools. [retrieved April 2021].

[14] C. Fischer, "Linux Filesystem Events with inotify," Linux Journal, 2018. pp. 1-17.

[15] R. McGovern, "inotifywait(1) - Linux man page," [Online]. Available from: https://linux.die.net/man/1/inotifywait. [retrieved April 2021].

[16] C. Wirawan, “iwatch - realtime filesystem monitoring program using inotify,” Ubuntu man page [retrieved 25 March 2021]

[17] Microsoft. FileSystemWatcher. https://docs.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher?redirectedfrom=MSDN&view=netframework-4.7.2, 2010. [retrieved April 2021].

[18] D. Armstrong, “An introduction to file integrity checking on unixsystems,” [Online] Available from: https://www.giac.org/paper/gcux/188/introduction-file-integrity-checking-unix-systems/104739. [retrieved April 2021].

[19] G. Kim and E. H. Spafford, “The Design and Implementation of Tripwire A File System Integrity Checker,” In:2nd ACM Conference on Computer and Communications Security, pp. 18–29. ACM, Fairfax, VA, USA (1994)

[20] Python Watchdog. [Online]. Available from: https://pypi.org/project/watchdog/, 2010. [retrieved April 2021].

[21] Facebook Watchman. “A file watching service. https://facebook.github.io/watchman/, 2015”. [retrieved April 2021].

[22] M. Kerrisk, "Filesystem notification, part 2: A deeper investigationof inotify," 14 July 2014. [Online]. Available from: https://lwn.net/Articles/605128/. [retrieved April 2021].

[23] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture," RFC 4251, January 2006. [Online]. Available from: https://www.rfc-editor.org/info/rfc4251. [retrieved April 2021].

[24] J. Knafo, “What is reverse SSH Porting” Available from: https://blog.devolutions.net/2017/3/what-is-reverse-ssh-port-forwarding. [retrieved March 2021]

 ISBN: 978-1-61208-835-8