Tleilax: Porting Tor to Arrakis

A senior thesis submitted in partial fulfillment of the requirements for the
degree of
Bachelor of Science with Departmental Honors
in
Computer Engineering
at
The University of Washington

Sunjay Cauligi
June 12, 2015
Abstract

While the Tor onion routing service is a popular and useful tool for anonymity and censorship evasion on the World Wide Web, it suffers from significant slowdowns due to processing bottlenecks at each relay. Using Arrakis, an experimental operating system developed by the University of Washington, I seek to mitigate some of these bottlenecks.

This thesis explores the feasibility of porting Tor to Arrakis for the purposes of increasing the bandwidth of the global Tor network. It evaluates the current state of Tor relay bandwidth at individual nodes to determine the source of packet throttling. Following these results, it describes the design and implementation of modifications to the Arrakis source code to allow Tor to properly build and run.

Contents

1 Introduction 2

2 Dependencies 2
   2.1 Tor (“The Onion Router”) 2
   2.2 Arrakis 3

3 Initial Evaluation 4

4 Design and Implementation 5

5 Discussion 7
   5.1 Security Implications 7
   5.2 Further Steps 7

6 Related Work 8

7 Conclusion 8
1 Introduction

With the recent upswing of interest in anonymity and privacy online, more and more users are investigating how to protect themselves from mass surveillance and information-gathering agencies. Additionally, Internet users in countries with restrictive governments are seeking out more ways to evade censorship laws and access and participate in content online that they otherwise would not be able to. The most popular software for anonymization and censorship evasion, by far, is the Tor onion router, which serves over 2.5 million users per day. Unfortunately, the Tor network has hit its scaling limits and its resources are strained; most users experience slow load times for content as a tradeoff for their protection.

The limited speed of the Tor network also drives away many users that would otherwise benefit from the software, but are unwilling to sacrifice significant connection speed for network privacy. If the network were able to support more users at faster speeds, Tor would receive a significant rise in usage, benefiting all participants of the network.

I chose to tackle this problem by porting Tor to run on an operating system known as Arrakis. At a high level, Arrakis allows for userspace applications to access hardware-level operations without the overhead of kernel context switching. In my initial analysis, I show that Tor spends a significant portion of time processing I/O and waiting on the kernel. With the benefits provided by Arrakis, Tor can more efficiently process and send network packets, improving its bandwidth and speeding up the overall Tor network.

2 Dependencies

In order to better understand the technologies underlying the Tleilax project, a more detailed background of the two main systems is presented below.

2.1 Tor (“The Onion Router”)

The Tor software is a tool that allows for anonymous access to the Internet via the global Tor network. Users of Tor are given a degree of protection from external traffic analysis and network surveillance, thus ensuring their privacy while browsing the web. Tor is also a valuable tool for censorship evasion and working around government firewalls, and has been the subject of much scrutiny and research dedicated to both improvements and possible threats.

When a Tor client connects to the network, it selects a randomized path through multiple Tor relay nodes, ending in an exit relay, where traffic is sent to the destination (see Figure 1). When sending traffic, the Tor client encrypts a packet multiple times, using the public key of each relay along its path in turn. This way, each relay can only decode enough information to send the packet to the next relay along the circuit. Furthermore, all circuits between two relay nodes are multiplexed into a single TCP link. Since each
relay server is involved in multiple circuits from multiple clients, and since each packet is encrypted, it is very difficult for an adversary to determine the origin and final destination of packets in the network.

The directory of all available relay and exit nodes is served by the Tor directory authority servers. A new relay, upon starting up, will send information about its capabilities to a directory authority. At regular intervals, a “vote” is conducted on the state of all relay and exit nodes, to determine whether nodes are still up or to collect information on nodes reported for incorrect or malicious behavior.

2.2 Arrakis

Arrakis [11] is an operating system developed at the University of Washington for the purpose of making I/O operations much more efficient for many I/O-bound server applications. In a traditional operating system, the kernel is responsible for handling all I/O requests. Each call by a user mode application to send or receive data to and from the network must execute a system call, causing a context switch into the kernel (see Figure 2). Once the kernel finishes processing data, it switches back to user mode. Unfortunately, for I/O-bound applications, the overhead associated with constant context switches is severe.

Arrakis solves this problem by giving user mode applications direct access to virtualized I/O devices (see Figure 3). Using the same technology that commercial VMs use to provide
virtual network cards and disk drives, Arrakis creates virtual I/O hardware and exposes the virtualized hardware registers to applications. This method is safe, as applications only have access to the virtual device associated with them, and the physical hardware takes care of routing data to the proper virtual device. The kernel is no longer a part of the I/O path, and thus the cost of I/O operations is drastically reduced.

3 Initial Evaluation

For the initial analysis, I conducted a simple speed test on a private Tor network and also collected CPU profiling information to verify the degree of I/O processing bottleneck.

For the initial analysis, I constructed a private Tor network spanning four Ubuntu Linux-based servers on a 10Gb/s connection. The network consisted of three directory authorities and six relay/exit nodes, distributed evenly among three of the servers. The fourth server ran the Tor client. I used the iperf tool to collect network measurement data by setting up an iperf server on one machine in the network and an iperf client on a second. For the first set of tests, I connected the client directly to the server over the raw network, and was able to achieve almost full link saturation. For the second set of tests, I used the torsocks tool to direct the iperf traffic through the private Tor network. Over Tor, I was only able to achieve less than 2% of the available bandwidth (see Table 1).

To observe what was blocking the network from achieving full capacity, I used the operf analysis tool on a Tor relay while it was handling the iperf traffic load to see where the relays were spending their processing time (see Table 2). I found that relays spend almost 38.1% of their time processing I/O data, with a full 27.9% spent waiting on kernel operations. While cryptographic computation constituted a major portion of the overall CPU time, it is clear the relays still spend a significant amount of processing time on I/O operations.
<table>
<thead>
<tr>
<th>Raw (Mb/s)</th>
<th>Tor (Mb/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9134</td>
<td>168</td>
</tr>
<tr>
<td>9394</td>
<td>171</td>
</tr>
<tr>
<td>9103</td>
<td>169</td>
</tr>
<tr>
<td>9492</td>
<td>179</td>
</tr>
<tr>
<td>9155</td>
<td>171</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>9246.72</strong></td>
</tr>
</tbody>
</table>

Table 1: *iperf* data of raw network and Tor packet speed across five sessions on a 10Gb/s private network.

<table>
<thead>
<tr>
<th>CPU time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptography</td>
</tr>
<tr>
<td>Kernel (I/O)</td>
</tr>
<tr>
<td>Libraries (I/O)</td>
</tr>
<tr>
<td>Tor, misc.</td>
</tr>
</tbody>
</table>

Table 2: Summary of CPU profiles for a Tor relay under load.

Since Tor is spending so much time waiting on the kernel, having Tor running on the Arrakis operating system will allow it to handle more packets and increase its local bandwidth. With less overhead on the relay’s I/O path, each relay will be able to process more traffic. A number of these nodes, if deployed on the Tor network, can substantially increase the overall speed of the Tor network.

4 Design and Implementation

As Arrakis is still under active development, much of my work consisted of adding patches to the Arrakis codebase to allow Tor to run. I used various debugging tools to analyze failures in Tor when running on Arrakis, and implemented missing functionality to allow Tor to run properly.

Since Arrakis is a library OS, all operating system functionality is presented as a user-mode library, *libarrakis*, which is statically linked into each application. Arrakis presents a native interface for interacting with hardware that is different from the standard POSIX programming interface, but also provides a POSIX compatibility layer to ease porting existing applications. Tor depends on many POSIX functions that were not yet implemented in the compatibility layer, and so it would not initially build. Much of my initial work was spent filling in functions to either extend the POSIX compatibility to hook in to existing Arrakis functionality, or creating function stubs to handle unimplemented or unnecessary
functionality in a sane manner. Missing implementations included `mmap/munmap`, as well as various pthread and mutex attribute modifiers.

Once Tor was able to fully build, the next stage of my work involved implementing missing functionality in Arrakis so that Tor ran without crashing or otherwise failing. There were three main issues I encountered while setting up a functioning Tor relay:

- **Source of random bytes**: Tor uses OpenSSL for both cryptography and for establishing secure connections. OpenSSL requires a source of randomness, which Arrakis does not provide.

- **Real time clock**: The TLS handshake to set up a secure connection requires both parties to have synchronized clocks. Arrakis does not yet have a reliable method of determining the current actual time.

- **TCP stack**: Tor depends heavily on TCP layer communication for all of its inter-relay transport. The current TCP layer implementation in Arrakis was only designed to provide basic support for a very simple application.

The simplest and most efficient way to solve the missing source of random bytes was to patch OpenSSL function `ssleay_rand_bytes` to return success even if it cannot fill the entropy pool. For the purposes of this work, I am not concerned with strong cryptographically secure relay communication, as my focus for this research is on packet I/O speed.

As part of setting up a secure communication stream using the TLS protocol, the server and client must agree on the current time to within some boundary. From my cursory testing, this boundary spanned roughly 20 minutes. Since Arrakis cannot currently determine real time information from the hardware it is running on, I solved this problem by creating a script to insert a time literal into the code before each rebuild of the kernel. This way, each new build of the kernel updates the system time. Kernel builds are fast enough that this was not an issue during testing and development. However, this is clearly not an ideal solution, and there is work underway on implementing an interface between Arrakis and the hardware real time clock.

Most previous applications that were built on Arrakis were configured to use the UDP protocol rather than TCP, as the UDP interface is much simpler to implement. The existing TCP implementation in Arrakis is hardcoded for specific cases and makes many assumptions that do not reflect its use under Tor. Modifying the TCP stack to properly function under Tor’s usage scenario is still an ongoing area of work, but much progress has been made. Currently, the Tleilax relay is able to both communicate and receive information about the Tor network from the other nodes in the private network. However, I have yet to successfully construct a full Tor circuit through a Tleilax relay.
5 Discussion

With this research work, I gained much valuable insight both into the implementation of Tor and the work involved in setting up a Tor network, as well as the design of a library OS such as Arrakis. I was able to configure and deploy a function private Tor network, build binaries for an experimental kernel, and debug and modify an unfamiliar operating system running in a virtual machine.

As the work on completing a stable port for Tor on Arrakis is not yet complete, I cannot present analysis information on bandwidth and CPU profiling as I did with the Linux-based Tor nodes. Despite this, my research work led me to explore some important security considerations that could come about as a result of deploying faster Tor nodes on the public network. Additionally, I will discuss further work that can be done upon completion of the basic Tor port.

5.1 Security Implications

In this section I present some security and privacy risks associated with the deployment of Tleilax nodes on the public Tor network. Although this discussion raises some key points, further research exploring a more detailed analysis is invited.

In the current Tor implementation, though Tor clients select a randomized path through the network, the selection algorithm is weighted by the bandwidth of each relay [4]. Thus, a relay with twice the bandwidth of another node will be twice as likely selected for circuit paths. While this algorithm helps improve the perceived speed of the network for individual clients, it opens up a potential privacy risk: an adversarial relay operator need not operate nearly as many relays to capture a significant portion of traffic if they run high-bandwidth nodes. A relay operator running a standard Tor distribution would be competing with all other similarly operated Tor relays, and so would not receive a disproportionate share of traffic. However, if an adversarial operator is running Tleilax nodes, then they would receive a significant portion of all Tor traffic and would be able to carry out de-anonymization attacks to a higher degree of accuracy than previously investigated.

Furthermore, Tleilax nodes decrease the homogeneity of the overall Tor network. A client running a circuit through Tleilax nodes would have faster throughput on their circuit than a client running a circuit through standard Tor nodes. If an attacker can see the traffic patterns of a client as they enter the Tor network, and can also observe traffic patterns from Tor exit nodes, they would be able to differentiate clients on Tleilax paths from clients on standard Tor paths, decreasing the anonymity of clients on the Tor network.

5.2 Further Steps

Although some implementation shortcuts were taken in the initial port to Arrakis, a more robust approach is necessary before public deployment of Tleilax nodes. During the initial research phase, it was not necessary to focus on making Tleilax nodes robust, so quick
workarounds and stub implementations were used in many places where Arrakis’s functionality could be enhanced.

As mentioned earlier, Arrakis does not currently expose a way to gather sources of entropy. The OpenSSL implementation on Arrakis thus cannot generate a pool of random bytes necessary for most cryptographic operations. Arrakis would need to be patched to gather sources of entropy from the system in a method similar to the Linux kernel, and expose an interface that OpenSSL can use to read a cryptographically secure stream of random bytes. Additionally, getting Arrakis to interface properly with the real time clock hardware would remove the need to rebuild the kernel before every run, allowing Tor and other applications that depend on time synchronization to function properly. Finally, a more sound implementation of the TCP stack would greatly benefit this and future projects utilizing Arrakis.

Though the basic porting procedure will increase the bandwidth of a Tor relay due to the decrease in I/O overhead, much of the I/O processing path will still be going through the POSIX compatibility layer rather than communicating with the hardware through the Arrakis native interface. Potential further increases in performance can be gained by modifying Tor to use the native interface rather the original POSIX API. This work would require significant modifications to the Tor source code, but would be valuable in improving the state of the Tor network as well as potentially re-examining some of the design choices made in the implementation of Tor.

6 Related Work

Much previous work on improving the speed of Tor has focused on improving the overall Tor network rather than individual relays. Recent research has been conducted on path selection [12, 10], circuit scheduling [5, 6, 2], additional sockets [9, 3], congestion control [1, 6], and incentivized routing [8, 6]. Several of these improvements have already been incorporated into the Tor source distribution.

My work with Tleilax is most similar to the KIST paper [7], which also focuses on improving local bandwidth at each relay. KIST tackles the problem of kernel congestion by using better strategies to schedule writes on the kernel sockets. While KIST and Tleilax are solutions for the same problem, Tleilax focuses on improving the kernel while KIST focuses on improving the Tor application.

7 Conclusion

A faster Tor network, enabled by Tleilax nodes, would not only allow many more users to join the Tor network, but would also allow users to access the Internet privately at more useful speeds. With an increasing number of users interested in privacy, it is vital to
scale up the Tor network to both handle the increasing load, and to provide a better user experience.

Though a finished implementation of Tor running on Arrakis is still a work in progress, I have identified and worked through several major hurdles and have contributed improvements to the Arrakis library OS. Through my work, I have shown Tleilax is both a feasible goal and will improve the performance of Tor relays running on these machines.

References


