École Polytechnique Fédérale de Lausanne

OpenSSL keys isolation benchmark suite

by Maxime Würsch

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Andrés Sánchez
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 10, 2022

Abstract

Isolation of OpenSSL key needs state of the art techniques to be secure and have good performance. Hence a simple portable benchmark needs to exist to provide a way to compare all the techniques together.

Contents

Chapter 1

Introduction

There exist many projects which try to implement good isolation techniques to improve the security of the software. They need to provide good performance otherwise they will never be used in practice since most software prefers to have a better performance than top-grade security. Hence, it exists a need to get a good benchmark system to evaluate all of them in the same manner and get good information on the difference in performance. It is also important to prove that their overhead is small to be used in practice.

Providing a good benchmark system is not easy. It needs to be simple to set up and to get metrics from it. The portability over different machines and eventually also systems is important.

Most of the papers that have created an isolation technique provides tests to prove the efficiency of their isolation, but it is not always clear to understand how their number has been obtained. The graphs are always different making the observation of the difference more difficult.

The objective is to provide a good and easy benchmark system and to provide some analysis of isolation techniques to isolate the OpenSSL keys.

Firstly, I will present important technologies used in this report. Then the design of the benchmark suite and of the isolation model will be addressed. I will continue by discussing the implementation. After the result will be evaluated. And for finishing related works will be presented before finishing with a conclusion.

Chapter 2

Background

In this chapter, several technologies used in this project are presented.

2.1 Isolation

Isolation is the fact to separate two components from each other. This is done by using a higher privilege manager that controls the interaction between the two isolated components and an abstraction to control how the components can interact together. The security manager monitors all the interactions and stops any request that would violate the property of the isolation [7].

Examples of techniques of isolation are for example; Process isolation, containers, ... During this report, some other isolation techniques will be presented.

2.2 Compartmentalisation

Compartmentalisation is the idea of breaking complex systems into several small parts called compartments. The interaction between all the compartments is controlled through a defined API. This allows isolating the fault in the compartment where it appears. This technique uses the technique of least privilege and isolation. All compartments need to be run with the least privilege to guarantee that the fault is minimal and kept in the least amount of compartments [7].

The two previous techniques need to provide good performance guarantees to be used in practice. If this is not the case, the benefit to use this technique will impact too much the performance of the program and will reduce the utility of the technique.

2.3 OpenSSL

OpenSSL is an open-source software library to secure communication over the network. Its implementation is done in C. It implements SSL and TLS protocols (use in networks to build secure communication) and useful cryptographic functions. The high compatibility with all major systems makes this library used in many machines connected to the Internet, such as web servers. Since this project is huge and used in many critical services. It is important to have a secure and fast implementation of OpenSSL [1].

2.4 Perf tools

Perf is a command-line software to analyse the performance of programs on Linux machines. It allows measuring the performance in both the kernel and the user space of the device. Multiple metrics can be captured such as cycles, cache misses, ... [2]

2.5 Memory Protection Keys (pkeys)

Memory protection keys are a protection mechanism built into the hardware. Currently, only some specific Intel CPUs support it. Pkeys create a mechanism to protect memory on a page table basis. This is done by using four bits in the page table entry that were not used. As this mechanism is implemented on a thread basis. This gives each thread 16 keys to protect memory data on a page basis. A CPU register is used by the kernel to know the type of protection. It can be a read/write or write-only protection. The developer can set protection with pkey on the memory region created with mmap [5].

Chapter 3

Design

3.1 Metrics

There are many different metrics that can be used to perform the analysis of the different isolation systems. In this benchmark suite, the metrics chosen can be separated into different categories: SSL requests, CPU metrics, memory metrics and time.

3.1.1 SSL requests

The SSL metrics are quite simple. You can measure how much time on average a request need to get a reply, the minimum and the maximum time needed for a request to execute over all the requests executed during the experiment. Another possible metric, which was not kept, is to measure the impact of the size of the OpenSSL's key. This one was not kept as most of the systems use the same key size. The size of the key is important for security reasons. Studies measure the performance of the attackers to choose the size of the key to reduce the risk of an attacker can recover the key. The other metrics are useful to see if the isolation model is stable in time and to show the difference in performance between different experiments.

3.1.2 CPU metrics

The following CPU metrics were chosen for this project: Context-switch, CPU migration, cycles, instruction, branches and branches missed. These are the main metrics that can be measured on the CPU and give a good overview of his performance.

A context switch is the action to deschedule a process from the CPU and to store its data to give space for another program to use the CPU. An I/O request or CPU's performance optimi-

sation is the major cause for this context switch. This metric is important since such a switch required a lot of time and therefore has a huge impact on the performance of the program.

Since most of the actual machines run multiple logical cores on a single physical core. The OS can choose to switch the process between different logical cores instead of a context switch since it is cheaper. As the metric before, CPU migration impacts heavily the performance.

Cycles and instructions show the difference in overall size between isolation systems. Since they are cheap, the difference can be big without impacting in a significant way the performance. A change in magnitude will have an impact.

Branching appears when the execution is not linear. Such execution depends on the input and the previous instructions to follow a certain branch between several others. The CPU will try to optimize the execution by guessing which branch will be follow. Therefore, the number of branches missed needs to be as low as possible. Some flags can be put in the program to help the CPU. Thus these metrics can help the developer to optimize the program and help to see how much the performance can be improved.

3.1.3 Memory metrics

The memory is structured in layers. From the ram to the level 1 cache beside the CPU. Each of these levels generates a fault when the requested data is not found and therefore the data need to be retrieved to a lower layer, which is slower. A page fault occurs when the data is not stored in the ram. The other metrics measure the use and the fault of the different caches of the system. Those metrics are useful to understand the memory utilisation of the program.

3.1.4 Timing

Measure the overall time of the benchmark. This is useful to compare if there is a performance overhead between different setups. The limitation of this metric is that it only shows a difference without any detail on where the overhead is located. Thus a more elaborated analysis with more fine-grained metrics is necessary to understand the differences between the different systems

3.2 Benchmark system

The system consists of a client and a server. The client will make requests to the server and measure the time it needs to perform the requests. This corresponds to the metrics defined in the subsection 3.1.1. It will use the same non-modified OpenSSL version for all the executions. All the other metrics will be measured on the server. The server will consist of a web server with one

thread processing all the requests from the client. It will have different versions of OpenSSL for different execution to measure the difference in performance between all the versions. During the experiment, there will be one client who sends requests as quickly as possible to one server. For each of the different versions of OpenSSL, the experiment will be run multiple times to get a better approximation of the result. This system was chosen to get a simple environment easy to set up. It can be useful to reproduce over different machines.

3.3 Data analyses

It will consist of a series of plots. Each plot will compare different metrics. Since each experiment will be run multiple times, the graphs will be a box plot. The box extends from the first quartile (Q_1) to the third quartile (Q_3), with a green line to show the median. The whiskers are drawn at 1.5x the interquartile range ($Q_3 - Q_1$). All points outside this range will be printed on the graph. These graphs are useful to show if the difference between experiments is not just due to some variation of the metrics, but if there exists a real statistical difference.

Another type of graph to analyse the execution of the program is the flamegraph [3]. The program is sampled at a certain frequency and will detect for some metrics where the execution was. This is useful as bottlenecks will be the large area since the function takes more time to execute. The peaks represent the depth of the execution.

3.4 Isolation model

Different isolation model will be compared and also the difference between OpenSSL version: 1.0.2, 1.1.1 and 3.0.1. The different type of isolation will be: ERIM [9], libmpk [6], mpk [5], light-weight Context (lwC) [4] and process isolation.

The first three are based on the technologies of isolation with MPK. The two first are open source projects. ERIM use a binary analysis to ensure that the memory used by the key of MPK is safe and keeps a low overhead for the isolation. libmpk goes further than ERIM, by eliminating the limit of 16 different keys by using virtualization and improving the security over multiple threads. The last one will be an isolation of the private key of the server in OpenSSL.

LwC and process isolation have been implemented by the same open-source project. Unlike the other projects, which run on Linux, these ones run on FreeBSD. The lightweight Context is an OS abstraction which provides isolation like process isolation but in a more efficient way and is kept inside threads. It needs a modified FreeBSD kernel to be implemented.

Chapter 4

Implementation

The programming language chosen to write the script was Python, as it is easy and quick to write small scripts and draw graphs. It has also the advantage to have many modules to solve common issues. Nginx was the software chosen for the web server. It has the advantage to be largely used and multi-platform. It is also easy to build it from the source to pass different versions of OpenSSL. For the client, ssl-handshake [8] was chosen. It is a simple command-line tool written in Go to measure the performance of the SSL handshake.

4.1 Benchmark script

I have written a command-line script which takes into input some arguments to set up the different experiments. You can set the OpenSSL source folder, same for Nginx. Choose which git tag to use for OpenSSL. Finally, some arguments to modify the number of requests and the frequency of sample of perf. The script starts by preparing the Openssl folder with the correct git tag. Then, Nginx is configured to run with the given Openssl and with the debug information. It is necessary to get the name of the functions in the flamegraph. The executable of the server needs to be run as root to get permission to open the HTTPS port. Therefore, the executable need to have the setuid to root, for the script to stop without the need to request the password from the user at the end of the experiment. Once the Nginx is built, it is started. Then, two different perf commands are launched to get the measurement. The first one is `perf stat` which counts the number of events during the total execution. The second is `perf record` which probes the execution at a certain frequency. Now, all the setup is ready. The client is launched to send the given number of requests. Finally, when the experiment is finished, the data collected by `perf record` are filtered to separate cache misses and context switches and flamegraph are created.

4.2 Analyse script

The data of the `perf stat` are parsed in a CSV file. After the CSV file is read with the pandas' module and boxplots are created with matplotlibs for each metric stored in the CSV. There is a column in the graph for each different setup used. The values are re-scaled by the mean of the values got during the experiment of the non-modified OpenSSL version 1.0.2.

4.3 MPK isolation

The MPK isolation uses the same model of isolation as the lwC one. It isolates the private key of the server when it does the key exchange with the client. When the private key is loaded, it is copied into a new memory region freshly allocated and then protected with pkey. The region is set as not writable and not readable. The key is set to readable only when its access is needed during the execution. The program will crash if the number maximal of pkey is reached or if the MPK is not supported by the system. This choice was made to keep security over availability.

4.4 Difficulties

Several problems arise during the project. I will explain the main ones and explain how I overcome them if was able to do it.

Setting the debug flag in the building of the server was not enough for `perf record`. It was needed to modify the parameters of the call graph to `dwarf` instead of the default one. Otherwise, a bunch of unknown functions was present in the flamegraph.

There was a major redesign of the structure of the OpenSSL between versions 1.0.2 and 1.1.1. Therefore it was not possible in a small amount of time to implement the MPK and the lwC isolation in more recent versions of OpenSSL. I also got difficulties copying an `EVP_PKEY` struct in OpenSSL. This is due to some checks done by the functions of the library. This is not enough to just use the `EVP_PKEY` 's copy function or to copy the memory with `memcpy`. Firstly, you need to manually initialize the struct and then you can use the copy function of OpenSSL. But, if you need to copy the RSA key, the function does not do it and you need to do it manually with diverse functions of the library.

LwC needs to run on a modified kernel. This leads to some technicalities to get it working on another device than the original developer's machine. His work was not really portable since some paths in the `Makefile` are absolute paths with his home folder. I also got that when I compile OpenSSL during the compilation of Nginx, the compiler was not found. Thus I need to pass it manually to get it. Unfortunately, it was not enough, Nginx works perfectly until I try

to use HTTPS. I was not able to identify the problem to get a working environment benchmark setup for lwC.

I also got problems with ERIM library. I got an issue during the compilation of Nginx. I try to build it with different versions and also their modified version of Nginx. All attempts were unsuccessful. Until now, I still was not able to build it.

Chapter 5

Evaluation

The benchmark system works great. You can easily choose your arguments. Thus it is easy to use git to change the tag to get the different setup to benchmark. Almost all the process is fully automated. Since perf needs to run with sudo, I cannot find a way to capture the output. Therefore, you need to copy it manually to the parser. Once you got all the results just need to run the script to create all the graphs.

5.1 Result

I will start to analyse the boxplots starting from the more general metrics to the most details ones. Hence, I will start with the total time to run the experiment. The experiment consists to run 300,000 requests in a row while monitoring the performance of the web server. Five runs were done for each setup to get a better vision of how the program behaves in general. All the graphs are represented in per cent in relation to the mean of the non-modified version of OpenSSL 1.0.2.

In the figure 5.1, we can see that as expected the isolation of the OpenSSL 1.0.2 with MPK is a bit slower. We get approximately an overhead of 1.5%. But, more surprisingly, the variance for the new architecture of the server that controls the SSL connection, the variance is quite large. The median of version 1.1.1 is still the same as the 1.0.2. However, for 3.0.1 the median is near to 4% slower. We can observe similar results in the graph of the CPU cycles and task clock. Since these two metrics are similar. I hope to find an explication in the analysis of the other metrics.

As expected the number of instructions is similar. As we can see in the figure 5.2. The augmentation of branches is expected for the MPK isolation since conditions are added to perform the separation. It can be observed in the figure 5.3. For version 1.1.1 there is a decrease and seems to have been better optimized since we can see in the figure 5.4 that the percentage of miss in comparison of branches has decreased to near 8% compared to the version 1.0.2. For

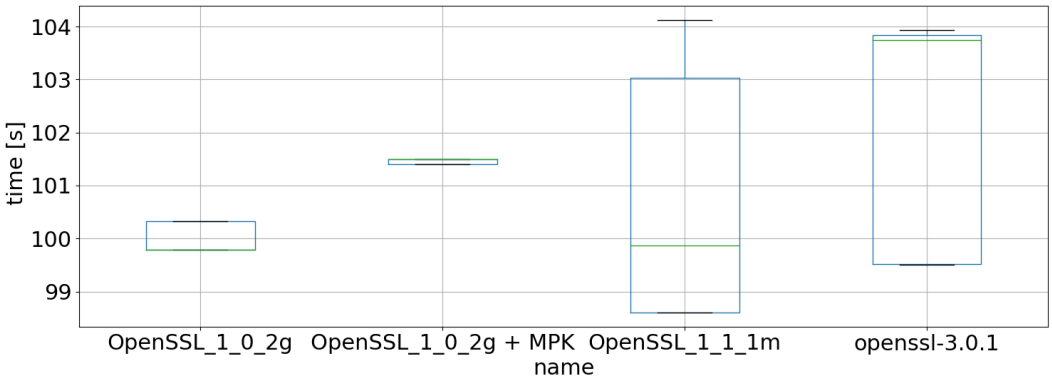

Figure 5.1: Timing plot

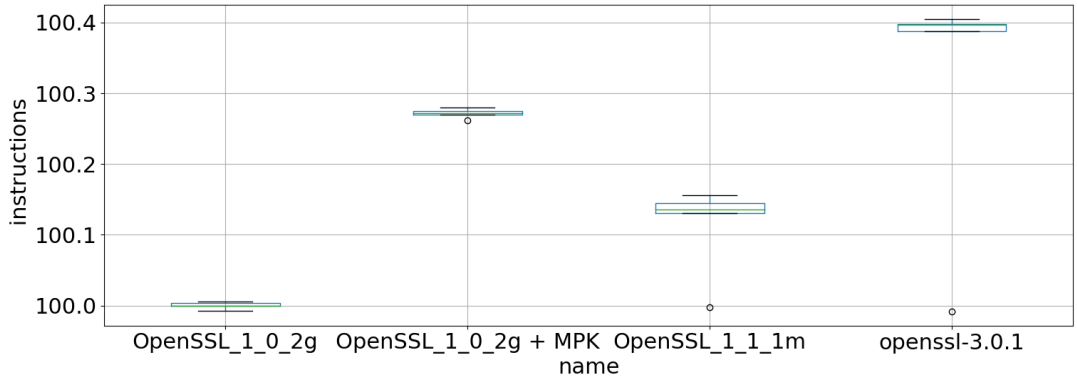

Figure 5.2: Instructions' plot

version 3.0.1 the difference is significantly lower, but this time the diminution of branches does not lead to fewer misses but to the inverse. This can be explained as there was a major update to go to version 3 and thus the code got less time to be optimized than version 1.1.2.

We can observe in the figure 5.5, about page fault. That the use of the isolation with MPK makes more than 1500 times more page faults than the version without it. This comes from that the isolation works on a page-based. Therefore, it is needed to allocate a new page to isolate the key. This situation continues to the other level of cache. With the number of misses going toward the original version as we move to the CPU.

With the metrics we got. I cannot find something relevant to explain why the timing of the version 1.1.1 and 3.0.1 have a larger variance. It will be needed to compare each subversion to find from which version it appears and read the change log to try to understand. A guess could be more asynchronicity was added to the code leading to less time determining code.

The flamegraphs of the cache misses are showing well the impact of the isolation in the

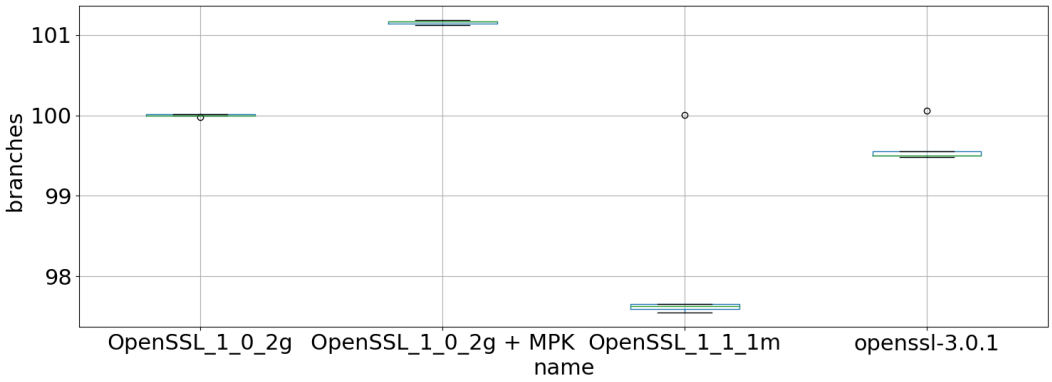

Figure 5.3: Branches' plot

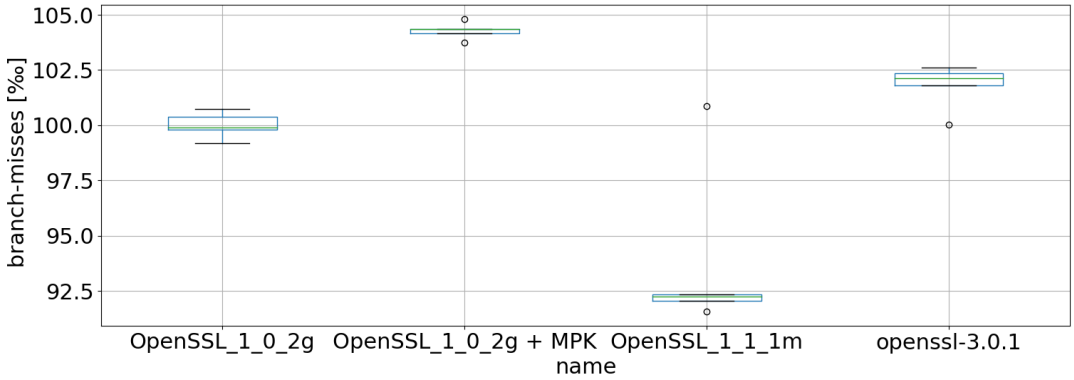

Figure 5.4: Branch miss's plot

OpenSSL version 1.0.2. The non-modified is the figure 5.6 and the one isolated with MPK is the figure 5.7. The colour violet represents all the functions that have SSL in their name. We can clearly see that the second part of the handshake, the one that deals with the key exchange (where the isolation is) takes more time. In the normal version, we got 20% for the first part and 35% for the second. In the isolated, the numbers are 12% and 68%. This is totally expected since to create the isolation another memory part needs to be allocated leading to some cache misses. For the versions 1.1.1 and 3.0.1, the percentage are equivalent but they look a bit different due to a refactoring of the SSL part between the version. You can find them in the appendix. However, the flamegraph for the context switch does not deliver relevant information. They are very similar for the four different experiments.

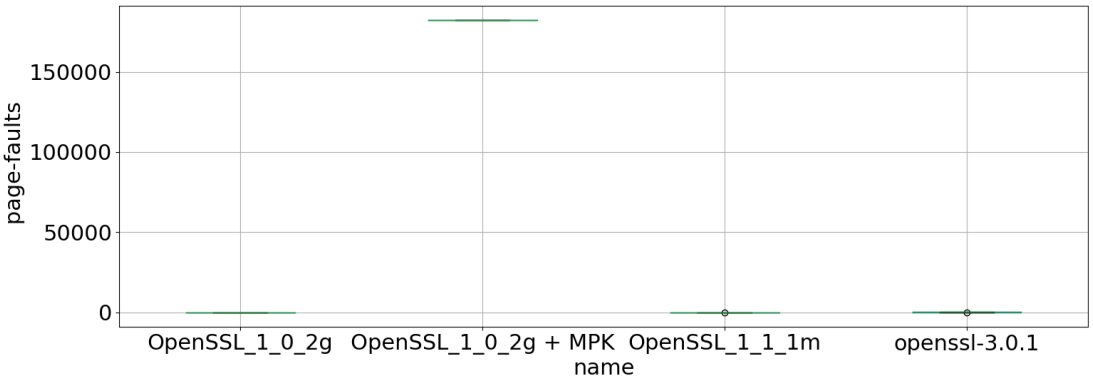

Figure 5.5: Page faults' plot

5.2 Discussion

With the previous results, we can see that the isolation has only a slight impact on the program. It can totally be used in practice. It is only a 1.5% overhead. However, the implemented isolation is only on an RSA key that is used during the key exchange and the isolation was only done in one function. A great amelioration that can be done as future work, will be to refactor the OpenSSL's malloc function to provide the core allocated function protection. It will be also important to look if all the keys used during the execution of the program are stored in such allocated memory. Otherwise, the implementation will not be very useful. This implementation has also a limitation of at most 16 different workers as each of them will use at least one key.

As there is some limitation of the plain MPK isolation, such as the limitation to only 16 different keys and more importantly that you can free a key without having free the protected region. This can lead to leaks of protected data if the adversary managed to control the liberation of the key. Hence, it will be a good idea to use a project that ameliorate the security of MPK like libmpk.

The benchmark suite gives a good overview of the performance of the system with interesting metrics. But, for a large website, the condition will be different. They will use more workers to process the data and the communication will also not just be a handshake. The first change will add difficulties to get the metrics. As you will need to monitor each worker individually. This will lead to more difficulties to understand the overall system. Simulating more real client conditions can be done with more advanced tools like apache bench. This will allow choosing the size of files to transmit to the server. This will lead to moving the bottleneck elsewhere and thus determining maximum usable overhead since the time to transmit the file will take more time than the handshake. By knowing the size of a typical request on the server you will be able to determine the maximum overhead you will accept for the isolation.

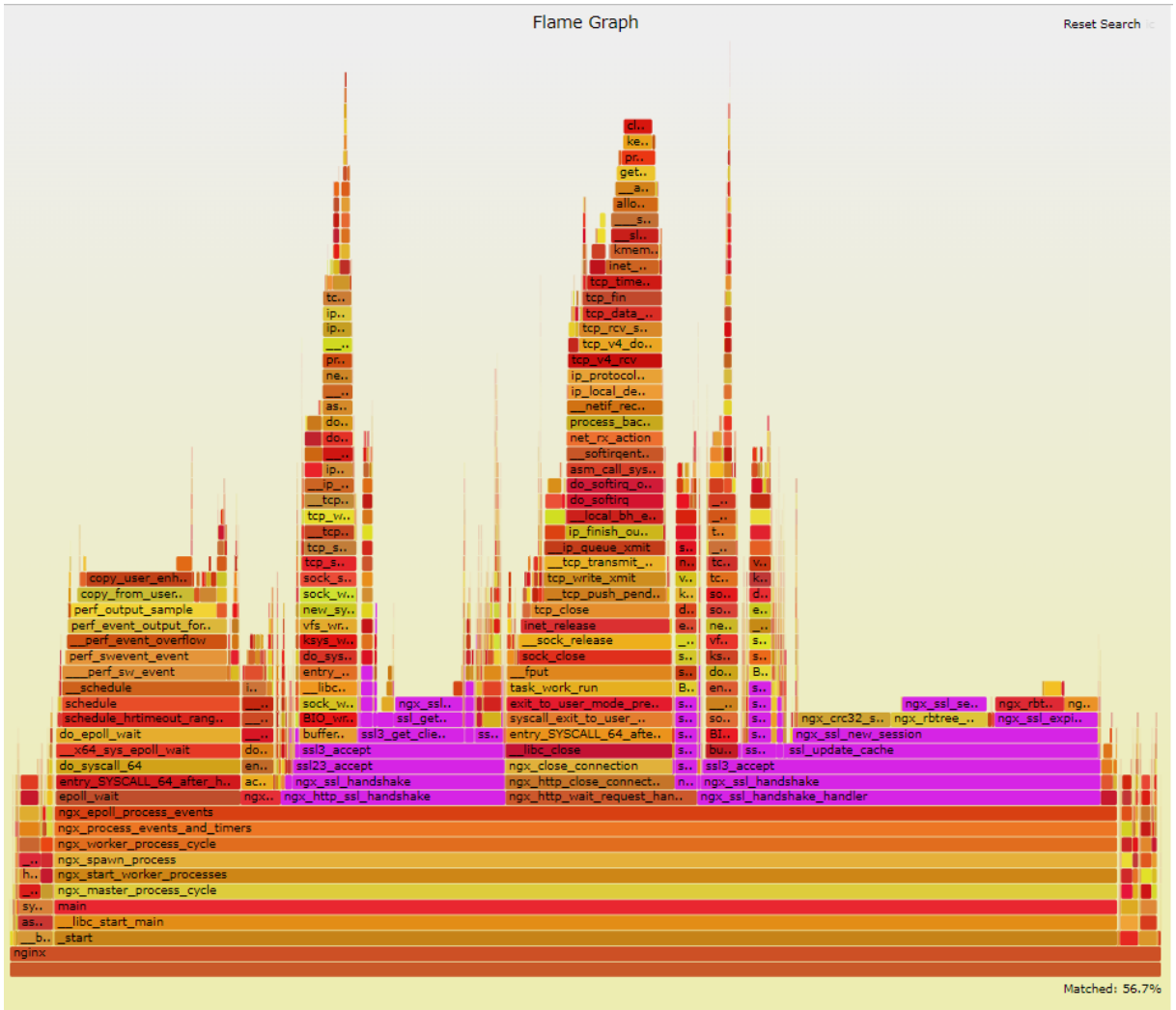

Figure 5.6: Flamegraph of the cache misses of the version 1.0.2

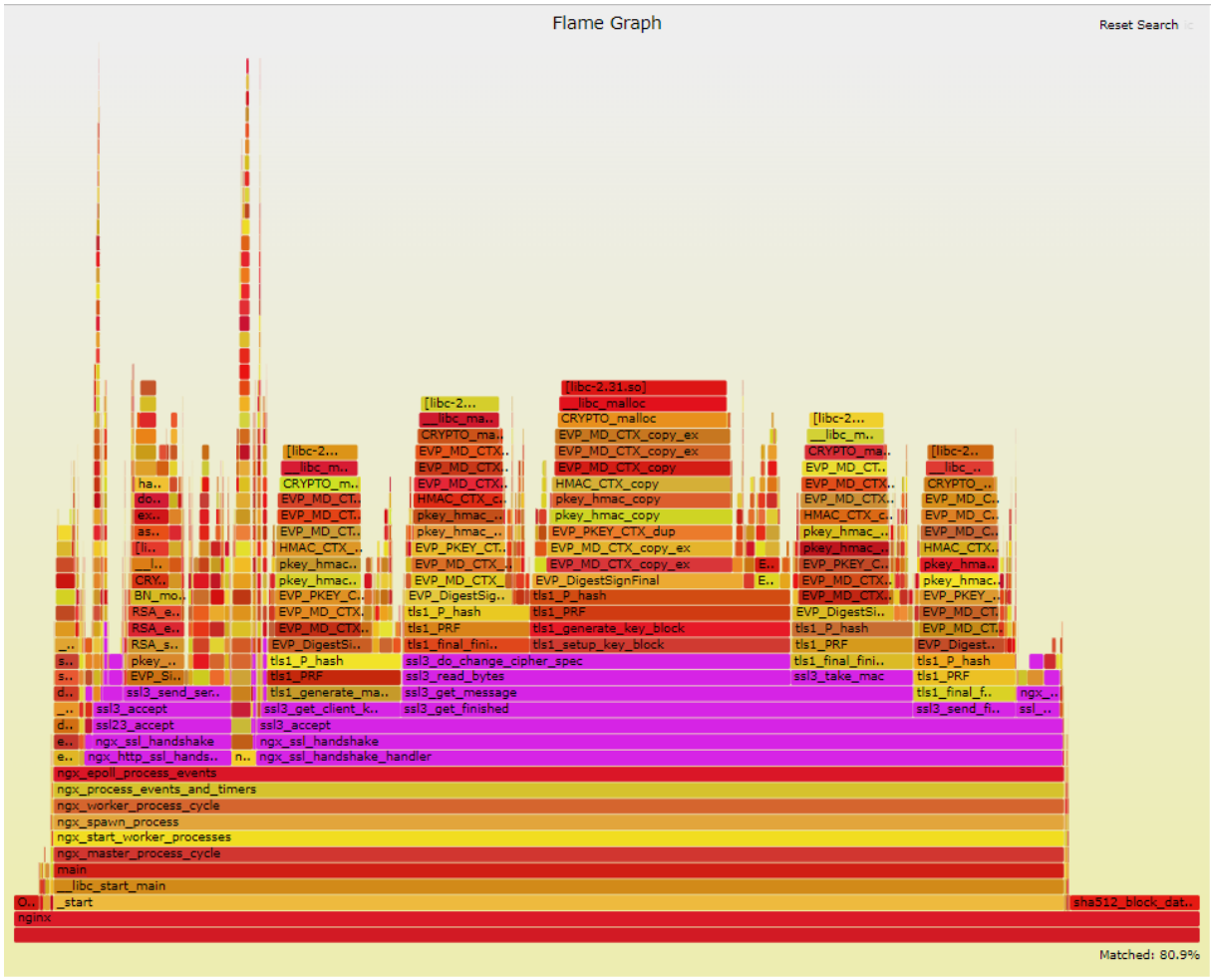

Figure 5.7: Flamegraph of the cache misses of the version 1.0.2 with MPK isolation

Chapter 6

Related Work

Three different isolation techniques will be presented. The first two are based on MPK isolation. And the last one uses a modified kernel to provide isolation similar to the process isolation, but at a lighter cost.

6.1 ERIM

ERIM use MPK normally. But to ensure that the calls to modify the CPU register that manages the permission of the protected region are safe, they use binaries analyses. This allows for providing memory safety to the register. These methods have a bigger overhead than our isolation. It is between 15 and 30 %. The isolation of this system is better than our isolation since all the memory regions of OpenSSL are put in a protected region.

6.2 Libmpk

Libmkp chooses to remove most of the limitations of mpk. Firstly, they use virtualisation of the key for getting rid of the limitation of 16 keys. Protection of the problem of reusing the key after a free and finally make isolation compatible to the `mprotect()`'s permission model. For example. MPK does not let you control the execution as oppose to `mprotect()`. They also make an experiment of isolation of the OpenSSL's keys. This time they replace the `OpenSSL_malloc()` with their own version to provide protection with MPK inside. This method of isolation provides an overhead of less than 1% if only one key is used and when more than a thousand are used it is about 5%. However, it is hard to understand what really means by this number. As their experiments use different file sizes. By looking at the graph, for files of size of 1KB, the number is closer to 15% of overhead. Seem to have a bit more overhead than ERIM, but has the advantage

to get rid of the limitation of 16 keys.

6.3 Light-weight Context (lwC)

As opposed to the two previous light-weight Context (lwC) does not use MPK. They create a new OS abstraction, hence a modification of the kernel is needed to use them. It is an entity orthogonal to the thread and provides memory isolation. They are bound to a process. Their benchmark for isolation OpenSSL key uses the same isolation as my implementation of isolation with MPK. They got less than 1% of overhead. However, their setup is different. They use four workers and a different setup for the clients. This seems to have the same performance as my implementation of MPK isolation.

MPK looks to be a good isolation primitive with the addition of libmpk, but there is a huge disadvantage in that compatible hardware is needed. Currently, the choice of processors is quite limited. Only some Intel processors support MPK. On the other hand, lwC needs only a kernel update to be supported. Both methods seem to have potential.

Chapter 7

Conclusion

During this project, I have implemented a simple benchmark system which consists of one client and one server with one worker. This allows having an easy setup to compare different isolation systems. Most of the project which proposes some isolation systems have their own test system and thus it is difficult to get a clear view of the advantage and disadvantage of each system. Hence, with my project, I propose a way to get comparable results between all isolation systems. I also see that my implementation of isolation of some OpenSSL keys with MPK has a reasonable overhead compared to the other similar isolation.

Many promising isolations with low overhead exist such as libmpk or lwc. For the moment none are largely available due to hardware or kernel limitations. It is important to get good benchmark systems to provide an easy way to compare the different systems and to show that they have a low overhead. It they are great, it is important to deploy them to provide to developers good tools to develop high-security software without sacrificing too much on the performance and on the development time.

Bibliography

[1] OpenSSL's community. *OpenSSL*. URL: https://www.openssl.org/ (visited on 06/04/2022).

[2] perf's community. *Perf Wiki*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 06/05/2022).

[3] Brendan Gregg. *FlameGraph*. URL: https://github.com/brendangregg/FlameGraph (visited on 06/09/2022).

[4] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschelh. "Light-Weight Contexts: An OS Abstraction for Safety and Performance". In: (2016).

[5] *Memory Protection Keys*. URL: https://www.kernel.org/doc/html/latest/core-api/protection-keys.html (visited on 06/05/2022).

[6] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)". In: (2019).

[7] Mathias Payer. *Software Security: Principles, Policies, and Protection*. URL: https://nebelwelt.net/SS3P/softsec.pdf (visited on 06/04/2022).

[8] Puru Tuladhar. *ssl-handshake*. URL: https://github.com/tuladhar/ssl-handshake (visited on 06/09/2022).

[9] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Michael Sammler Nuno O. Duarte, Peter Druschel, and Deepak Garg. "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)". In: (2019).

Appendix A

The rest of the flamegraph

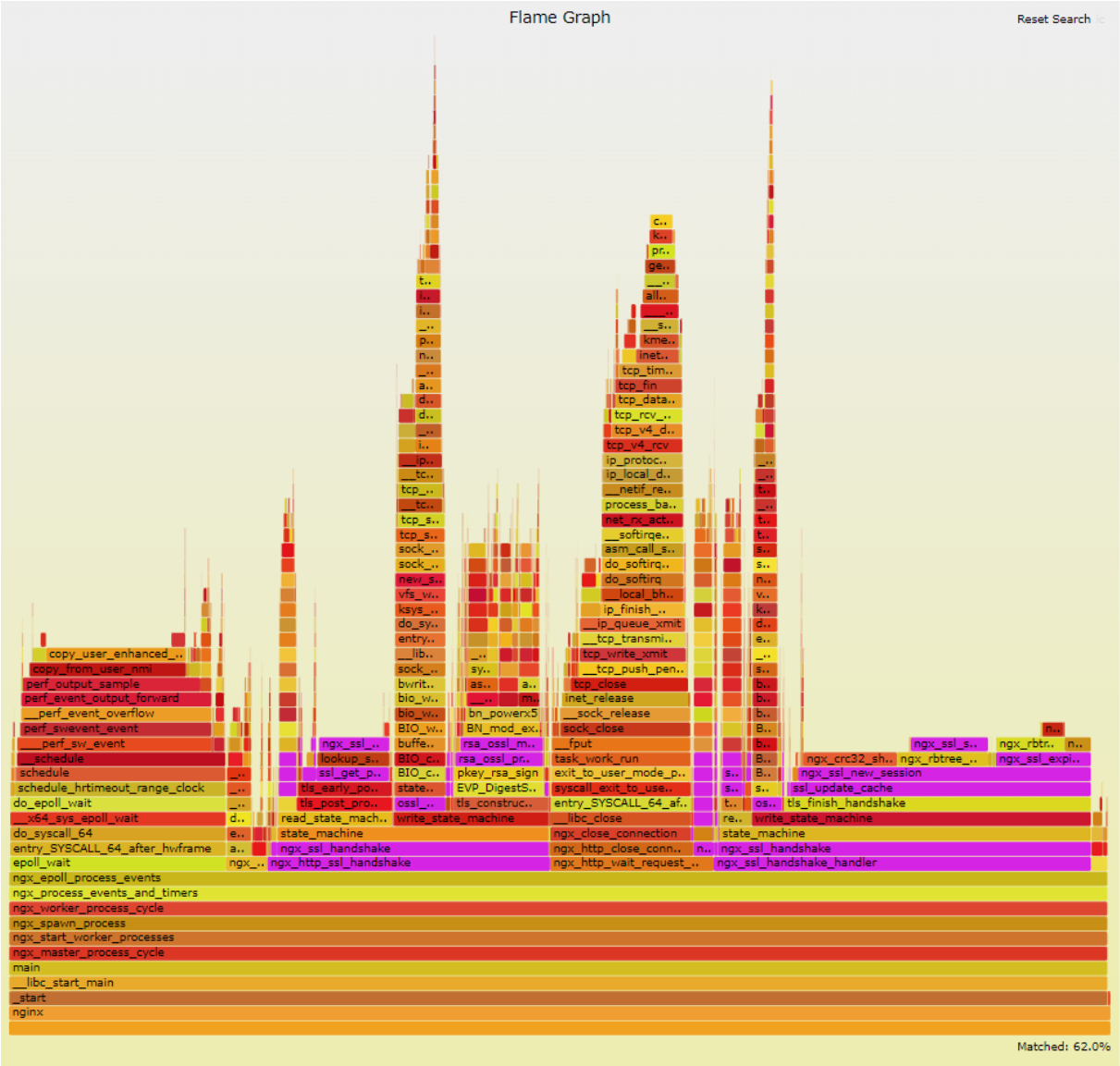

Figure A.1: Flamegraph of the cache misses of the version 1.1.1

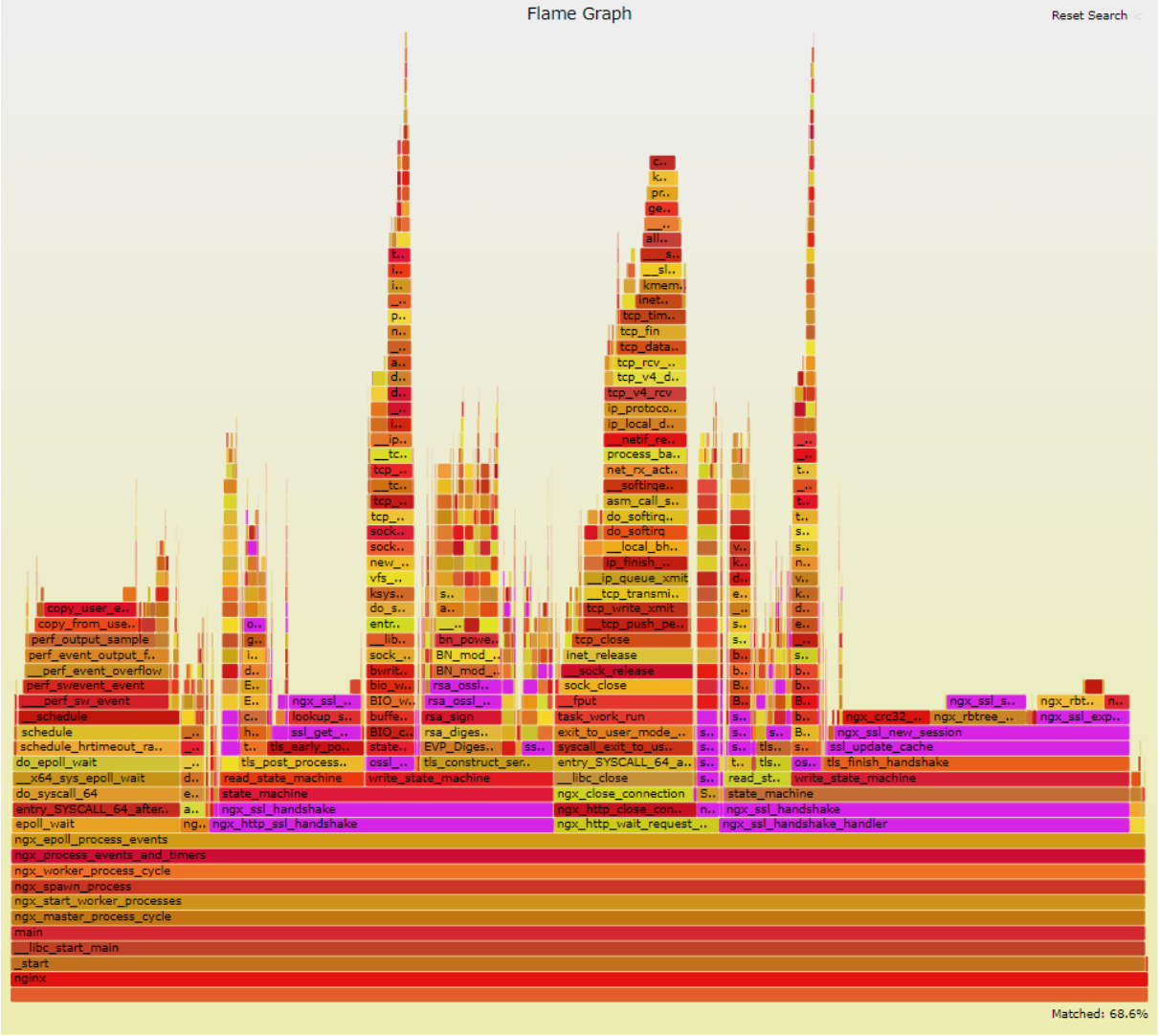

Figure A.2: Flamegraph of the cache misses of the version 3.0.1