Analyzing the Impact of Software Configurations on Dynamic Code Coverage

by

Swapnil Kumbhar

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2023 by the
Graduate Supervisory Committee:

Yan Shoshitaishvili, Chair
Ruoyu Wang
Xusheng Xiao

ARIZONA STATE UNIVERSITY

May 2023

ABSTRACT

Large software tend to have a large number of configuration options that can be tuned to a varying degree in order to run the software in a specific way. These configuration options cause a change in the execution of the software, and therefore affect the code coverage of the software. This gives rise to the problem of understanding how much a certain configuration change affects the code coverage of the software in a measurable way. It also raises the question of effectively mapping code coverage to a configuration change. Solutions to these problems could give way to increasing efficiency in various areas of software security, like maximizing code coverage in fuzz testing and vulnerability identification in specific configurations.

In this work, I perform analyze widely used software, such as the database cache 'Redis' and web servers like 'Nginx' and 'Apache httpd'. I perform fuzz tests on multiple configurations of each of these software to measure the difference in code coverage caused by each configuration. I use Coverage Instrumentation to obtain traces for each software in their configurations, and then I analyze these traces to understand the configuration's impact on the software's code coverage.

In conclusion, I describe a method to measure how much code coverage differs for each configuration with respect to the default configuration of the software, and how certain configurations have a much larger difference in code coverage with respect to the default configuration than others, analyze the overlap in code coverage between the configurations and finally find the root causes of the differing code coverage.

DEDICATION

I'd like to dedicate my thesis to Dr. Yan Shoshitaishvili, for inspiring me into security research and for providing the opportunity to dig deeper into this topic. Thank you for creating pwn.college, I've learned countless things from that course that will stay with me for life, including wax on/wax off.

I'd also like to dedicate this thesis to Pooja. You were a rock throughout my time here, and I would not have made it without you.

ACKNOWLEDGMENTS

I'd first like to thank my advisor, Dr. Yan Shoshitaishvili, for giving me the chance to work with him. I've always wanted to do research at the academic level, and you're the reason I had the chance to do it.

I'm grateful to the amazing people at SEFCOM who have provided me with a lot of valuable inputs during our weekly calls, many of which helped shape this work. I'd like to extend my gratitude to Dr. Fish Wang, Dr. Tiffany Bao and Dr. Adam Doupé. Your guidance was crucial in shaping my research here.

Finally, I'd also like to extend my gratitude to my peer, Robert Wasinger. You've been there to answer my questions from day one. Thank you for all your help.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Software, historically, was built to be versatile. Consider one of the simplest programs that is part of the GNU Core Utilities[1] package: `date`. `date` happens to have eight configuration options. These configuration are of various types. There's options that act like a switch. `--universal` is an example of that, where the `date` program outputs the UTC time. The number of arguments or parameters here is not surprising. As requirements grow, a software is expected to perform its role in a variety of modes, which gives rise to configurations. As the number of configurations increase, the amount of code that the software needs also increases.

If we choose to look at a larger software, like a web server like Nginx, we see that the number of configurations is much higher. The core HTTP module, which takes care of the web server's core functionality to serve webpages, has 79 configuration options. This is the number of configurations for one HTTP module, of the 59 different HTTP modules that can be configured in Nginx. The other modules take care of many specific options like Gzip compression, HTTP Proxy, HTTP Headers, etc. The total number of configuration options that Nginx provides is considerably larger than that of a somewhat simple software like `date`.

The problem we face with such configuration options is this: It is hard to create a map of configuration changes to code coverage. To solve this problem, we need defined methodologies to quantitatively measure the effect a certain configuration option may have on code coverage. Such methods, apart from aiding in the creation of a map of

configuration changes to code coverage, could also be used to analyze the raw impact on code coverage caused by these configurations.

This work focuses on the analysis of the effect configuration changes may have on code coverage. First, I fuzz tested Redis, Nginx and Httpd, each run in about 10 different configuration options. I gathered trace information for each configuration and then contrasted them with the software's default configuration traces. The contrast helped me find how much each configuration differed in code coverage in terms of basic blocks, lines of code and functions.

Through this work, I contribute the methods I use for accurately analyzing and measuring the impact of configurations on dynamic code coverage, along with documenting the application of these methods on three major software – Redis[2], Nginx[3] and Apache httpd[4].

Chapter 2

BACKGROUND

I used a range of tools to uncover code coverage information for each software in their respective configurations. In order to fuzz test the software, I relied on AFL++[5]. AFL++ in the binary mode can fuzz software that hasn't been instrumented for fuzzing, essentially allowing a user to perform black-box fuzzing.

With modern compilers like `clang`, there are ways to add instrumentation to the compiled code. One such instrumentation is for measuring coverage and profiling. When an executable with such instrumentation is run, it creates a log of the details of its execution. This log, present in the form of a binary profile file, correlates with the program's source code. Code coverage information is also a part of this log. There are various ways to visualize and export this log. HTML is a common way to view code coverage, and a preferred way for many software developers. In my work, I used the `lcov` format, because of its ease of parsing and how much information an `lcov` analysis contains.

As a comparative to my main approach, I used a popular software emulator called QEMU[6] to extract traces. Internally, QEMU translates a target software's code to QEMU's own intermediate representation. Then, this intermediate representation is translated to the host machine's architecture. QEMU does this on the fly at the basic block level. A basic block in QEMU's intermediate language is called a Translation Block. QEMU is also used in AFL++ internally.

The software that I have analyzed in my work are used widely in the world. Redis is an open source data store, that is used commonly as a cache or a database. Nginx

is an open source web server. It is also, as of March 2023, world's most used web server[7]. In addition to being a web server, it also has capabilities to act as a reverse proxy and a load balancer. Apache httpd is also an open source web server that is widely used on the internet. It is the second most employed web server on the internet, with a share of 32.4%[7].

Chapter 3

EXPERIMENTATION SETUP

To run my experiments, there were three major areas that needed attention. An ideal experimentation setup needed to have software that is widely used, large and is highly configurable. Another important factor was that the software should work well with my fuzz testing setup.

In the following sections, I elaborate on each of these goals.

3.1 Choice of software

The software under test needs to be a widely used software. One reason for this is that a larger user base ensures that the software has complexities because of a high number of requirements. These complexities can be thought of as the number of slightly different ways a certain task can be achieved. If we consider a trivial example of the program `date`, it displays the current date and time, in the timezone of the machine. However, it can display time in Coordinated Universal Time (UTC) or in a different format like RFC 5322, which is used in E-Mails.

The complexity of the software also ties in with how large it is. There is a direct correlation between the size and complexity of software. A larger software, apart from adding complexity, also gives some level of confidence that the code for various configurations is loosely tied with each other. This makes it slightly easier to analyze the impact of a specific configuration on the overall code coverage. While a larger

software can have tight coupling, the sheer size of the software increases the possibility of a code coverage difference that can be reliably detected and measured.

The software needs to be a network service. While not a hard requirement for the purpose of my research, it did become a requirement specific to my fuzz testing setup. However, the requirement did not feel unrealistic, as many large software happen to be network services, like web servers, databases, load balancers, etc.

Finally, the software needs to be highly configurable. This mostly ensures that the software under test can be configured in various configurations that are relatively unrelated to each other, thus yielding reliable results.

3.2 Choice of Test Configurations

A Test Configuration is a software run in a specific configuration that differs from the default configuration. For the experiments, I chose configurations that could cause a varying degree of difference in code coverage, if contrasted with the default configurations. For configuration options that are mandatory, like the path to the root directory of a web server, the values are kept the same across all Test Configurations.

Among these Test Configurations, I chose a few configurations that I expected to make a considerable difference in code coverage. An example of this is having Transport Layer Security (TLS) enabled on the software for any incoming communication. Clearly, the flow of the software will change drastically in the early stages to establish encryption (TLS Handshake) and to decrypt any data that the connecting client has sent.

In contrast to the Test Configurations described above, I chose configurations that I expected to not diverge the code coverage by a lot. An example of this was configuration options like ‘Keep-alive Timeout’. My fuzz testing setup was fully local,

so the possibility of this configuration coming into action was low. While these types of Test Configurations above are expected to make a very small impact on the code coverage, they are still functionally relevant to the software. That is a key factor to choosing these configurations, which is also why configuration options like logging are not chosen for this experiment.

3.3 Fuzz testing setup

The fuzzing setup involves the use of AFL++ and QEMU. QEMU is not used directly in this setup, but fork of it is used internally by AFL++ to obtain signals that feedback into the fuzzer. When invoked in the 'binary only' mode, AFL++ is able to run the software in the forked version of QEMU and send seeds as inputs to the software.

The main reason for using fuzz testing is to ensure that for each configuration, the maximum amount of coverage has been gained. Fuzzing involves mutating seeds, running the software with the mutated seeds, analyzing signals to measure an increase in coverage, repeating the cycle with mutated seeds. Advanced fuzzers like AFL++ pick mutated seeds that add considerable coverage to the software and further mutate them to maximize coverage. This helps my experiments because it provides reliability to the results of the analysis.

Further, the experiments involve fuzzing each Test Configuration for multiple days, three to five days on average. Fuzz tests eventually reach a convergence, when no new seeds are being mutated and the coverage is relatively similar to the earlier mutation. In my experiments, this is the part when I shut down the fuzzer. At this point, the fuzzer has explored a good amount of the Test Configuration.

3.4 Chosen software

Considering these factors, I decided to go with these three software –

1. **Redis**. Redis is an in-memory cache, message broker and key-value data store. Redis has further applications too, like a Publisher-Subscriber service or a message queue. Redis is a pretty widely used software, employed by companies like Microsoft, FedEx and Gojek[8].
2. **Nginx**. Nginx is a web server, that has the capability of acting like a reverse-proxy, load balancer and an HTTP cache. Nginx hosts 34.4%[7] of the world's web pages.
3. **Apache httpd**. Apache httpd is a web server maintained by the Apache Foundation. It is the second most widely used web server after Nginx, hosting 32.2%[7] of the world's web pages.

Chapter 4

SOFTWARE CONFIGURATIONS

4.1 Redis

In my experiment, I fuzzed Redis version `6.2`. I tested Redis in the following 10 configurations, as described in Table-1, with descriptions in Table-2.

Name	Test Configuration	Default Configuration
rds01	appendonly yes appendfsync always	appendonly no appendfsync everysec
rds02	hash-max-ziplist-entries 32 hash-max-ziplist-value 16	hash-max-ziplist-entries 512 hash-max-ziplist-value 64
rds03	tracking-table-max-keys 10	tracking-table-max-keys 1000000
rds04	lazyfree-lazy-user-del yes	lazyfree-lazy-user-del no
rds05	maxmemory 100mb maxmemory-policy allkeys-lru maxmemory-samples 10	maxmemory (80% of total memory) maxmemory-policy noeviction maxmemory-samples 5
rds06	protected-mode yes	protected-mode no
rds07	bind *-::* protected-mode yes	bind 127.0.0.1 -::1 protected-mode yes
rds08	rdbcompression no rdbchecksum no	rdbcompression yes rdbchecksum yes
rds09	save 300 1	save 3600 1 save 300 100 save 60 10000
rds10	port 0 tls-port 6379	port 6379

Table 1. Redis Test Configurations

Name	Description
rds01	Runs Redis in append only mode
rds02	Change the thresholds for saving hashes
rds03	Reduce the number of tracked keys on the client
rds04	Reprogram `DEL` to work as `UNLINK`
rds05	Reduce max memory used by redis and the cache eviction policy
rds06	Enable protected mode without a `bind` directive
rds07	Enable protected mode with a `bind` directive
rds08	Disable RDB compression, checksums and sanitization of saved data
rds09	Change the frequency of saving data to disk
rds10	Enable TLS

Table 2. Redis Test Configuration descriptions

These configurations are described in detail in Redis' documentation[9]. There are a few configurations that are expected to give a very different code coverage, `rds10` with TLS enabled is one example of this. There are other configurations which may make a relatively smaller impact on the overall coverage. `rds09` is an example of this. In this configuration, we change the save frequency to a lower value. However, there may be cases where Redis may not need to save to disk, like in cases where the only operation was a retrieval.

To test these configurations, I created three base seeds. These three seeds performed the most common operations that are performed on Redis: `GET`, `SET`, and `DEL` (delete).

4.2 Nginx

For my tests, I used Nginx version `1.22.1`. There are 11 configurations that I built for this test. Table-3 describes the configuration options that I set on my Test Configurations.

Name	Test Configuration	Default Configuration
ngx01	aio on	aio off
ngx02	aio_write on	aio_write off
ngx03	keepalive_timeout 0s	keepalive_timeout 75s
ngx04	merge_slashes off	merge_slashes on
ngx05	autoindex on	autoindex off
ngx06	etag off	etag on
ngx07	expires -1	expires off
ngx08	gzip on gzip_comp_level 9	gzip off gzip_comp_level 1
ngx09	worker_processes 64	worker_processes 1
ngx10	worker_priority 20	worker_priority -10
ngx11	listen 80	listen 443 ssl

Table 3. Nginx Test Configurations

The configuration options in Nginx are elaborated upon in their documentation[10]. The Test Configurations I created for my tests is described in Table-4.

Name	Description
ngx01	Turns on Async I/O
ngx02	Turns on Async I/O writing
ngx03	Changes Keep-Alive timeout to zero seconds
ngx04	Merges trailing slashes in the request URL
ngx05	Lists the contents of the directory when the request URL ends in a '/'
ngx06	Adds an ETAG to the response
ngx07	Turns off Cache-Control by setting it to `no-cache`
ngx08	Enables Gzip compression and sets the compression level to maximum
ngx09	Reduces the number of worker processes to 1
ngx10	Changes worker priority to 20, like `nice` in Linux
ngx11	Enable HTTPS on port 443

Table 4. Nginx Test Configuration descriptions

In these configurations, there are a few configurations that are expected to have very different code coverage than the default. `ngx11`, where HTTPS is enabled, is one such configuration.

On the other hand, there are also a few configurations that should make a very small difference to the code coverage. An example of this could be `ngx03`, as the possibility of a Keep-Alive request coming to the software is low in a controlled testing environment like mine.

4.3 Apache httpd

The last software that I tested was Apache httpd, version `2.4`. The exact configuration options for the Test Configurations are described in Table-5.

Name	Test Configuration	Default Configuration
httpd01	HttpProtocolOptions Unsafe LenientMethods Require1.0	HttpProtocolOptions Strict LenientMethods Allow0.9
httpd02	AllowEncodedSlashes ON	AllowEncodedSlashes OFF
httpd03	KeepAliveTimeout 1ms	KeepAliveTimeout 5
httpd04	MergeSlashes OFF	MergeSlashes ON
httpd05	Options Indexes DirectoryIndex None.html	Options FollowSymlinks DirectoryIndex index.html
httpd06	FileETag All	FileETag MTime Size
httpd07	ExpiresActive ON ExpiresDefault access plus 0 seconds	ExpiresActive Off
httpd08	DeflateCompressionLevel 9	// not set
httpd09	MaxRequestThreads 1	MaxRequestThreads 16
httpd10	StrictHostCheck Off	StrictHostCheck On
httpd11	SSLEngine On	SSLEngine Off

Table 5. Apache httpd Test Configurations

Apache httpd's documentation[11] mentions these configurations in detail. One thing to note is that the exact description of specific options exist in their respective modules only. `DeflateCompressionLevel`, for example, is present in the `mod_deflate` module documentation. A summary of these configurations are described in Table-6.

Name	Description
httpd01	Allow requests with unknown HTTP Request methods
httpd02	Allow slashes to be encoded in HTTP Encoding (`%2F`)
httpd03	Reduce the Keep-Alive timeout to 1 millisecond
httpd04	Allow trailing slashes at the end of the URL
httpd05	List the directory index if the request URL ends with a '/'
httpd06	Set FileEtags to have all attributes
httpd07	Set Cache-Control to `no-cache`
httpd08	Enable Gzip and set compression level to maximum
httpd09	Reduce the maximum request threads to 1
httpd10	Enable strict host checking, disallow requests with unknown hosts
httpd11	Enable HTTPS

Table 6. Nginx Test Configuration descriptions

As it is evident from the summary above, many configurations overlap with those in Nginx. This is intentional to allow a comparison between the two software that provide similar configurations and features.

As was the theme between the last two software, this software too has configurations that should make a huge difference to the code coverage, like `httpd11` with HTTPS. There is a possibility that `httpd10` with strict host checking makes little difference to the code coverage as many requests to it may fail at the host checks.

Chapter 5

METHODOLOGY – COVERAGE INSTRUMENTATION

In order to ensure that the software in each Test Configuration is covered to a reliable extent, I fuzzed each of them. During the fuzz tests, AFL++ keeps track of the seeds that cause an increase in coverage. In the subsequent sections, I shall refer to them as 'generated seeds'. These generated seeds form the basis of my analysis in both of the following methodologies.

5.1 Premise

At a high level, instrumentation is adding additional code to the binary executable during compilation, that is responsible for providing metadata about the program's execution. For example, instrumented code is capable of reporting the number of times a certain function is invoked, or how much time a certain operation took.

In my case, I used instrumentation to obtain code coverage information for the execution of the software under test. In order to do this, I used the 'clang' C compiler that is part of the LLVM project. Since all my target software were written either in C or C++, `clang` was a great fit for compilation. `clang` and its backend LLVM[12] allow us to generate various types of instrumented code, and in my case I used Profile Instrumentation and Coverage Mapping. The former, activated by the flag `-fprofile-instr-generate`, emits information in terms of how many times a certain piece of code is executed. The latter, activated by the flag `-fcoverage-mapping`, maps this information to the exact file and line of code that is executed.

Upon running any of these instrumented programs, a binary file ending in '`.profraw`' will be generated by the program. I process these files further and convert them to a plain-text format called 'lcov', to make it easier to parse and store. `lcov`[13] is a popular format for storing coverage information, used widely for its simple structure for storing coverage information while also being human-readable. These files form the basis of my analysis in this method.

5.2 Analysis Model

The trace file generated as a result, in the 'lcov' format, contains detailed coverage information of the software's execution. A count of lines found, lines hit, branches found, branches hit, functions found, functions hit is part of this information. The trace file also contains the execution count of each function, line and branch. This information is grouped by every source file in the software.

In my case, I focused on a summarized version of the trace. This summary included the following parameters for every source file –

1. Number of lines found.
2. Number of lines executed.
3. The exact lines that are executed.
4. Number of functions found.
5. Number of functions executed.

These factors from the trace file are sufficient to quickly judge the impact on code coverage a certain Test Configuration has on the software under test. The reason being that I can obtain raw statistics on the broad components of the software, like functions and lines. As the trace data is also grouped by source files, I can also

measure how much a Test Configuration affects the code coverage of a specific file or a module. The final advantage that this approach gives me is having a quantifiable value on how much code coverage increased or decreased holistically because of the Test Configuration. This value can also be obtained for a certain file or set of files if I narrow my scope of measurement to that set of files.

The software that is run in default configuration produces the reference trace and the Test Configuration produces the test trace. Then, I contrast the test trace from the reference trace for two data points: lines executed and functions executed.

Lines executed gives an idea of the total code coverage in terms of raw lines of code. The information I gain from here is the amount of total coverage that the Test Configuration affects. The count of functions executed, on the other hand, gives a measure of how much the Test Configuration affects the code coverage functionally. There could be a scenario where the lines of code executed can be within a specific set of functions only. Therefore the lines of code alone does not act as a strong signal for how much the coverage is affected by the test configuration. If we combine the count of lines executed with the number of functions executed, we can get a more holistic idea of how much functionally the Test Configuration affected the overall coverage in that specific execution.

As the fuzz test generates a number of seeds during its run, I repeat the above approach for each of the generated seeds. After gaining trace information for all the seeds, I aggregate information for each Test Configuration trace contrasted with the corresponding reference configuration trace. The aggregated information includes the maximum, minimum and average difference from the reference trace measurements for lines executed and functions executed. I also show the difference in traces for specific

cases, if these are edge cases where the difference in coverage is expected to be very similar or very dissimilar to the reference trace.

Algorithm 1 Obtaining coverage gain using coverage instrumentation traces

$LFs \leftarrow \emptyset$ ▷ Initialize empty list of lines executed
$FNFs \leftarrow \emptyset$ ▷ Initialize empty list of functions executed
for *seed* in *generated_seeds* **do**
    $base\_trace \leftarrow$ **cov_trace**$(base\_config, seed)$
    $test\_trace \leftarrow$ **cov_trace**$(test\_config, seed)$
    $LF \leftarrow test\_trace.lf - base\_trace.lf$ ▷ Difference in count of lines executed
    $FNF \leftarrow test\_trace.fnf - base\_trace.fnf$ ▷ Difference in count of functions executed
    $LFs \leftarrow LFs \cup LF$ ▷ Add to list of lines executed
    $FNFs \leftarrow FNFs \cup FNF$ ▷ Add to list of functions executed
end for

cov_trace, in this pseudocode, is a procedure that obtains the coverage trace information from the software run in a specific configuration.

This gives an approximate but close measure of how much code coverage difference is caused by the Test Configuration. After I have identified the configurations that have caused a stark difference in coverage, I will perform deeper analyses on those configurations. These deeper analysis include analyzing the exact source code files and lines that have differed, then reading the source code to understand what the differing code is performing.

Finally, to eliminate any level of randomness due to fuzzing, I perform a cross-seed analysis on every configuration. In this analysis, I will a configuration with the generated seeds of all the other configurations. After obtaining traces for every configuration, I will compare the code coverage of each crossing seed to see how much difference has the seed caused in the code coverage for that configuration.

5.3 Overlap Analysis

After a holistic idea of how certain configurations have differed from the Base Configuration, I performed overlap analysis. Here, I am looking for configurations that have code that overlaps. The reason I needed to evaluate this was to understand the configurations that, while making impact on the coverage, are making the same impact on the configuration. So for many purposes, except functional, selecting the same configuration makes very little difference.

The results of this analysis can help systems that need to know coverage of the software, irrespective of what the software is doing. Fuzzers, for example, could deeply benefit from these results. Knowing that two or more configurations nearly have the same effect on the coverage of the software, then just fuzzing one of them and eliminating the others helps decrease the search space of the fuzzer.

5.4 Causality Analysis

After understanding the configurations that have caused the most difference in coverage than the Base Configuration, and also understanding the configurations that have the most and least overlap with each other, I will perform Causality Analysis on Test Configurations that really stick out.

I perform this to expose the exact code that is covered just by that configuration and is not caused by the randomness of the fuzz test. The way I do that is finding the lines of code that are always triggered in every run of that configuration, that are not part of any other configuration.

Finally, I read the lines of code that are identified through this process, and

then conclude what caused the divergence in the first place. This analysis helps us understand the root cause of the divergence and gives us a way to isolate the lines of code that are covered only because of the configuration change.

Chapter 6

ANALYSIS RESULTS – COVERAGE INSTRUMENTATION

6.1 Redis

Figure-1 and Figure-2 describe how average of Lines Hit and Functions hit, respectively, differed from the base configuration.

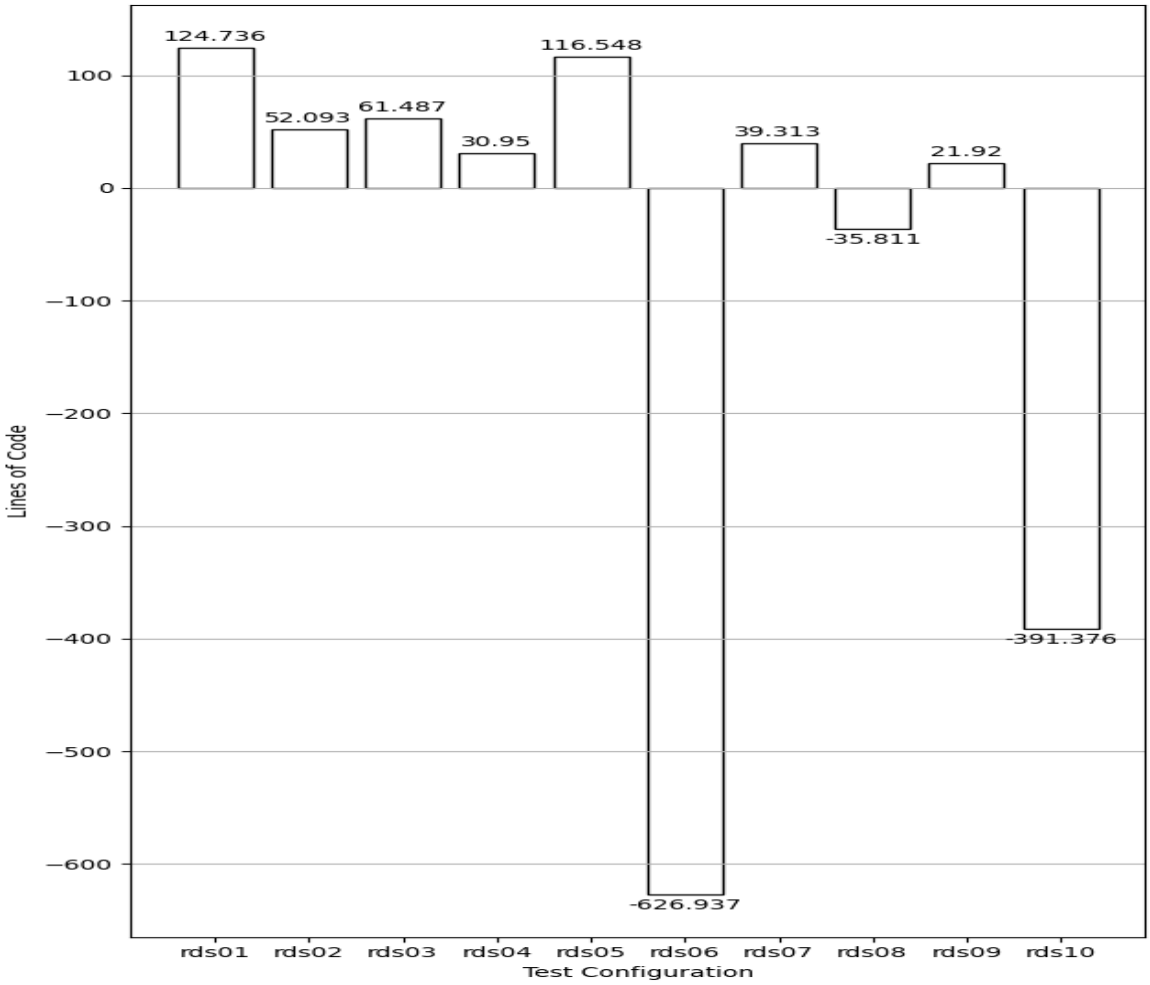


Figure 1. Average Lines Hit Difference for Redis

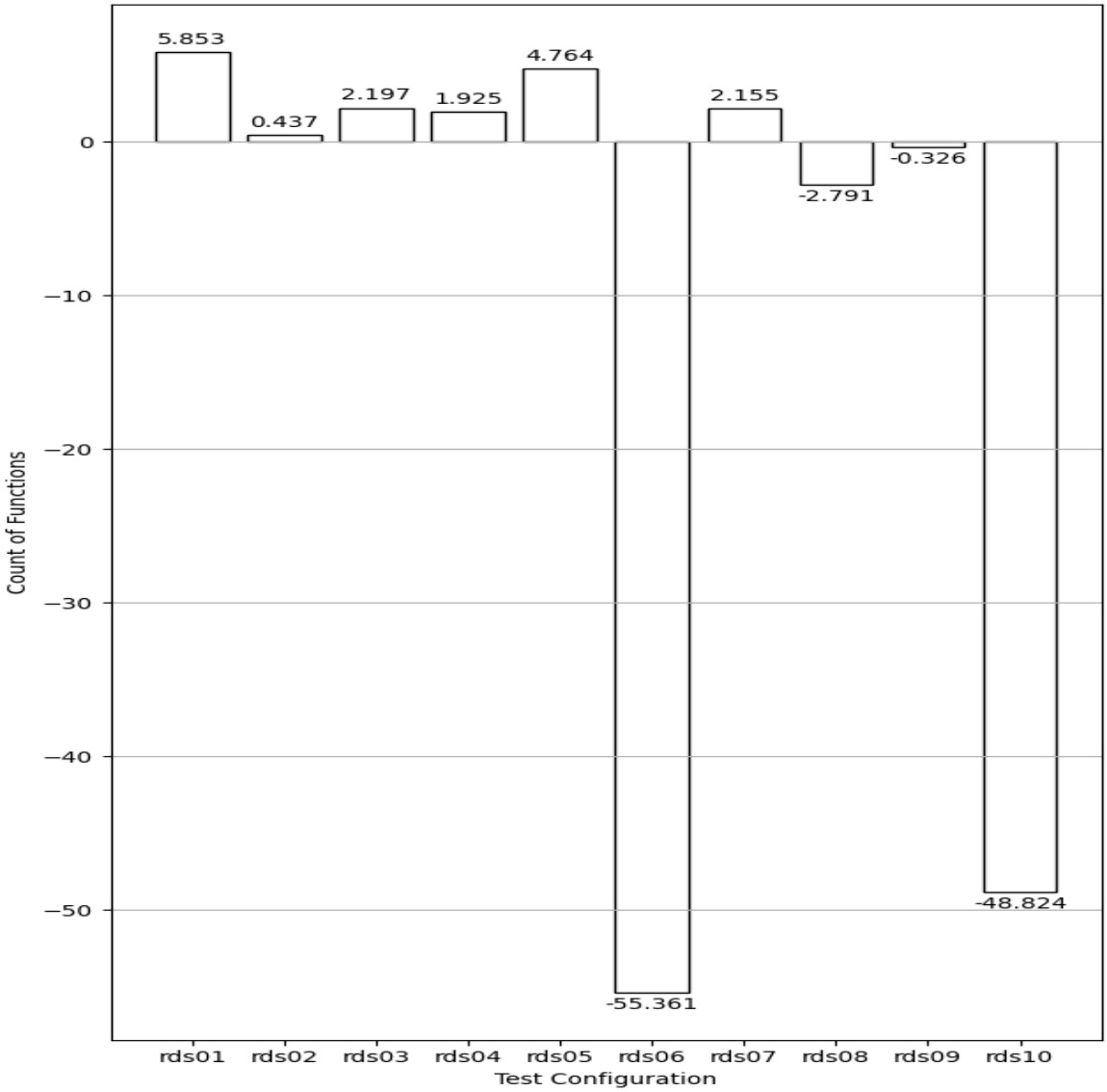


Figure 2. Average Functions Hit Difference for Redis

Table-7 shows the average Lines Hit difference and Functions Hit difference for all the Test Configurations of Redis. It is worth noting that the line coverage and the function hit coverage for each configuration vary in relation to each other. `rds10` has more lines and functions, while `rds02` has fewer lines and functions differing.

`rds10` again seems to have a huge difference in Lines and Functions Hit, as was

Test Configuration	Avg Functions Hit Difference	Avg Lines Hit Difference
rds01	5.853	124.736
rds02	0.437	52.093
rds03	2.197	61.487
rds04	1.925	30.95
rds05	4.764	116.548
rds06	-55.361	-626.937
rds07	2.155	39.313
rds08	-2.791	-35.811
rds09	-0.326	21.92
rds10	-48.824	-391.376

Table 7. Average Lines Hit and Functions Hit Difference for Redis

expected given the configuration it runs in. `rds06`, on the other hand, had much lower code coverage than the base configuration. This could be because most requests sent to `rds06` were dropped before they were even parsed as Redis was running in protected mode. If we contrast this with `rds07`, where Redis runs in protected mode but while accepting on all interfaces and all addresses, I observed that the average difference in number of Lines and Functions was much smaller, 30.95 lines and 1.925 functions respectively. This is because the requests must have reached Redis, and were subsequently executed.

Also worth noting is the `rds01` configuration. The difference in Lines Hit is significant at 124.736, but Functions Hit is around 5.8. The statistic indicates that

the code coverage within Redis for this configuration was high. This is also the case for `rds05`. For `rds05`, the coverage could be because of `jemalloc`. `jemalloc` is a general purpose implementation of `malloc` from `libc` that Redis is statically complied with. As `rds05` changes how much memory Redis should use, the coverage must have changed in `jemalloc`'s code as it is statically compiled within Redis.

Finally, there's configurations like `rds04`, `rds09` and the remaining configurations that averaged less than 65 differing Lines Hit. This could be because the functionality required to implement these configurations must not be too large in code. `rds09`, for example, could just be a small piece of code that checks if it is time to save the in-memory database down to the disk.

Overall, the results of the analysis line up with how much difference the configurations could have made. `rds06` is the only configuration that happened to have made a larger difference than imagined.

6.2 Nginx

Figure-3 and Figure-4 describe how average of Lines Hit and Functions hit, respectively, differed from the base configuration.

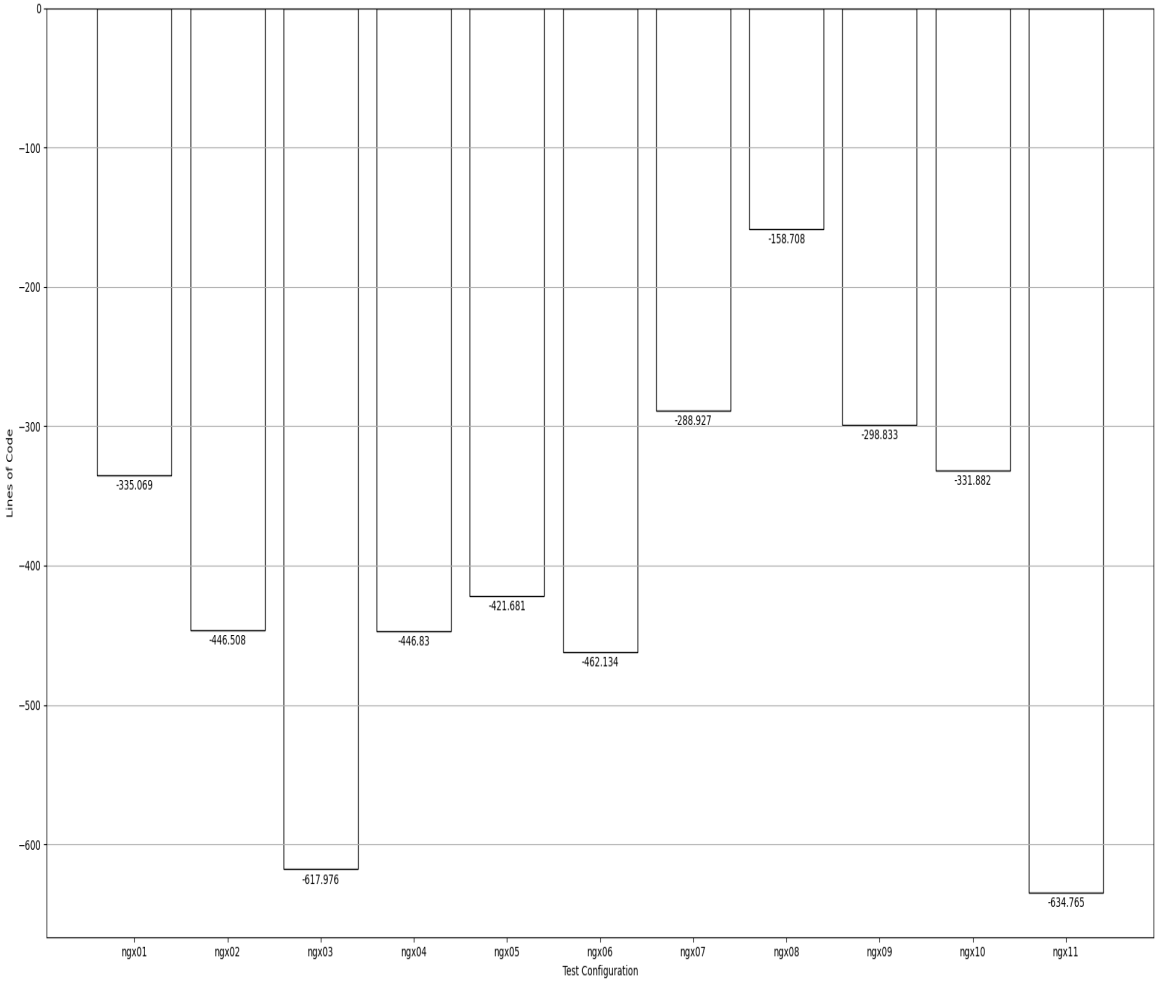


Figure 3. Average Lines Hit Difference for Nginx

Looking at Table-8, Figure-3 and Figure-4, the clearest thing to observe is that every Test Configuration covered less overall code than the base configuration. This implies

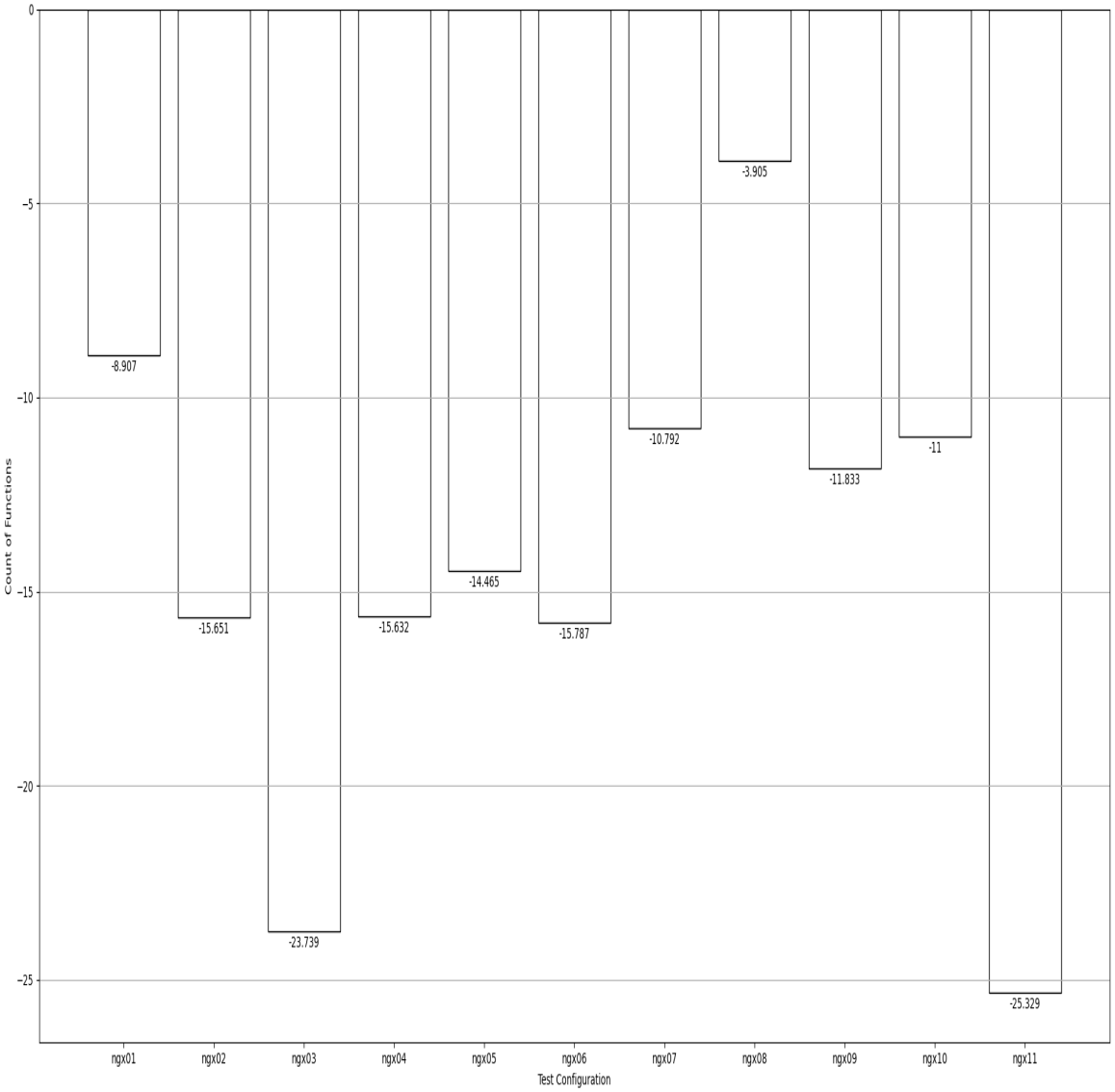


Figure 4. Average Functions Hit Difference for Nginx

that every Test Configuration caused considerable parts of the Base Configuration to not be covered in their execution.

The `ngx11` and `ngx03` configurations were the most divergent in terms of Lines Hit and Functions Hit, covering 617.976 and 634.765 fewer lines than the base configurations. `ngx11` was expected to have such a coverage, as it enables TLS, which causes the coverage of code that parses TLs requests. However, `ngx03` was unexpected.

Test Configuration	Avg Functions Hit Difference	Avg Lines Hit Difference
ngx01	-8.907	-335.069
ngx02	-15.651	-446.508
ngx03	-23.739	-617.976
ngx04	-15.632	-446.83
ngx05	-14.465	-421.681
ngx06	-15.787	-462.134
ngx07	-10.792	-288.927
ngx08	-3.905	-158.708
ngx09	-11.833	-298.833
ngx10	-11.0	-331.882
ngx11	-25.329	-634.765

Table 8. Average Lines Hit and Functions Hit Difference for Redis

`ngx03` reduces the keep-alive timeout to zero seconds. It indicates that changing the keep-alive timeout affects a larger part of the source code than expected.

Another interesting find is that `ngx08` happens to have the lowest coverage difference than the base, while it was expected to have a considerable difference. `ngx08` turns on Gzip compression for incoming requests and responses, serving compressed resources to the client. It only differs in 158.708 lines (fewer) than the Base Configuration. Upon further investigation, it seems that Nginx calls `zlib` to compress resources. This explains why Nginx in this configuration has lower coverage difference, as it just ships off the main functionality of resource compression to the `zlib` library.

Finally, the other configurations have a lower difference in relation to `ngx11` and `ngx03`, yet the difference in the absolute sense is pretty high. The only configurations that had a rather modest difference in coverage were –

1. **ngx07**. Adds an `Expires` header to the response.
2. **ngx09**. Changes number of worker processes from 64 to 1.
3. **ngx10**. Changes worker priority from -10 to 20.

On average, the configurations have 403.94 fewer lines than the base Nginx configuration.

6.3 Apache httpd

Figure-5 and Figure-6 describe how average of Lines Hit and Functions hit, respectively, differed from the base configuration.

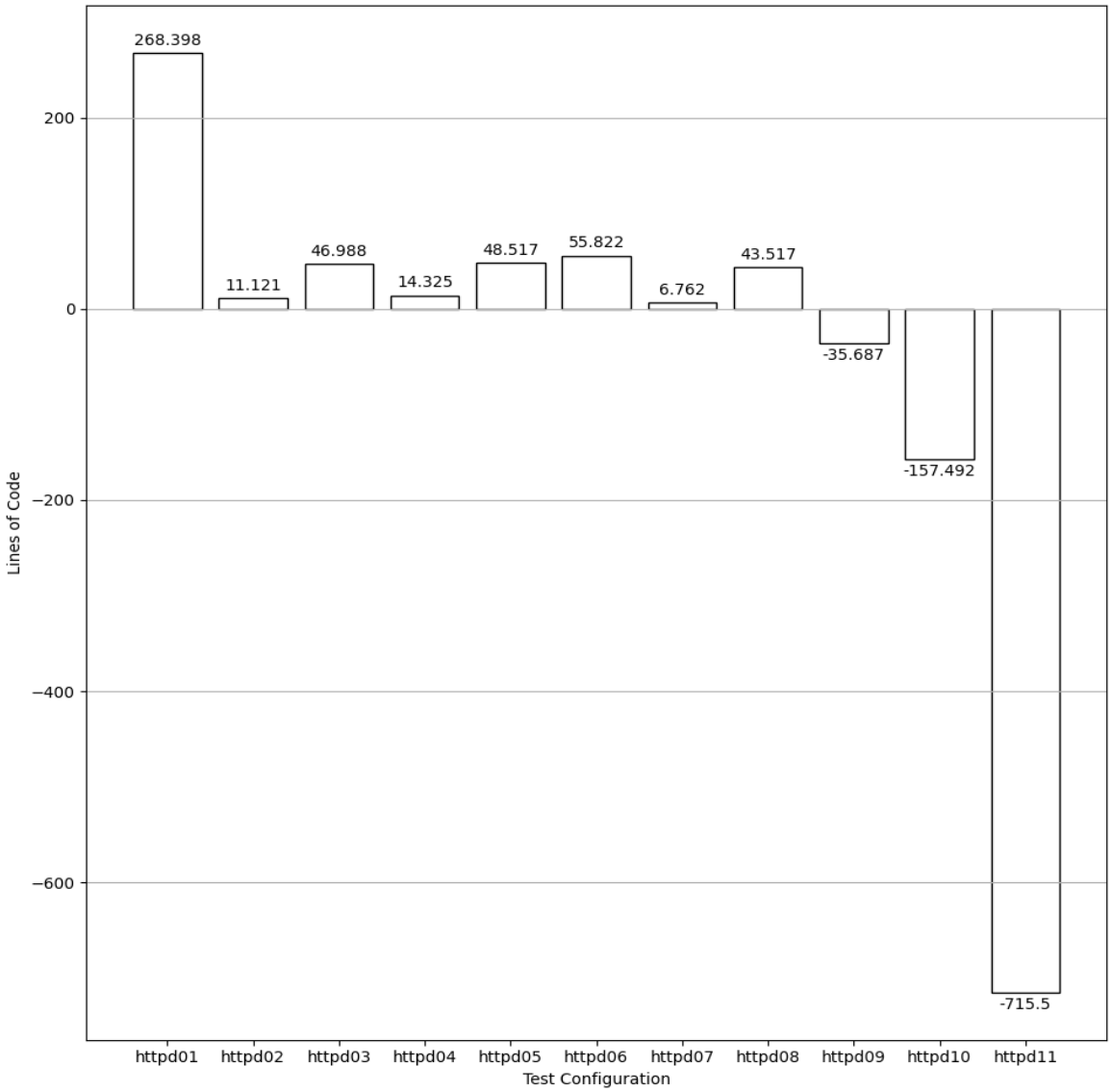


Figure 5. Average Lines Hit Difference for Apache Httpd

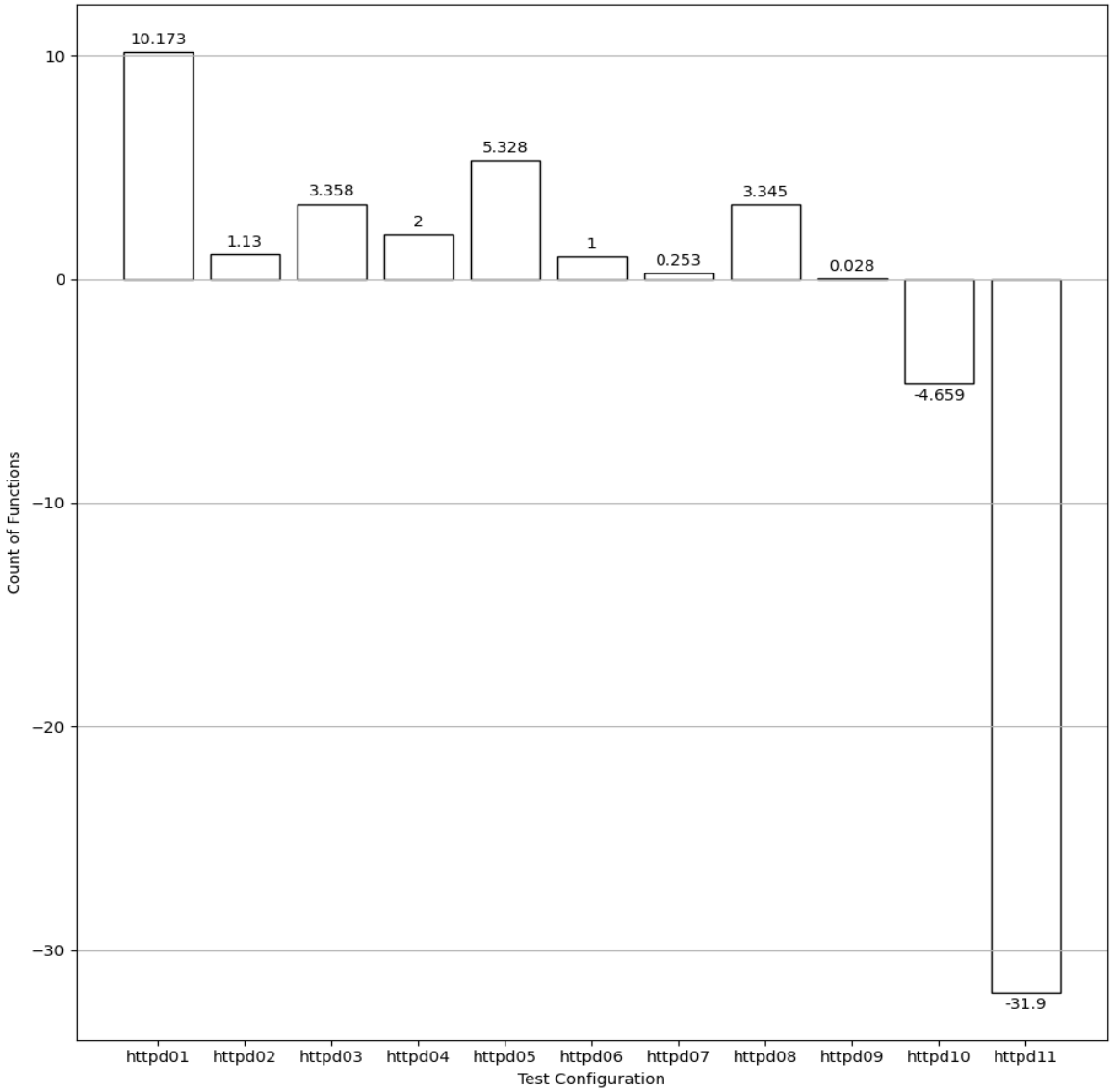


Figure 6. Average Functions Hit Difference for Apache Httpd

Table-9 shows the average Lines Hit and Functions Hit for httpd. It appears to be a mix of more and fewer code coverage than the Base Configuration.

As was the pattern with the other two software, the configuration with TLS enabled, `httpd11`, happens to have a very huge difference in code coverage than the Base Configuration. `httpd11` had 715.5 fewer Lines Hit and 31.9 fewer Functions Hit

Test Configuration	Avg Functions Hit Difference	Avg Lines Hit Difference
httpd01	10.173	268.398
httpd02	1.13	11.121
httpd03	3.358	46.988
httpd04	2.0	14.325
httpd05	5.328	48.517
httpd06	1.0	55.822
httpd07	0.253	6.762
httpd08	3.345	43.517
httpd09	0.028	-35.687
httpd10	-4.659	-157.492
httpd11	-31.9	-715.5

Table 9. Average Lines Hit and Functions Hit Difference for Apache Httpd

than the Base Configuration. Understandably so, as the reasons for this would be same as the reasons for the other two software run with TLS.

The other configurations that clocked in a pretty high difference were `httpd01` and `httpd10`. The former, disallowing lenient parsing of HTTP Methods, had 268.398 more coverage than the Base Configuration. The additional code covered should be the code that is required to strictly parse HTTP Methods. The latter, enabling strict host-checking has 157.492 fewer Lines Hit. This makes sense as many requests must not have been processed by Nginx in this configuration, leading to a lower coverage than the base.

Among these, the configurations that had significant coverage, but lower than the above mentioned configurations, are `httpd03`, `httpd05`, `httpd06`, `httpd08` and `httpd09`. `httpd05` is an interesting configuration in this list, as it serves the directory listing of the web root folder as opposed to a webpage. While it has a high Lines Hit difference, it has a fairly high Functions Hit difference too, higher than the others in this category. This must be because of the rendering of the listing web page or just listing the contents of the directory along with `stat` information for all files.

`httpd06`, a configuration that enables the resources ETag to have all information (inode number, modified time and size), has the highest Lines Hit difference while only having a Functions Hit difference of 1.0. This could be because while the ETag is calculated by a single function. `httpd08` also has a high number of Lines Hit difference, as is expected from the configuration that enables Gzip.

The configuration with the lowest coverage was `httpd07`, which added an `Expires` header on the served resource. This is extremely interesting as `httpd07` had one of the most differing code coverage in the Translation Blocks analysis.

Chapter 7

OVERLAP ANALYSIS

In this analysis, I checked how many configurations of a software happened to have similar code coverage. To do this, I obtained the 'lcov' coverage data for each software and converted the line coverage into a set, call this a 'trace set'. Now I find the intersection of every configuration's trace set with every other configuration's trace set. The configurations that intersect a lot are very similar in code coverage, while those that are not too similar have covered different parts of the source code.

7.1 Redis

Figure-7 shows how much overlap existed between configurations. Most configurations seemed to have less than 10 percent divergence. In fact, some configurations like `rds04` and `rds07` were 97% similar in terms of code coverage. Other configurations that had very similar coverages are `rds01` and `rds04` at 94%, and `rds02` and `rds05` with 96% similar code coverage.

However, there were two configurations that happened to consistently have a large difference in coverage than the other configurations. These are `rds06` and `rds10`, respectively. The former is running Redis in protected mode and the latter is running Redis with TLS enabled. These findings actually line up with the Coverage Instrumentation findings in Chapter 6, where these configurations had a large divergence from the base Redis configuration's code coverage.

`rds10` had only 60% of its code overlap with `rds01`, and scores similar figures

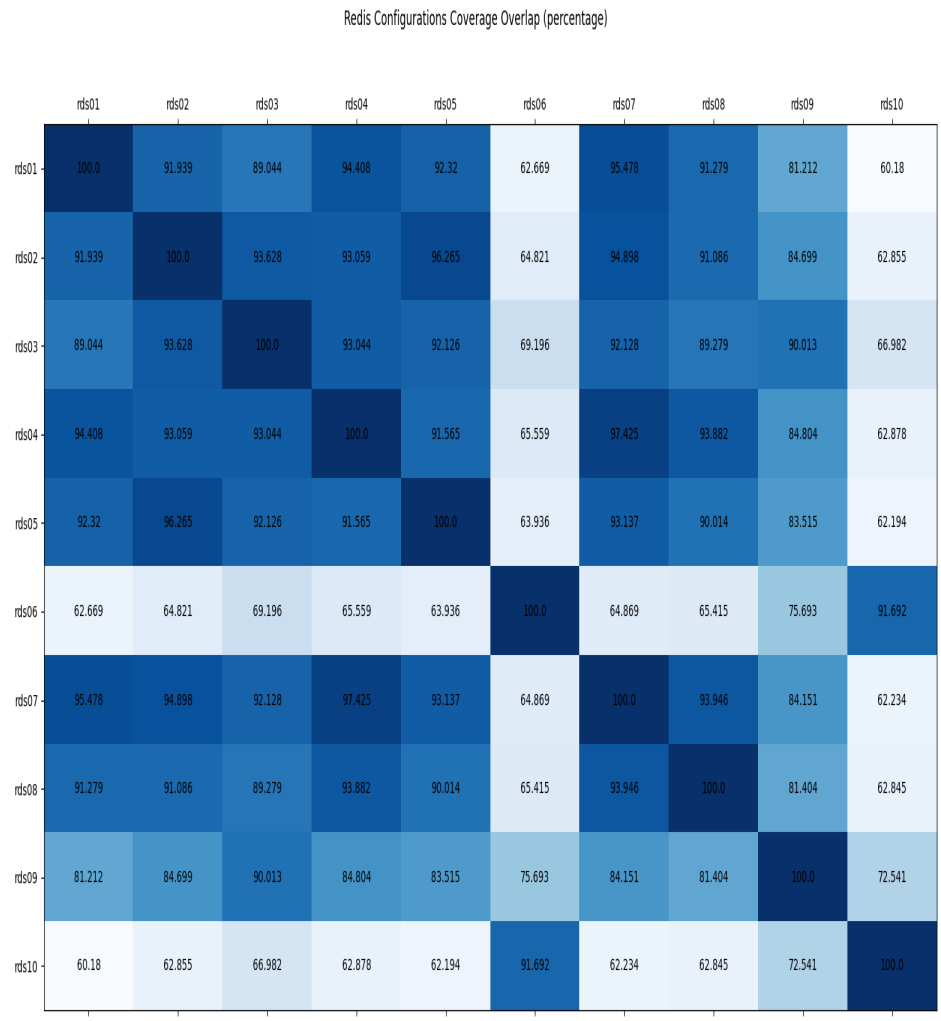


Figure 7. Code Coverage Overlap for Redis Configurations (in Percentage)

with the other Redis configurations. However, with `rds06`, it has 91% overlap. As for `rds06`, it has the lowest overlap with `rds01` at 62%, and scores similar overlap percentages with other configurations. The cause of this is analyzed in the subsequent chapter. An interesting thing to note is that there were as much as 5060 lines of code that were part of every configuration in this experiment. If we consider these

to be configuration-independent, then my analysis showed that 92.3% and 96.3% of configuration-independent code was present in `rds10` and `rds06`.

7.2 Nginx

Figure-8 plots the coverage overlap of Nginx configurations.

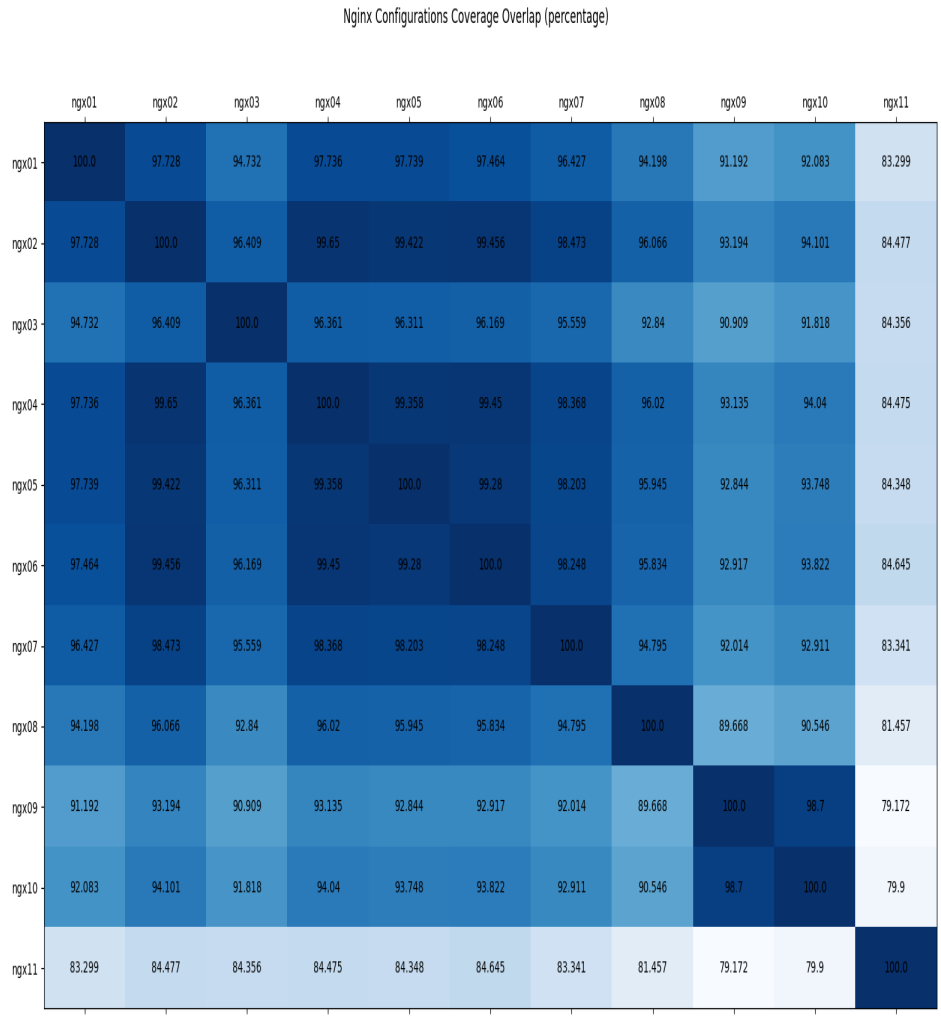


Figure 8. Code Coverage Overlap for Nginx Configurations (in Percentage)

Many configurations of Nginx had a large overlap. Some configurations overlapped

up to 99%. ngx04 alone has approximately 99% similarity with ngx02, ngx05, ngx06, ngx07.

Now let's consider ngx11. ngx11 enables TLS on Nginx. It has consistently low overlap with every configuration on the graph. This finding, again, lines up with the results in Chapter 6, where ngx11 had a very different coverage than the base configuration. ngx11 overlaps the least with ngx09, at 79.172%. The most it overlaps with is ngx02 at 84.477%. Even then, it contained more than 15% unique, non-overlapping lines. The cause of this is analyzed in the subsequent chapter.

Nginx also happened to have 11530 lines of code intersecting in every configuration. In this seemingly configuration independent set of lines, ngx11 happened to have 87.1% overlap with this code.

7.3 Apache httpd

Figure-8 plots the coverage overlap of Apache httpd configurations.

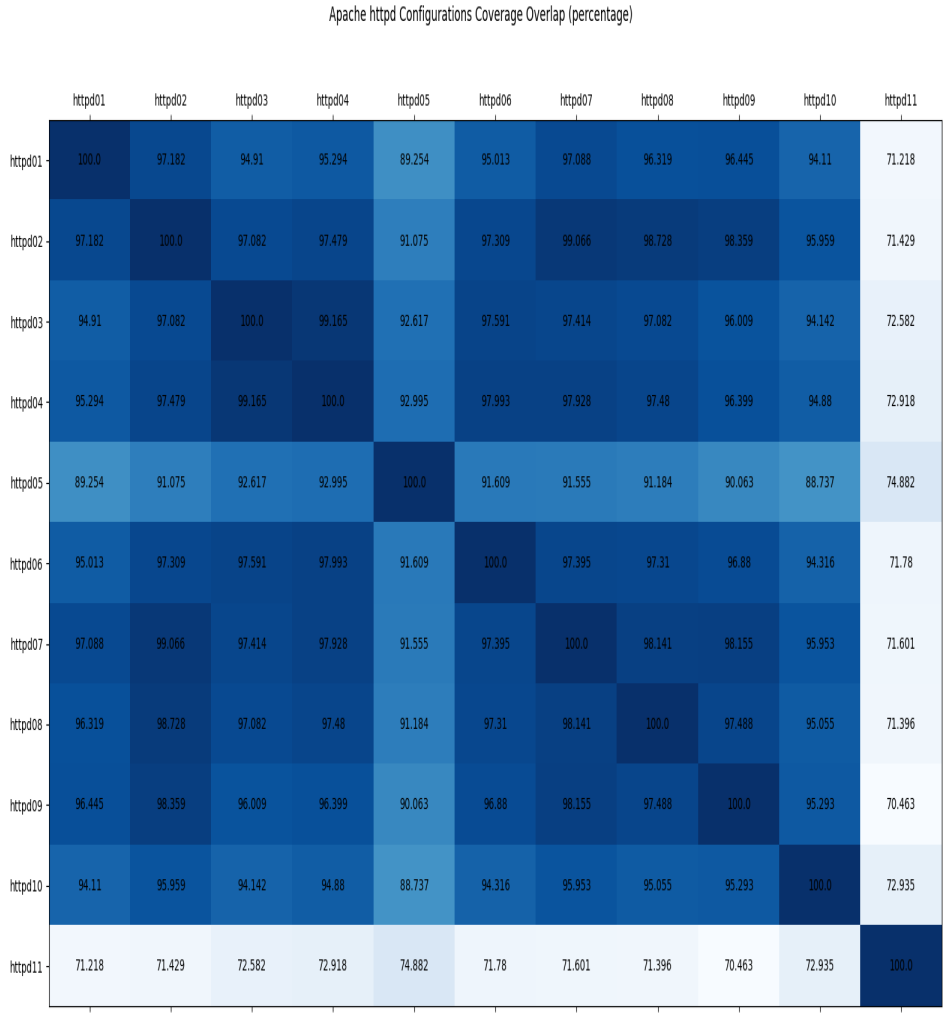


Figure 9. Code Coverage Overlap for Apache httpd Configurations (in Percentage)

Like Nginx, httpd too had many overlapping configurations. Most configurations

had more than or close to 95% overlap. Configurations like `httpd02` has over 98% overlap with `httpd07`, `httpd08` and `httpd09`.

`httpd11` is httpd running with TLS enabled, and it had the lowest overlap with other configurations in the list. The highest overlap it had was approximately 75%, which is still 25% non-overlapping code. The lowest overlap it had was with `httpd09` at 70%. It isn't unreasonable to assume that the causes for this difference in overlap in httpd is quite similar to that of Nginx. In the next section, I perform deeper analysis of this configuration, by understanding the exact code that was covered by this configuration and none of the other configurations.

In this analysis, I found that 6252 lines of code were common across configurations, thus were seemingly configuration independent. `httpd11` had 98% overlap with this configuration independent code, indicating that it had really low coverage induced by the configuration change.

Chapter 8

CAUSALITY ANALYSIS

In order to analyze causality of every configuration that had very little overlap in coverage with other configurations, I had to look for the lines of code that were uniquely and consistently covered in these configurations, and nowhere else. The method of finding these unique lines of code can be boiled down to a set expression.

Let's consider a configuration C, and we wish to find the lines of code that are uniquely and consistently part of C. Let the coverage of C be a set of lines called S_c. I first find a set difference between S_c and every other trace set. This gives me the lines of code that are present in S_c and not in the other configuration. After I have performed this for every configuration, I obtain the intersection of all the set differences that I have obtained. Now, I have all the lines of code of C that are in set S_c and present in every difference.

$$CAU_c = \bigcap \left(set\_diff_{i=0}^{n} \Big(S_c, S_i \Big) \right)$$

Here CAU_c is the set of lines of code that is uniquely present in Configuration C and is present in every set difference against C. I repeated this process for every configuration to find the lines of code that are caused by that configuration. This analysis helps understand why these configurations have different code coverage, and what is the code that differs among them.

I perform this analysis on configurations that have either showed a considerably low overlap than other configurations, or have showed a pretty high coverage difference

than the Base Configuration. The findings that did have something useful to show out of this analysis are mentioned in the 'Other Findings' subsections for each software.

8.1 Redis

8.1.1 Redis in Append Only Mode

In the Append Only mode, Redis saves an AOF (Append Only File) that saves every operation that Redis performs. One file that uniquely stood out was `src/aof.c`. This file contains all the code required by Redis to perform operations on AOF. The functions `aofWrite` – writes AOF file to disk, `flushAppendOnlyFile` – flushes AOF to disk and `loadAppendOnlyFile` – loads AOF on startup, were among the functions called during the execution of this Test Configuration.

Apart from that, the `src/server.c` file also had coverage unique to this configuration. The snippet in Figure-10 shows how Redis checks if Append Only mode is enabled and then calls `flushAppendOnlyFile`. This is called in the `beforeSleep` function in `src/server.c`, which is called every time Redis is entering its event loop. There are other lines all over the file that point to loading AOF, writing to AOF, etc.

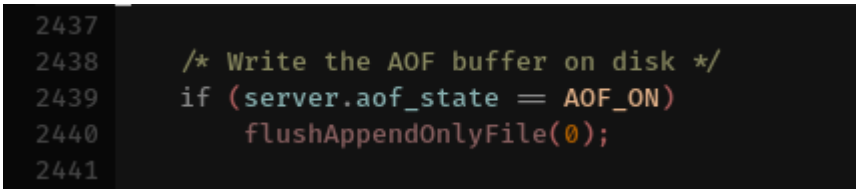


Figure 10. Code Snippet From Redis Executed in Append Only Configuration

8.1.2 Redis with faster saves

The file `src/config.c` had code that wasn't covered in any configurations. When I examined the code, the differing lines had code that would translate strings to integers (using `atoi`) and records that as the `save` configuration for the execution.

As this configuration only changes the frequency at which data is saved, these were the only unique lines of code that I could find.

8.1.3 Redis in Protected Mode

Majorly, Redis in protected mode received coverage from two files: `src/networking.c` and `src/anet.c`. The former manages incoming connections to the Redis server and the latter is an abstraction over Socket API that the OS exposes. In `src/anet.c`, Redis calls the `anetFdToString` function to convert the IP address and port to text form, by internally calling Linux's `inet_ntop` function. This could be for initializing Protected Mode's allowed IP addresses.

In `src/networking.c`, the only unique code covered is the warning that Redis provides when a host from an unknown IP or interface connects. Figure-11 shows this code.

There's some coverage in `src/connection.c`, however it is only a wrapper function around `anetFdToString`. I also calculated the negative overlap of this configuration to see how little the coverage was compared to other configurations. There were 1370 lines of code that were not present in this configuration but were present in all other configurations. I found this by calculating the set difference of the lines intersecting in all configurations, differed with the lines that were found in this configuration.

```
1005    if (server.protected_mode &&
1006        server.bindaddr_count == 0 &&
1007        DefaultUser->flags & USER_FLAG_NOPASS &&
1008        !(c->flags & CLIENT_UNIX_SOCKET))
1009    {
1010        char cip[NET_IP_STR_LEN+1] = { 0 };
1011        connPeerToString(conn, cip, sizeof(cip)-1, NULL);
1012
1013        if (strcmp(cip,"127.0.0.1") && strcmp(cip,"::1")) {
1014            char *err =
1015                "-DENIED Redis is running in protected mode because protected "
1016                "mode is enabled, no bind address was specified, no "
1017                "authentication password is requested to clients. In this mode "
1018                "connections are only accepted from the loopback interface. "
1019                "If you want to connect from external computers to Redis you "
1020                "may adopt one of the following solutions: "
1021                "1) Just disable protected mode sending the command "
1022                "'CONFIG SET protected-mode no' from the loopback interface "
1023                "by connecting to Redis from the same host the server is "
1024                "running, however MAKE SURE Redis is not publicly accessible "
1025                "from internet if you do so. Use CONFIG REWRITE to make this "
1026                "change permanent. "
1027                "2) Alternatively you can just disable the protected mode by "
1028                "editing the Redis configuration file, and setting the protected "
1029                "mode option to 'no', and then restarting the server. "
1030                "3) If you started the server manually just for testing, restart "
1031                "it with the '--protected-mode no' option. "
1032                "4) Setup a bind address or an authentication password. "
1033                "NOTE: You only need to do one of the above things in order for "
1034                "the server to start accepting connections from the outside.\r\n";
1035            if (connWrite(c->conn,err,strlen(err)) == -1) {
1036                /* Nothing to do, Just to avoid the warning... */
1037            }
1038            server.stat_rejected_conn++;
1039            freeClientAsync(c);
1040            return;
1041        }
1042    }
1043
```

Figure 11. Code Snippet From Redis Executed in Protected Mode

8.1.4 Redis Running with TLS Enabled

This is the only configuration that showed a lot of coverage in the `src/tls.c`. Needless to say, this source file contains multiple methods that pertain to parsing and decrypting encrypted TLS traffic. 206 lines from `src/tls.c` have at least been covered in every run of Redis with TLS enabled.

As for other files, `src/networking.c` also has some unique coverage. Its function

`acceptTLSHandler` is uniquely covered only in this configuration. This function internally calls `src/tls.c` file's `connCreateAcceptedTLS` function, which further handles the incoming TLS connection. Interestingly, there were only 77 lines of code that were present in every other configuration but this, which is a lower number than the rest of the configurations of this type in the other software.

8.1.5 Other Findings

As was the case with the previous analysis, the configuration `rds03` had very little impact on code coverage. In this analysis, it did not have any lines of code that were unique to it. `rds04` lazily deletes keys and had triggered the `dbAsyncDelete` function from `src/lazyfree.c` uniquely and for every execution of the Configuration. Interestingly, the only code that `rds07` (protected mode with allowing all connections) had unique was configuration code that set the bind addresses. This makes sense, as the rest of the configuration was untouched and worked as a regular Redis server. Overall, the configurations pointed to exactly the code that was pertinent to them, and the configurations that did not make much impact did not have a lot of unique code.

8.2 Nginx

8.2.1 Keep-alive Timeout Set to Zero

When Nginx is run with keep-alive timeout set to zero, two files show unique coverage – `src/core/ngx_parse.c` and `src/http/ngx_core_http_module.c`. In the

first file, a lot of the unique lines are around parsing the configuration file. As the timeout has to be specified as a time, which can be seconds or minutes, the value set in the configuration file is parsed. After parsing, this value is sanitized and calculated. This happens in a function called `ngx_parse_time`.

In `src/http/ngx_core_http_module.c`, there are two regions of code that are covered in every run of this configuration. The first is on line 1340, where the keep alive timeout is set to zero. This happens after the code checks if the timeout is set to zero in the configuration file. Further, it calls the function `ngx_http_core_keepalive` which sets the keep-alive value in the configuration file into an internal configuration object.

8.2.2 Nginx Running with TLS Enabled

Upon just raw analysis, it appears that running Nginx in TLS causes 559 unique lines of code to be covered. Most of these lines are from two files – `src/http/modules/ngx_http_ssl_module.c` and `src/event/ngx_event_openssl.c`. In `src/event/ngx_event_openssl.c`, the function `ngx_create_ssl` is covered in this configuration, which is a function that initializes the SSL context. Other functions initialize data that is required by SSL, like Diffie-Hellman parameters, SSL session caches, etc.

The `src/modules/ngx_http_ssl_module.c` file contains code to initialize SSL, which includes loading certificates, keys, etc. A lot of the code that parses configuration specific to SSL is also in this file and is covered by the configuration.

Further, there are lines in the `src/http/ngx_core_http_module.c` that enable SSL in the main *listen* function of Nginx. Line 2042 onwards in

`src/http/ngx_http_request.c` upgrades a plain HTTP connection to an HTTPS connection. There are also functions that initiate the SSL handshake and close connections that are part of the covered lines in this file. As for the negative overlap, there were 1230 lines of code that were present in every configuration but this.

8.2.3 Other Findings

`ngx06` disables ETag, and it has two unique lines that are always covered, and they disable ETag in the response header. `ngx07` calls the function `ngx_http_set_expires` from file `src/http/modules/ngx_http_headers_filter_module.c`.

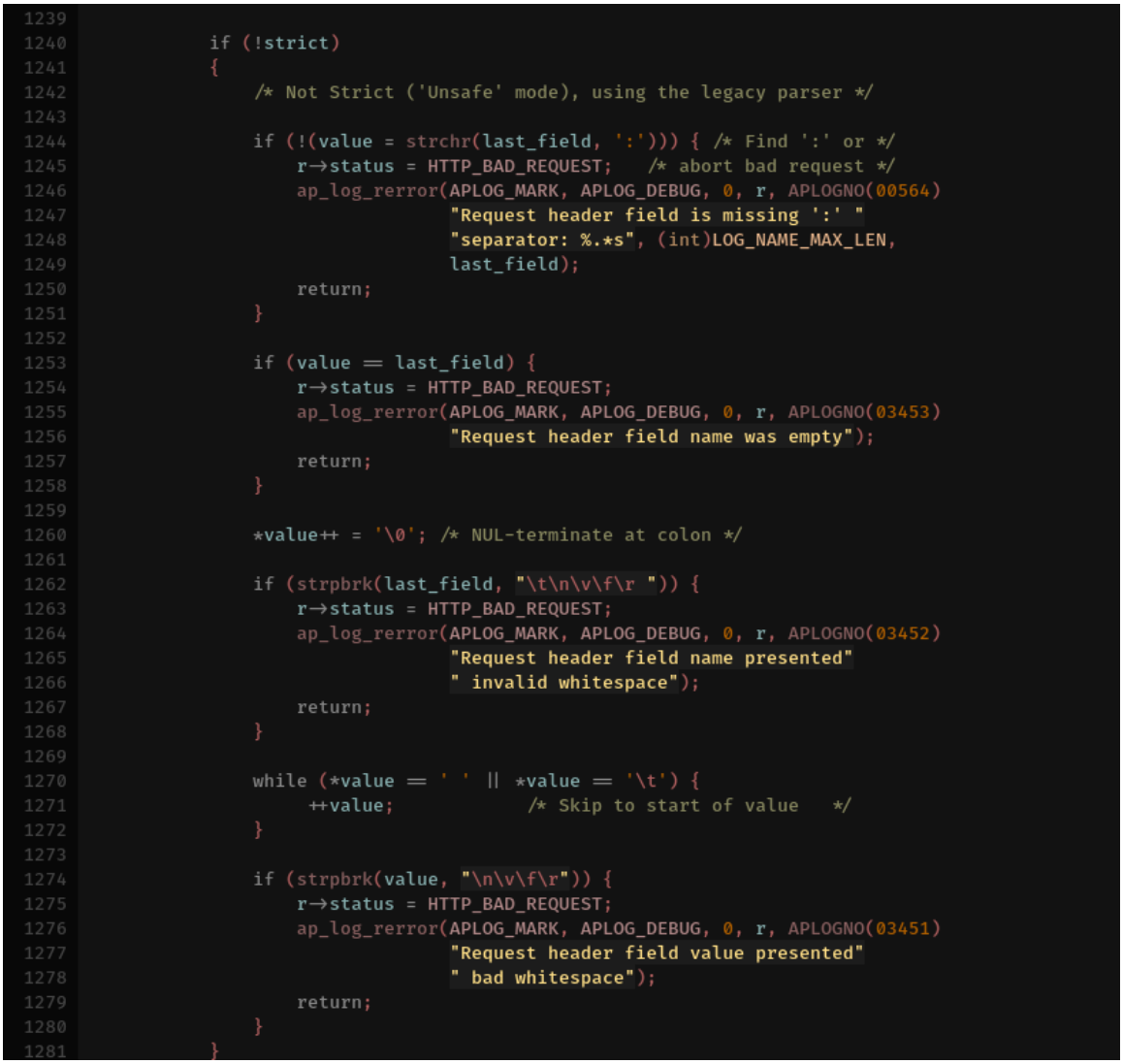
Similarly, `ngx09` sets worker priority, and the only unique coverage that the configuration had was in `src/core/nginx.c` in the function `ngx_set_priority`. All other configurations, similarly, had coverage unique to them as a result of this analysis.

8.3 Apache httpd

8.3.1 Allow Lenient Parsing of HTTP Methods

Since this configuration allows HTTP methods that are not formatted correctly, it is expected to have some parsing code that is uniquely covered by it. Function `set_http_protocol_options` from file `src/server/core.c` parses the actual configuration from the configuration file and sets flags according to the read values.

In the file `src/server/protocol.c`, lines 1240 onwards is covered uniquely in this configuration. This block of code, depicted in Figure-12, shows the legacy parser in action, as specified in the configuration.



```
1239
1240            if (!strict)
1241            {
1242                /* Not Strict ('Unsafe' mode), using the legacy parser */
1243
1244                if (!(value = strchr(last_field, ':'))) { /* Find ':' or */
1245                    r->status = HTTP_BAD_REQUEST;   /* abort bad request */
1246                    ap_log_rerror(APLOG_MARK, APLOG_DEBUG, 0, r, APLOGNO(00564)
1247                                  "Request header field is missing ':' "
1248                                  "separator: %.*s", (int)LOG_NAME_MAX_LEN,
1249                                  last_field);
1250                    return;
1251                }
1252
1253                if (value == last_field) {
1254                    r->status = HTTP_BAD_REQUEST;
1255                    ap_log_rerror(APLOG_MARK, APLOG_DEBUG, 0, r, APLOGNO(03453)
1256                                  "Request header field name was empty");
1257                    return;
1258                }
1259
1260                *value++ = '\0'; /* NUL-terminate at colon */
1261
1262                if (strpbrk(last_field, "\t\n\v\f\r ")) {
1263                    r->status = HTTP_BAD_REQUEST;
1264                    ap_log_rerror(APLOG_MARK, APLOG_DEBUG, 0, r, APLOGNO(03452)
1265                                  "Request header field name presented"
1266                                  " invalid whitespace");
1267                    return;
1268                }
1269
1270                while (*value == ' ' || *value == '\t') {
1271                     ++value;            /* Skip to start of value   */
1272                }
1273
1274                if (strpbrk(value, "\n\v\f\r")) {
1275                    r->status = HTTP_BAD_REQUEST;
1276                    ap_log_rerror(APLOG_MARK, APLOG_DEBUG, 0, r, APLOGNO(03451)
1277                                  "Request header field value presented"
1278                                  " bad whitespace");
1279                    return;
1280                }
1281            }
```

Figure 12. Code Snippet From httpd Executed With Lenient Parsing

8.3.2 Apache httpd Running with TLS Enabled

The line that stands out in `src/server/listen.c` is line 849, which sets a variable called `proto` to 'https'. This line is uniquely covered in this configuration at all times.

In `src/server/ssl.c`, the function `ap_ssl_add_cert_files` is called that adds SSL certificates to the server. This, again, is unique to the configuration.

Finally, some utility files are covered in this configuration. These files are `utils.c`, `utils_md5.c` and `utils_mutex.c`, where the functions provide functionality around registering mutexes, calculating MD5 digests and converting binary data to printable string. As for the negative overlap, there were 1580 lines of code that were present in every configuration but this.

8.3.3 Other findings

There were other configurations that also had very unique code that they covered. For example, `httpd02` that allows the `%2F` character in URIs had a uniquely covered function called `set_allow2f` in `src/server/core.c`, along with line 268 in `src/server/request.c` that normalizes the URI after checking that the character is allowed in the URI.

Interestingly, `httpd04` that allows merged slashes did not have any unique lines. This is mostly because every line that this configuration touches is covered in other configurations too. Generally because those are `if` statements with multiple conditions, and those are evaluated for every configuration.

Configurations that added ETag and Gzipped the response had unique coverage in the files that assist in these functionalities. For ETag, the `src/server/core.c` file also calls a function called `set_etag_bits`, that is unique to the configuration.

Chapter 9

RELATED WORKS

ConfigFuzz[14] is a project that fuzzes the configuration file of a software, given the grammar of the configuration file. The project is aimed towards obtaining the configuration that provides the maximum code coverage.

T-Fuzz[15] is a grey box fuzzing technique that works by mutating the software under test to obtain maximum coverage in the fuzz test. It does so by using a tracer to check where the input fails and removing the check from the target software that causes the input to fail.

As a Proof-of-Concept, I also tried this analysis with QEMU Translation Blocks as the trace information instead of LCOV, and it gave results similar to the ones described in Chapter 6. These results were difficult to get details out of, but for a black-box approach it did give an approximate measure of how much code coverage differed for a specific configuration.

REFERENCES

Richard Stallman, D. M. (1987). *Gnu core utils*. Retrieved March 31, 2023, from https://www.gnu.org/software/coreutils/

Redis [Redis Ltd]. (2009). Retrieved March 18, 2023, from https://redis.io/

Nginx [Nginx Inc]. (2004). Retrieved March 28, 2023, from https://nginx.org/

Apache httpd [Apache Software Foundation]. (1995). Retrieved March 31, 2023, from https://httpd.apache.org

Fioraldi, A., Maier, D., Eißfeldt, H., & Heuse, M. (2020). AFL++ : Combining incremental steps of fuzzing research. *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. https://www.usenix.org/conference/woot20/presentation/fioraldi

Bellard, F. (2005). QEMU, a fast and portable dynamic translator. *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

Usage statistics and market share of web servers, april 2023 [W3 Techs]. (2023). Retrieved March 31, 2023, from https://w3techs.com/technologies/overview/web_server

Redis reviews & customer case studies | redis [Redis Ltd]. (2023). Retrieved March 31, 2023, from https://redis.com/customers/

The self documented redis.conf for redis 6.2 [Redis Ltd]. (2023). Retrieved March 31, 2023, from https://raw.githubusercontent.com/redis/redis/6.2/redis.conf

Nginx documentation [Nginx Inc]. (2022-05-24). Retrieved March 31, 2023, from http://nginx.org/en/docs/

"module index - apache http server version 2.4" [Apache Software Foundation]. (2012). Retrieved March 31, 2023, from https://httpd.apache.org/docs/2.4/mod/

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation.

Oberparleiter, P. (2004). *"LCOV - the Linux Test Project Coverage tool"*. Retrieved March 28, 2023, from http://ltp.sourceforge.net/coverage/lcov.php%22

Zhang, Z., Klees, G., Wang, E., Hicks, M., & Wei, S. (2023). Fuzzing configurations of program options. *ACM Trans. Softw. Eng. Methodol.*, *32*(2). https://doi.org/10.1145/3580597

Peng, H., Shoshitaishvili, Y., & Payer, M. (2018). T-fuzz: Fuzzing by program transformation. *2018 IEEE Symposium on Security and Privacy (SP)*, 697–710. https://doi.org/10.1109/SP.2018.00056