ÉCOLE POLYTHECHNIQUE FÉDÉRALE
DE LAUSANNE

BACHELOR PROJECT REPORT

From Ground-Truth Fuzzing to Visualization

Supervisors:
Payer Mathias, Hazimeh Ahmad

Author:
Egli Marc (283232)

June 14, 2020

Abstract: This report deals with the process of benchmarking fuzzers by providing a coherent, simple to understand, visual interpretation of the data.

1 Introduction

1.1 Context

Fuzzing is an automated software testing method that runs a target program on mutations based or generation based inputs. If one of the input crashes the program it will generate a crash report. Compared to manual testing, which requires a lot of human resources to be accomplished and be correct, fuzzing only requires hardware resources. Fuzzing has a high scalability and was proven to be very effective on programs with a lot of code/huge code base [1]. Nevertheless, fuzzing a program takes a lot of time, can be challenging to achieve on limited resources and most importantly doesn't guarantee a result. It is thus important to use the most performing fuzzer among all that exist. To do so, we need a valid metric, that would allow us to compare their performances, and finally find a way to give a clear benchmark result.

Among many benchmark suites like OSS-Fuzz [2] and FuzzBench [3]- Magma [4] is a ground-truth fuzzer benchmark framework that aims to provide this by using popular libraries and re-implementing old bugs into them. Magma evaluates mutation-based grey box fuzzers as AFL [5], FairFuzz [6] or HonggFuzz [7]. All benchmarked fuzzers are ran over all target libraries multiple times. One run is named a campaign. The goal is to provide an understandable and easy to access report that would be generated after magma has finished running.

1.2 Approach

To be able to interpret the data resulting from Magma correctly, it is essential to first understand how Magma works and therefore have a look at the complete process. First of all, we will see how new bugs are added to the target libraries, and how they are selected. Then, once we have run a benchmark of the fuzzers with the newly implemented bugs we will discuss the generation of the report. Finally, we will have a look at the meaning of the resulting graphs created from the data and present in the Magma result report [8].

2 Forward-porting bugs

Magma achieves ground-truth fuzzing by evaluating fuzzers against target libraries where real bugs were injected. These bugs come from an older version of the libraries, hence why this process is named forward-porting, as opposed to back-porting which is the action of implementing newer parts of a software into an older version of it. To understand how Magma works, it's essential to first have a look at how bugs are chosen and implemented into specific target libraries to serve as benchmark for fuzzers.

2.1 Target selection

Magma aims to provide a reliable and realistic results. Therefore magma uses few, widely used libraries as fuzzing targets. Choosing popular libraries that are still used nowadays is very important to provide significant results. Also, it is easier to find bug reports related to them.

First, we added bugs for an already present target in magma which was OpenSSL [9]. OpenSSL is software library for Transport Layer Security and Secure Sockets Layer protocols. The bug reports weren't always complete and a lot of interesting bugs couldn't be injected because the concerned file got refactored.

Once we were more familiar with magma we choose to implement new bugs for PHP [10]. Magma didn't contained a library used for web development. Therefore we found it interesting to see how different fuzzers would behave with this library. The bug reports where very clear and complete. They even contained the Proof of Vulnerability which can be used to ensure that the injected bug works.

2.2 Bug Selection

All the bugs added to Magma are CVEs (Common Vulnerabilities and Exposures). [11] The CVE system is a dictionary of encountered bugs in specific libraries. For every popular library and framework there is a bug list per year, where for every bug the underlying problem is explained and a link is provided that leads to the fix made in the code. A CVE also includes a description of the vulnerability and can be identified by a unique ID. Each CVE also has a CVSS Score that represents the impact that the vulnerability has. This score is computed by taking into multiple criteria. Among those we can find the integrity impact of the vulnerability, the availability impact and also the access complexity. To get more information about a precise bug we can consult the CVE Detail about it [12].

One of the main criteria to look for when adding a new bug is the date of the patch that fixed it. Often a target library like OpenSSL [13] exists for more than 20 years, which means that the code base has changed a lot. Since we are Forward porting bugs, we have to ensure that the behaviour of the code didn't change to be able to trigger it. Thus, selecting a very old bug will rarely pay off as the code part where the fix is, has likely been refactored. This means that a lot of older bugs that would have been significant can't even be injected anymore.

Another other criteria is how the bug was found for the first time. If the bug was originally found by a fuzzer we can be more confident in the ability of fuzzers to detect it. On the other hand if a bug was found manually or by running into it by accident there is a chance that the bug lies in a deep code path. Thus the fuzzer will have a difficult time detecting it, especially if the bug is beyond the coverage wall. These types of bugs can still be interesting to implement.

To be able to know when a bug is triggered or not, we have to provide a Boolean trigger condition to Magma. We thus have to study the underlying program and understand the conditions needed for the bug to happen. However, the conditions can sometimes be very specific and hard to define as a Boolean expression.For example, if the fix covers many different files of the library. A bug should thus be selected if its conditions can be defined as a Boolean expression.

2.3 Implementation

A bug is implemented by creating a patch that then can be added to Magma. As we can see on Figure 1, a patch consists of 3 essential parts.

The first one is the **MAGMA_ENABLE_FIXES** flag, which, if defined at compile time, allows the fixed code to execute, however if it's not, the faulty part will execute.

The second part is the **MAGMA_ENABLE_CANARIES** flag, which can only be defined if the previous flag isn't. This flag allows the execution of an oracle.

An oracle is the third and last part of a patch. It is a Magma function taking two arguments : a unique bug ID as a String and a trigger condition as a Boolean expression. It is only executed if the canaries flag is enabled. First of all, it will report to Magma that the bug was reached. Reaching a bug means that the part of the code where it is located was covered by the fuzzer. Afterwards, it will check if the trigger condition evaluates to true. If it is the case, Magma will know that the bug was reached and also triggered. The oracle always has to be executed before the bug. This is necessary in case the bug crashes the program. The crash would happen before the call to the oracle and Magma would not be able to mark this bug as reached and triggered. The structure of the patch only allows oracles to execute if the faulty code is run.

```

#ifdef MAGMA_ENABLE_FIXES
    //fixed code

#else
#ifdef MAGMA_ENABLE_CANARIES
    MAGMA_LOG(bug_id,trigger_condition); //Oracle

#endif
    //faulty code

#endif
```

Figure 1: Common Magma patch structure

Magma has to be able to know if a bug was reached and even more if it was triggered without altering the program structure. This means that Magma must not create new paths for the fuzzer to explore when checking if a bug was triggered or not. Therefore, if the trigger condition contains logical operators like AND(&&) or OR(||) they must be replaced by the corresponding Magma functions **MAGMA_AND(e1,e2)** , **MAGMA_OR(e1,e2)**. Both functions are using bit-wise operations to avoid the creation of new branches by short-compiler behavior. Bitwise operations always evaluate both sides of the expression. In total, during this process, 17 patches were added for OpenSSL and 22 for PHP. After that, a Magma benchmark was started.

3 Report Generation

We needed a way to represent the results obtained from the benchmark in a report. Also, it was very important to make this report easy to access in a way that everyone could find this report, see it and understand the meaning of the data represented on it. This is why the report is made out of multiple HTML pages.

3.1 General Structure

The report is made of one main page that contains global information and plots concerning the totality of the data. However, for each fuzzer and for each library, an individual page is generated containing specific plots. These pages are generated with the help of templates. There is a library template,a fuzzer template and a main page template that are then filled with the help of the python library Jinja2 [14].

3.2 Program flow

A report is generated by running the main executing the main file and by passing as first argument the path of JSON file where the data of the benchmark is stored. Then, all the plots will be generated from the input data and stored into the *plots* folder. Once all the plots are generated and saved, we use Jinja2 to create the HTML pages from the templates. The templates already contain the path to the plots folder, which allows the generated pages to contain them. To modify and add new plots to the report you simply have to modify the template and be sure that all the new plots are generated into the correct folder.

4 Data Visualization

Magma runs multiple campaigns for each pair of fuzzer and target library and outputs for each of them when a certain bug was reached and when it was triggered. For example, a bug can be reached after 5 seconds, but only triggered after 20 seconds. This data is then represented in the report. With the previously added bugs, Magma contains a total of 121 injected bugs, but after analyzing the data 72 were reached and only 38 were triggered. In this section we will discuss multiple representations of the data included in the generated report, and what we can conclude from it. The represented graphs were plotted with data obtained after running 10 campaigns of 24 hours for each fuzzer, target pair. Running multiple campaigns is very important for Magma because each of them are independent. Furthermore, Magma is evaluating grey box mutation-based fuzzers which have evolving states during a campaign. This means it is important to run a lot of campaigns to get an approximation of the overall distribution of the number of bugs found and reached.

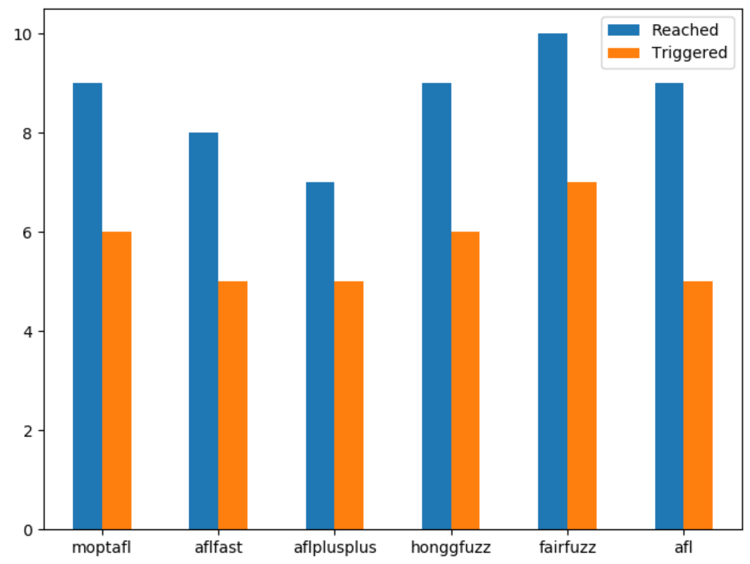

Figure 2: Number of unique reached and triggered bugs in Libtiff

4.1 Total number of unique bugs

The first, and most basic information to display is the number of unique bugs reached and triggered per library for all fuzzers. This plot can be found in every specific library page and aims to give a general idea of the performance of the fuzzers against each other. As we can see on Figure 1, this plot also compares the number of bugs triggered against the number of reached ones for every fuzzer. However, it doesn't show which specific bugs were reached and which out of them were triggered.

4.2 Mean number of bugs found

Another relevant information to display is the mean number of bugs triggered by all the fuzzers on each target, alongside the standard deviation.

This representation helps to get an idea of how many bugs are triggered on average. The included standard deviation then represents the general distance to the mean. A high standard deviation is typically representative of fuzzers that are finding a lot of bugs only in a few campaigns and are not doing well during the rest of them. Furthermore, two fuzzers having the same mean number of bugs found on the same target can be differentiated through the standard deviation.

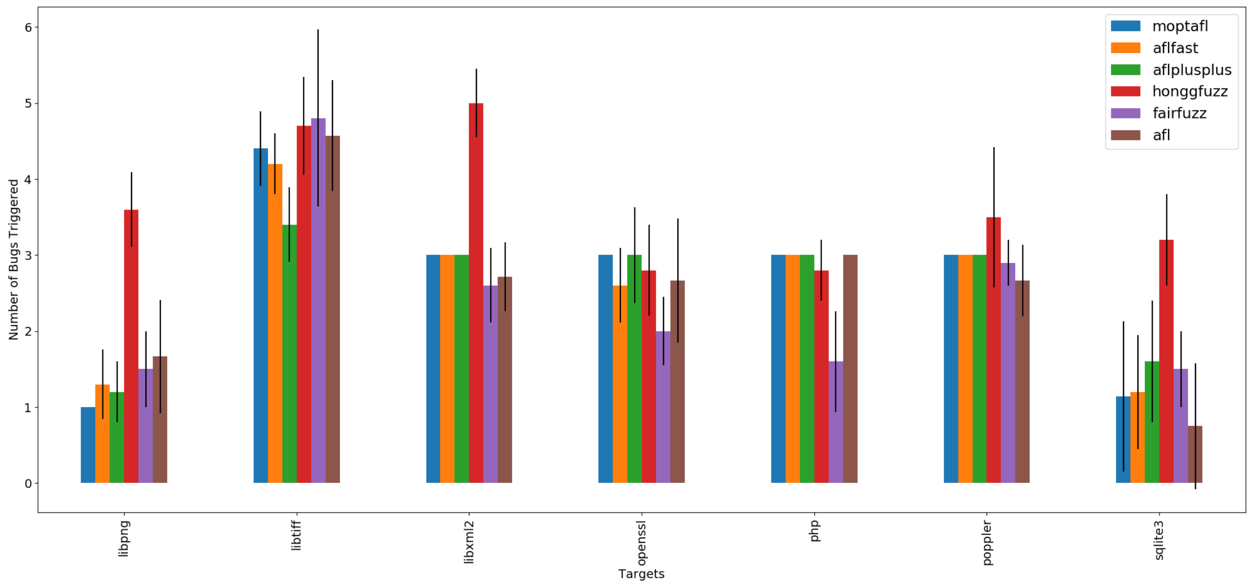

Figure 3: Mean number of bugs triggered for each target library

This plot aims to take into account the different campaign results and to identify the general trend of the number of bugs found. On the other hand, this representation doesn't provide any information about what bugs were triggered. It also doesn't show when during a campaign the bugs were triggered.

4.3 Expected time-to-trigger-bug

Finally, an important metric to evaluate the fuzzers is the expected time-to-trigger-bug, stated in the Magma paper [15]. The formula is used as an over approximation as it fits the black-box fuzzer behavior where there is no memory of previous events and inputs are random.

A grey-box mutation based fuzzer on the other hand evolves its inputs during a campaign according to the result of the previous ones. This metric penalizes the expected time if a bug was only found M times in N campaigns with M < N.

$$E(X) = \frac{M \times \bar{t} + (N - M) \times \frac{T}{\lambda_t}}{N}$$

$$where : \lambda_t = \ln\left(\frac{N}{N - M}\right)$$

If a bug is found in all N campaigns, then the expected time-to-trigger-bug will be inferior to the length of one campaign which in this case is represented by T. However, if a bug is not found in all campaigns the expected time-to-trigger-bug will be greater than T. We can use this to calculate the aggregate time which is simply the expected time-to-trigger-bug over all fuzzers for one bug. A bug with a small aggregate time will have been found by all of the fuzzers in nearly all campaigns as a a bug with a big aggregate time will have been found by certain fuzzers and not in all campaigns. Therefore, aggregate time gives a usable metric to order bugs by their complexity.

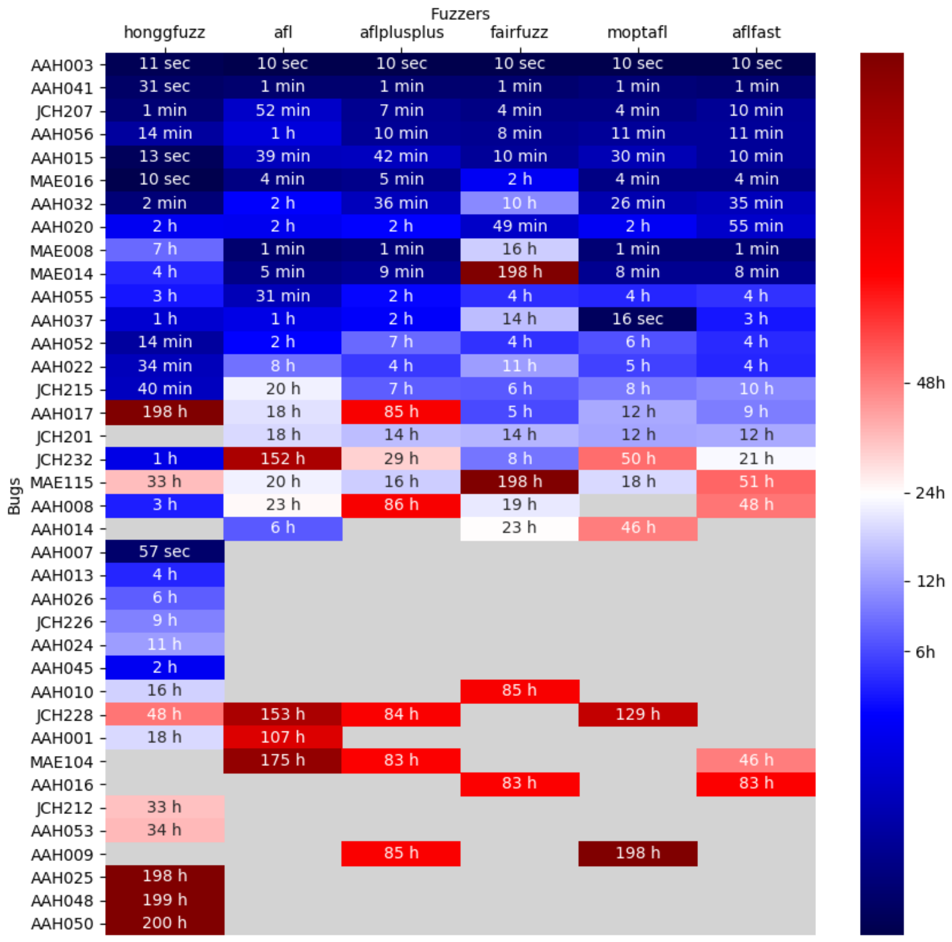

Figure 4: Expected time-to-trigger-bug

A very complex bug that was found only once in 10 campaigns by one fuzzer is not significant enough to assume that this particular fuzzer is the best performing one. Performance is deducted from the general trend of the samples, thus finding a bug only once is not representative of the general trend and is considered as an outlier.

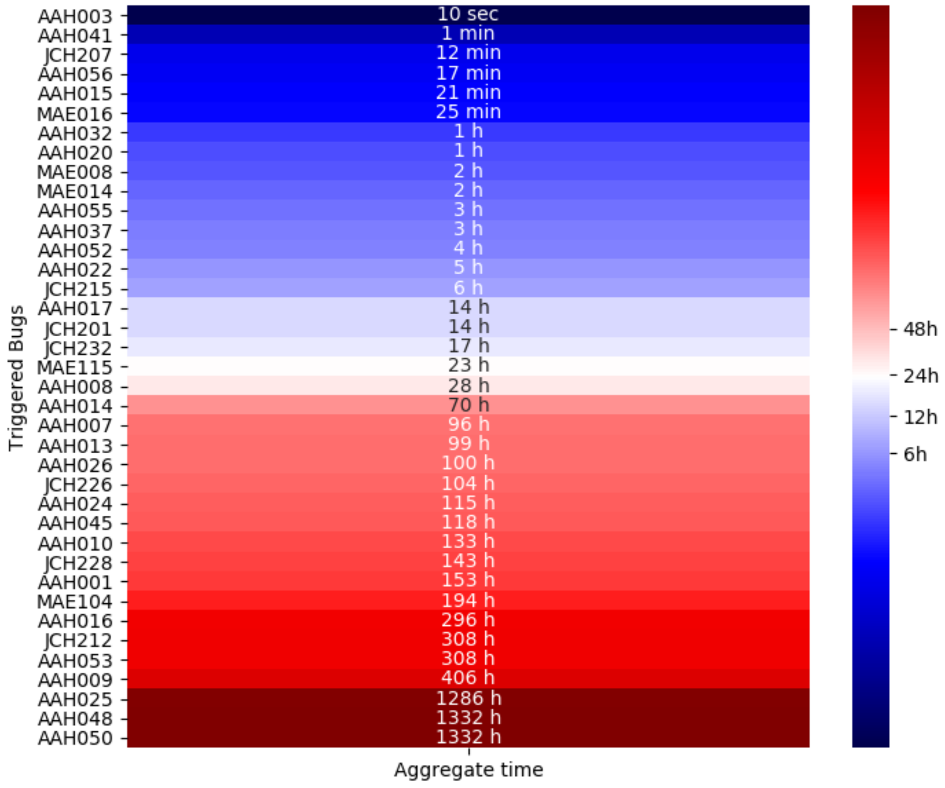

Figure 5: Aggregate time-to-trigger-bug

5 Result Analysis

On Figure 3 we can see that Honggfuzz is performing better on targets where other fuzzer struggle to find bugs in. For example, a lot of fuzzers found a mean of 3 bugs in libxml2 where Honggfuzz found a mean of 5 with the same standard deviation. Fair-fuzz on the other hand didn't perform well on many targets, but found the most number of bugs in libtiff. This shouldn't be taken at face value, due to the huge standard deviation going with it.

It is not correct to assume from Figure 4 that Honggfuzz did better then all other fuzzers because of the last three bugs - *AAH025* [16], *AAH048* [17], *AAH050* [18]- that were very hard to trigger. This is because all the three have an expected time nearly equal to 200h which corresponds to a bug only found once in 10 campaigns. As given in the Magma paper [15], such bugs do not represent the general trend of the samples, and are therefore not relevant for the benchmark. The more a bug has an expected time to trigger close to 24 hours, the more it will be useful for the benchmark. A good example are the bugs *JCH226* [19], *AAH026* [20] and *AAH013* [21] , which have an expected time to trigger below 24 hours for Honggfuzz but weren't even triggered by the other fuzzers. All of these bugs have similar trigger conditions as all of them require a variable to equal a precise value. We can conclude that this could be one of the strength of Honggfuzz. This also shows the weakness of fuzzing, exploring paths that can only be reached if some very specific conditions are met is very hard to accomplish for fuzzers. This behaviour can be related to the coverage wall which happens when fuzzer do not manage to explore more code paths after some time.

It is also interesting to have a look at bugs that are considered as easy because some of them reveal weaknesses of precise fuzzers. Bug MAE014 [22] is one of those. Every fuzzer did well with it and found it every time except FairFuzz that only found it once. Figure 4 shows that Fair Fuzz found less bugs in 3 targets than all other fuzzers. One of this target is PHP , the target where the patch MAE014 was applied. Such bugs also help to identify the strength and weaknesses of the evaluated fuzzers.

All the result show that different fuzzers all have different strength and weaknesses to detect certain type of bugs. Globally Honggfuzz did best and Fairfuzz was the less consistent overall by finding some "easy" bugs only by luck once in 10 campaigns. AFL [5] , AFL++ [23], MOpt-AFL [24] and AFLfast [25] had nearly equal performances. They all found the same bugs consistently with exception of 1 or 2. We can also see that they found the same bugs by luck which shows again that they have a very similar behaviour. The fact that the magma benchmark shows that all of those fuzzers are performing in a very similar way is very encouraging as AFL++, MOpt-AFL and AFLfast were all build on top of AFL. In the same way Fairfuzz was also build on top of Fairfuzz but it seems like that the number of bugs used is still not high enough to show its strength.

6 Conclusion

In conclusion, we were able to generate an easy to access and simple to understand report containing multiple plots representing the data of a magma benchmark. The visualization of the data allowed us to clearly conclude that Honggfuzz is the fuzzer that did the best overall. It also helped showing the individual strength of the different fuzzers.

To extend the number of plots displayed on the report and to achieve a more precise analysis of the fuzzers, it would be interesting to have access to the real time coverage of a target in a campaign. This would allow to compute the trend of the coverage among all campaigns. Adjusting the duration of one campaign could also be interesting for the benchmark as fuzzers would have more time to find bugs.

To further improve the effectiveness of the benchmark we could analyze the bugs that were reached , triggered and those that weren't. Each bug has a vulnerability type and has also an *Access complexity*, represented in Figure 6 and 7. This could be used to infer some rules that would be able to predict if a bug has a good potential to be reached or not. We tried this with the newly injected bugs but the number of samples was too low to infer reliable rules. Thus, adding new bugs is also an important future point. It would also be interesting to use the proof of vulnerability of the non-reached bugs to ensure that they are indeed reachable and triggerable.

References

[1] Code size visualization. https://www.informationisbeautiful.net/visualizations/million-lines-of-code/. Accessed: 2022 06-11.

[2] Gooogle oss-fuzz project. https://google.github.io/oss-fuzz/. Accessed: 2022 05-30.

[3] Gooogle fuzzbench project. https://google.github.io/fuzzbench/. Accessed: 2022 05-30.

[4] Magma repository. https://github.com/HexHive/magma. Accessed: 2022 06-09.

[5] Google afl repository. https://github.com/google/AFL. Accessed: 2022 05-30.

[6] Fairfuzz repository. https://github.com/carolemieux/afl-rb. Accessed: 2022 05-30.

[7] Google honggfuzz repository. https://github.com/google/honggfuzz. Accessed: 2022 05-30.

[8] Magma report sample. https://hexhive.epfl.ch/magma/reports/sample/. Accessed: 2022 05-30.

[9] Openssl official website. https://www.openssl.org/. Accessed: 2022 06-11.

[10] Php official website. https://github.com/php/php-src. Accessed: 2022 06-11.

[11] Common vulnerabilities exposures. https://cve.mitre.org/. Accessed: 2022 05-30.

[12] Common vulnerabilities exposures details. https://www.cvedetails.com/. Accessed: 2022 05-30.

[13] Openssl repository. https://github.com/openssl/openssl. Accessed: 2022 05-30.

[14] Jinja2 library. https://pypi.org/project/Jinja2/. Accessed: 2022 05-30.

[15] Ahmad Hazimeh. Magma: A ground-truth fuzzing benchmark. 2020. Accessed: 2022 05-30.

[16] Cve detail for aah025. https://www.cvedetails.com/cve/CVE-2017-0663/. Accessed: 2022 05-30.

[17] Cve detail for aah048. https://www.cvedetails.com/cve/CVE-2019-10872/. Accessed: 2022 05-30.

[18] Bug detail for aah050. https://bugs.freedesktop.org/show_bug.cgi?id=106061. Accessed: 2022 05-30.

[19] Cve detail for jch226. https://www.cvedetails.com/cve/CVE-2017-2518/. Accessed: 2022 05-30.

[20] Cve detail for aah026. https://www.cvedetails.com/cve/CVE-2017-7375/. Accessed: 2022 05-30.

[21] Cve detail for aah013. https://www.cvedetails.com/cve/CVE-2016-10269/. Accessed: 2022 05-30.

[22] Cve detail for mae014. https://www.cvedetails.com/cve/CVE-2019-9638/. Accessed: 2022 05-30.

[23] Afl++ repository. https://github.com/AFLplusplus/AFLplusplus. Accessed: 2022 06-11.

[24] Mopt-afl repository. https://github.com/puppet-meteor/MOpt-AFL. Accessed: 2022 06-11.

[25] Afl fast repository. https://github.com/mboehme/aflfast. Accessed: 2022 06-11.

Injected Bugs in OpenSSL				
Bug ID	CVE ID	Vulnerability Type(s)	Access Complexity	CVE Score
MAE100	CVE-2016-2105	Denial of Service	Low	5.0
MAE101	CVE-2016-2106	Denial of Service	Low	5.0
MAE102	CVE-2016-6303	Denial of Service Overflow	Low	7.5
MAE103	CVE-2017-3730	Denial of Service	Low	5.0
MAE104	CVE-2017-3735	Overflow	Low	5.0
MAE105	CVE-2016-0797	Denial of Service Overflow Memory corruption	Low	5.0
MAE106	CVE-2015-1790	Denial of Service	Low	5.0
MAE107	CVE-2015-0288	Denial of Service	Low	5.0
MAE108	CVE-2015-0208	Denial of Service	Medium	4.3
MAE109	CVE-2015-0286	Denial of Service	Low	5.0
MAE110	CVE-2015-0289	Denial of Service	Low	5.0
MAE111	CVE-2015-1788	Denial of Service	Medium	4.3
MAE112	CVE-2016-7052	Denial of Service	Low	5.0
MAE113	CVE-2016-6308	Denial of Service	Medium	7.1
MAE114	CVE-2016-6305	Denial of Service	Low	5.0
MAE115	CVE-2016-6302	Denial of Service	Low	5.0
MAE116	CVE-2016-2196	Denial of Service Execute Code Overflow	Low	10.0

Figure 6: OpenSSL bugs

Injected Bugs in PHP				
Bug ID	CVE ID	Vulnerability Type(s)	Access Complexity	CVE Score
MAE001	CVE-2019-9020	Read after free	Low	7.5
MAE002	CVE-2019-9021	Heap Buffer over-read	Low	7.5
MAE003	CVE-2019-9025	Overflow	Low	7.5
MAE004	CVE-2019-9641	Overflow	Low	7.5
MAE005	CVE-2019-6977	Overflow	Low	7.5
MAE006	CVE-2019-11041	Overflow	Medium	6.8
MAE007	CVE-2019-11042	Overflow	Medium	6.8
MAE008	CVE-2019-11034	Overflow	Low	6.4
MAE009	CVE-2019-11039	Overflow	Low	6.4
MAE010	CVE-2019-11040	Overflow	Low	6.4
MAE011	CVE-2018-20783	Out of bound read	Low	5.0
MAE012	CVE-2019-9022	Out of bound read	Low	5.0
MAE013	CVE-2019-9024	Out of bound read	Low	5.0
MAE014	CVE-2019-9638	Overflow	Low	5.0
MAE015	CVE-2019-9640	Overflow	Low	5.0
MAE016	CVE-2018-14883	Overflow	Low	5.0
MAE017	CVE-2018-7584	Overflow	Low	7.5
MAE018	CVE-2017-11362	Denial of Service Overflow	Low	7.5
MAE019	CVE-2014-9912	Denial of Service Overflow	Low	7.5
MAE020	CVE-2016-10159	Denial of Service Overflow	Low	5.0
MAE021	CVE-2016-7414	Denial of Service Overflow	Low	7.5

Figure 7: Php bugs