Bachelor Project - Evaluating Fuzzer Benchmark

By Cosme Jordan

Supervisor: Prof. Dr. sc. ETH Mathias Payer

Adviser: Ahmad Hazimeh

Laboratory: HexHive

June 14, 2020

1 Introduction

Today, we live in a never ending growth of the number of softwares developed around the world. The number of bugs and possibilites of bugs are also constantly increasing. Indeed, according to the author Steve McConnell [1], there are on average a few dozens of bugs per thousands of lines of code. To be able to find bugs, there are tools called fuzzers. The principle behind fuzzers is that fuzzers create random inputs and can mutate them depending on the type of fuzzer. Fuzzers have been able to find thousends of bugs. Software engineers also need to find out bugs as fast as possible and to do so, they have to use the best fuzzers to meet their needs. Benchmarking is the best way to determine what a fuzzer can do and what capabilities it has. In the current day, most benchmarks of fuzzers are made by counting the number of bugs found in an elapsed time and on different campaigns. This solution isn't that efficent. Indeed, current benchmarks of fuzzers only tell us about the number of bugs found, but it doesn't say anything about what kind of bugs it has found. It doesn't say anything about the number of unique bugs it can find. For example, if one benchmarking result finds 42 bugs. Does it mean 42 different bugs ? Or are all the same bug ? To cope with this problem, researchers from the EPFL have developed a new way to benchmark fuzzers by developing a software called Magma [2].

Let's now talk about fuzzers. The idea behind fuzzers is that each fuzzer works by creating random inputs for softwares and tries to find new bugs over different campaigns and different iterations. What one does with the input depends on the type of the fuzzer. In consequence, fuzzers come in different ranges and types.

Now, we could ask the question what is a good fuzzer benchmark. A good one must have the following properties:[3]

- Diversity: A fuzzer benchmark must have a sufficient number of bugs. Indeed, one must have a sufficient number of bugs or else it would be difficult to have a sufficient comparison between fuzzers.
- Accuracy: A fuzzer benchmark must give consistent results. Indeed, it wouldn't be good tests if one benchmark says that fuzzer A is the best and another says that fuzzer B is the best.
- Usability: A fuzzer benchmark should be simple to use and deploy. Researchers must be able to reproduce the results.

Let's now give a few definitions that we will use later.

- A bug is defined by its identifier.
- We say that two bugs are the same if they have the same identifier. Otherwise they aren't the same. For example, if we have two identifiers A and B which has the same type. Then both of them are different bugs.
- We call a library used by a fuzzer a *target*.

- If a fuzzer reaches a bug but the bug isn't found with the particular condition of the bug we will say that the bug is *reached*.
- If a fuzzer reaches a bug and the condition is right for the bug to be triggered and the fuzzer finds the bug then we say the bug is *triggered*.

1.1 The idea behind Magma

Magma has beeen created to have better fuzzer benchmarks. Magma injects bugs into real world softwares and targets and records whether a bug was *reached* or *triggered* [3].

To inject a bug we have to use an *Oracle*. An *Oracle* is simply a call to the following Magma function:

```
// C code
MAGMA_LOG(char* bug_identifier, bool trigger_condition)
```

An *Oracle* allows us to track a bug and find if it was either *reached* or *triggered*.

The `bug_identifier` allows Magma to know precisecly what bug was *triggered* or *reached*. At the time of the writing, the `bug_identifier` is a string: three letters followed by three numbers. Each bug has a corresponding identifier. The `trigger_condition` allows Magma to know precisely if the bug was *triggered* by a fuzzer.

This function allows Magma to know when a fuzzer arrives at a particular location. It allows Magma to know when a bug is either *reached* or *triggered*. This in turn allows Magma to collect informations about bugs that have been inserted.

Magma can then run campaigns. Magma can be run independently of the machine. Though, it must be able to install and run dockers. Currently, Magma can be run and installed on any computer. The creation of the docker will install everything Magma needs to run campaigns.

The following command is used to install Magma on a *docker*:

```
#!bin/bash
FUZZER=fuzzer_name TARGET=target_name ./build.sh
```

First, one has to move to the tools/captain folder. Then we just have to run the previous command and the build script will create a docker and install the corresponding `fuzzer_name` and `target_name`. Then we can use the following commands once we enter the docker

```
#!bin/bash
mkdir -p ./workdir
FUZZER=fuzzer_name TARGET=target_name PROGRAM=program_name\
SHARED=./workdir POLL=polling_time\
TIMEOUT=timeout_time ./start.sh
```

to start a campaign. We can choose the fuzzer, the target and the length of the campaign.

1.2 The idea behind the project

As Magma is fairly a new project. There are still a lot of work to do. Fo example, Magma has to increase its number of targets and its number of bugs.

My work is to add new bugs and to add a new library. Then I have to understand how Magma works so that I could implement webpages and create graphs for evaluating fuzzer benchmarking.

We will first start with the bugs that was implemented in Magma. After this, we will continue on to speak about the webpages and how they were designed. We will also talk about the different graphs that are displayed on those pages and how they can be useful. At the end, we will simply benchmark fuzzers using those graphs.

2 Implemented Bugs

As one the goal of Magma is to benchmark fuzzers and it has to comply with the *diversity* requirement. Then more bugs has to be added to Magma and especially a sufficent number has to be added to each target.

For both libraries, we tried to implement as much bugs as possible and as different as possible. Indeed, the most bugs a target as, the better it is because we must have many bugs which make comparisons more thorough and complete. Though, even if the type of the bug has been given and that at first glance it would be a good idea to see what type of bugs a fuzzer is better with. The type of the bug won't affect the benchmark because even if an injected bug is a simple bug. The bug still can be difficult to find for a fuzzer because it could be hidden deeply inside the code source itself. Furthermore, no fuzzer that Magma supports works to find specific kind of bugs but tries to find every kind of bugs. On the other hand, one could try to look how deep a bug is in the source code but this isn't practical and wouldn't be an efficient use of time as the fuzzers themselves usually record the coverage of their search.

Unfortunatley, other bugs couldn't be implemented. Here are some reasons as why it was the case:

- It can be difficult to find the correct trigger conditions as many files could have been changed and that part of the code that was implemented a few years ago doesn't exist anymore.
- They are a lot of changes to make in the source code. It would be a complicated and tedious task to find out if the bug can still be triggered as a lot of the code doesn't exist anymore. Furthermore, if we make too much changes, the bug may no longer be *triggered*.
- Most of the time when we implement a bug, it is better to find a simple trigger condition, but at times making changes can be complicated and impact a lot of files and quite a lot of logic.
- Some bug reports are lacking in details and at times it is even quite difficult to find out what caused a bug.

At times, we may have one or more of the above reasons as to why a bug can't be imported in *Magma*. The following bugs have been added to *poppler* and *sqlite3*. Let's start with *poppler*.

2.1 Poppler

Below is a list of bugs that could be implemented:

Bug ID	CVE	Type of bug
JCH201	CVE-2019-7310	Heap buffer overflow
JCH202	CVE-2018-21009	Integer overflow
JCH203	CVE-2018-20650	Type confusion
JCH204	CVE-2018-20481	0-pointer dereference
JCH206	CVE-2018-19058	Type confusion
JCH207	CVE-2018-13988	Out of bounds, incorect memory access
JCH208	CVE-2019-12360	Stack buffer overflow
JCH209	CVE-2018-10768	0-pointer dereference
JCH210	CVE-2017-9776	Integer overflow
JCH211	CVE-2017-18267	Stack overflow
JCH212	CVE-2017-14617	Floating point exception
JCH213	CVE-2017-7511	0-pointer dereference
JCH214	CVE-2019-12493	Stack overflow

2.2 Sqlite3

Here are a few reasons on why sqlite3 was chosen. First of all sqlite3 is a well known library. Furthermore sqlite3 has enough bugs to be implemented and finally sqlite3 was proposed because it has parsing code, recursion, it deals with strings and especially because sqlite3 is database management system.

One of the interesting thing with this target is that many bugs were found by fuzzers. In result, it was interesting to see if different fuzzers could find them.

Below is a list of bugs that could be implemented:

Bug ID	CVE	Type of bug
JCH214	CVE-2019-9936	Heap buffer overflow
JCH215	CVE-2019-20218	Stack unwinding
JCH216	CVE-2019-19923	Mishandles can cause 0-pointer dereference
JCH217	CVE-2019-19959	Error can lead to memory problems
JCH218	CVE-2019-19925	Null pointer
JCH219	CVE-2019-19244	Improper input validation
JCH220	CVE-2018-8740	0-pointer dereference
JCH221	CVE-2017-15286	0-pointer dereference
JCH222	CVE-2017-2520	Heap buffer overflow
JCH223	CVE-2017-2518	Use after free
JCH224	CVE-2017-2513	Use after free
JCH225	CVE-2017-10989	Heap buffer overflow
JCH226	CVE-2019-19646	Mishandles not null
JCH227	CVE-2013-7443	Buffer overflow
JCH228	CVE-2019-19926	Mishandles certain errors during parsing
JCH229	CVE-2019-19317	Omits bits
JCH230	CVE-2015-3415	Does not properly implement comparison operators
JCH231	CVE-2020-9327	0-pointer dereference
JCH232	CVE-2015-3414	Does not properly implement the dequoting of collation-sequence names
JCH233	CVE-2015-3416	Stack buffer overflow
JCH234	CVE-2019-19880	Invalid pointer dereference

3 WebPages

When we have the data of campaigns, we want to distribute them and have them available for different people to access. We also want different people to create their own graphs and analyze the data from the campaigns they ran. The best choice is to use webpages as they are quite simple to access from anywhere and everyone has experience with operating websites.

The webpages are generated by using data from a json file that is made by running campaigns. The site itself has three different types of pages. The report page, the fuzzer pages and the target pages. The report page is used to have an overview of all the different fuzzers and the different targets supported by the json file. Each of the target page represents a *target*. Finally, we have the fuzzer pages. Each fuzzer page represents a different fuzzer and each of them has dedicated graphs for the fuzzer they represent. Later on, we will present those graphs.

3.1 The Design of the Website

Let's now talk about the design of the website, both in programmatic and usage terms. The website is composed of three different parts. We start on a report page. On this page, we find links to the fuzzer and target pages.

The website is generated in three different phases. First, we create the different directories. Then we generate all the uesful plots and finally we create the fuzzer and target pages using the templates. Now, we will dive a little more in how the process works.

- The creation of the different directories. The different directories are: the outputs directory, the plot directory, the fuzzers directory and the targets directory. The outputs directory contains the

html pages, the other directories and all the graphs that will be stored after they are generated. Second, we have the plots directory. The plots directory contains all the graphs that will be generated in a later phase. Third, we have the targets directory, as the name itself says it will be the directory where each targets pages are stored and finally we have the fuzzers directory. The fuzzers directory stores every fuzzer pages.
To implement the creation of directories, we have created a class called `Path`.

```
class Path:
    def __init__(self, template_dir, output_dir,
                 tables_dir, plot_dir)
```

This class takes as arguments all the paths to the different directories, namely the outputs directory, the template directory where the templates for the fuzzer, target and report page are stored and finally the plots directory. This class also implements a function called `write`.

```
def write(self, output_file_name, rendering)
```

that writes to a file using the name of a template in the corresponding path.

- After the creation of the different directories, we have to create the different plots of the website. The different plots generated are: the bar plots, the box plots, the heatmap with p-values plot and the heatmap plot. We will explain later why each of those plots were chosen.
- When we have finished generating the directories and the different plots. We can then proceed to the creation of the report, fuzzer and target pages using the three different templates in our disposal. Let's now see how the different templates are used.
  - The Report Template: It is used to generate the report page of the website. It displays links for each fuzzers and targets that are part of the json file we give to the program as input. The report page also gives the number of bugs that have been injected in Magma for each targets.
  - The Library Template: It is used to generate the target pages. It contains a bar plot graph so that we can compare the number of bugs *reached* and *triggered* for each fuzzers. The library template also gives a library description for each library. It gives its name, wheter it is open source and a link to the repository of the library.
  - The Fuzzer Template: It is used to generate each of the fuzzer pages. It contains a box plots for each targets and for bugs that have been *reached* and *triggered*. The fuzzer template also implements a description for each fuzzer that is currently supported by Magma. The description gives the name, the type of the fuzzer, whether the fuzzer is open source and a link to the repository of the fuzzer itself.

In summary, we have the library template, the fuzzer template, and the report page template. The fuzzer template is used for each fuzzer. The library template is used for each *target*. The report page template is used to navigate between the pages and have access to the fuzzer and library pages.
Each template has a dedicated file called a `NAMETemplateGenerate` which takes a `Path` argument and where `NAME` is either a `fuzzer`, a `library` or a `mainPage`.

Each generator of templates implements an abstract class called `Render`:

```
class Render:
    def render(self, file_name, output_file_name):
        pass
```

The fuzzer pages are generated using the following class:

```
class FuzzerTemplate(Render)
```

With the `Path`, this class also takes as input the libraries so that it can find the different plots. Indeed, the plots have special names (e.g: afl_poppler_reached_boxplot.svg) so that the different pages that use them can find them more easily. Another solution would have been that each template that needs to use graphs search them in the `Plots` directory but this solution is overkill as it would lead to boilerplate and unnecessary code.

As for the target pages, they are generated using the following class:

```
class LibraryTemplate(Render)
```

And finally the report page is generated using the following class:

```
class MainPageTemplate(Render)
```

This class also takes as input the list of fuzzers and libraries it has to generate. The function `render` of the class `Render`.

3.2 Choice of graphs

Different graphs have been chosen in the website but as you will see different graphs have advantages and disadvantages. In consequence, it would make no sense to use only one graph because we want to have a full and complete analysis of benchmarks of fuzzers. Thus, we need different graphs: these graphs are the box plot graph, the bar plot graph and two different heatmap graphs:

- The Box plot:
  - Advantages: We can see how different bugs fare on different campaigns. We can see at a glance if bugs are difficult to be found or not by a fuzzer.
  - Disadvantages: It only touches the bugs but it has no direct comparison for different fuzzers.
- The Heatmap using p-values [1]: This plot is used for each fuzzers and targets. This graph is displayed on the report page.
  - Advantages: We can compare different fuzzers with each other, what performance each of them has and how each of them can find bugs and how efficient one is at finding them.
  - Disadvantages: This graph doesn't give any indication over how different bugs are *triggered*.
- The Heatmap plot: The heatmap plot shows the expected time to be *triggered* by a fuzzer for a bug of all targets. This plot is used on the report page as it shows all bugs for all fuzzers.
  - Advantages: It can show trends over time as we can directly see the expected time for a bug to be *triggered*. We can also compare different fuzzers and different bugs quite easily.
  - Disadvantages: The current heatmap can be quite hard to read and distinguish between different targets. We can't distinguish between different targets and we may be mislead into thinking that they are relations between bugs.
- The Bar plot: The bar plot is used in the target pages where we can make comparison between different fuzzers. Each bar plot shows the number of bugs that was *reached* and *triggered* for each fuzzers.
  - Advantages: The bar plot shows how a number of bugs fare between different fuzzers. This plot summarizes data for all fuzzers that have found bugs for a target. We can make comparisons between *reached* and *triggered* and this accross all fuzzers for a target.
  - Disadvantages: We don't have any clear idea how different bugs fare for different fuzzers. For example, we could have one fuzzer that finds many bugs but all of them late in campaigns.

[1] p-values from the Mann-Whitney U-Test

4 Discussion

Now that we have introduced all the graphs, we can start our analysis. But what analysis critera should we use ? Below we are going to talk about different criteria, their advantages and disadvantages.

- Average number of bugs found per campaign.
  - Advantages: We can make simple comparisons over different campaigns and different fuzzers.
  - Disadvantages: We don't distinguish bugs and this metric doesn't say anything about the bugs themselves.
- Total number of unique bugs found.
  - Advantages: We can directly say who is better. Obviously the fuzzer that finds the most unique bugs would be the best.
  - Disadvantages: This doesn't tell us when different bugs are *reached* or *triggered*. Some bugs could even be found only once in hundreds of campaigns and still be counted.
- The time when a unique bug is found.
  - Advantages: If a bug is found fast then it is much better because the sooner we are aware of a bug, the sooner we can resolve the bug.
  - Disadvantages: This doesn't give any information about the number of bugs a fuzzer finds.
- The ratio of *triggered* bugs by *reached* bugs. This can gives us an idea of the number of *reached* and *triggered* bugs a fuzzer can find.
  - Advantages: This can give us a good idea on which fuzzer could be the best.
  - Disadvantages: This could also give us a wrong idea because many bugs could be found later on in campaigns and thus making them difficult to reach.
- The time it takes to be triggered after it was reached once. Then, we could have the possiblity of testing how the coverage change between reached and triggered for each bugs.

Let's now make some observations about the different targets. Notice, we can't make comparison between different targets as those have ran campaigns that are totally independent of each other.

4.1 Libpng

If we look at the bar plot we can directly compare all fuzzers for the target *libpng* as seen in Fig. 1a.

We see that all fuzzers can reach six bugs. Thus, in this comparison all of them are equivalent. But we also see that *Honggfuzz* is the best one at triggering unique bugs because it triggers four bugs. But this doesn't necessarly mean that *Honggfuzz* is the best fuzzer because the bar plot doesn't say anything about when the fuzzer can find bugs as it is one of the disadvantages of the bar plot. In result, we have to look more closely at the box plot graphs for each fuzzer. If we take the box plot graph for *Honggfuzz* as seen in Fig. 1b, we can then see that the bug *AAH001* takes about 50,000 seconds (or about 13 hours and 53 minutes) to be *triggered*. This means that *AAH001* is a difficult bug for *HongFuzz*. But the other bugs (*AAH008*,*AAH007*, *AAH003*) can be found quite fast by *Honggfuzz* compared to the other fuzzers.

4.2 Libtiff

As before, we can make comparisons using the bar plot between all fuzzers and see that *FairFuzz* is the best fuzzer at triggering unique bugs and also finds the most bugs as seen in Fig. 4a.

As before, we need to make a more thorough comparison and look more closely at the different bugs and how they fare against each other. If we look at the box plot of *FairFuzz* as seen in Fig. 4b. We see

that the bug *AAH022*, *AAH010* and *AAH014* did take a while to be found in different campaigns. But *AFL++* and *AFLFast* find more unique bugs and faster than *FairFuzz* does. In this respect, *AFL++* is better than *FairFuzz*. If we take *AFLFast* as seen in Fig. 5b. We have only two bugs before the 10,000 mark. Futhermore, *AAH020* is very close to this mark. We also see (as seen in Fig. 6c) that *AFL* can find three bugs below the 10,000 mark. *HongFuzz* finds about 4 bugs before the 10,000 mark.

In conclusion, we can say that *HongFuzz* finds more bugs and faster than other fuzzers. Even though, *FairFuzz* finds more triggered bugs.

4.3 Libxml2

If we look at the bar plot, we see that *HongFuzz* is the clear winner with the number of bugs *reached* and *triggered* bugs as seen in Fig. 7a.

HongFuzz finds the bugs (*AAH032*, *AAH037* and *AAH041*) below the 2,000 mark as seen in Fig. 7b. We also see that *Mopt-AFL* (as seen in Fig. 8a) and *AFLFast* (as seen in Fig. 8b) can find all their three bugs below and around the 2,000 mark. We have something very similar with *AFL++* (as seen in Fig. 9a) and *AFL* (as seen in Fig. 9b).

In conclusion, *HongFuzz* is the best fuzzer in this case. Yes, all fuzzers finds bugs below and around the 2,000 marks which makes them all equal in terms of the time when they find bugs but *HongFuzz* finds more bugs and thus making it the best fuzzer for *libxml2*.

4.4 Openssl

We can see in the bar plot (as seen in Fig. 10a) that *AFL* (as seen in Fig. 12b), *AFLFast* (as seen in Fig. 11b), *AFL++* (as seen in Fig. 12a) and *HongFuzz* (as seen in Fig. 10b) are quite similar because they trigger four bugs. We also see the same thing with *Mopt-AFL* (as seen in Fig. 11a) and *FairFuzz* (as seen in Fig. 12c) where both of them triggers three bugs.

For *AFL* and *AFLFast*, three bugs can be reached way faster than the last one but for two bugs it takes a while to be *triggered*. We have the same kind of behavior for *AFL++*. All fuzzers are quite close to each other. Each of them finds at least three bugs below and around the 1,000 seconds mark. The two exceptions are *Mopt-AFL* and *HongFuzz*. *Mopt-AFL* finds one bug (*MAE115*) late in the campaigns. But *HongFuzz* finds two bugs late in the campaigns (*AAH053* and *MAE115*).

4.5 Php

As we can see in the bar plot, all fuzzers are quite equal in terms of ratio *triggered* by *reached*. As they all finds the same number of *triggered reached* bugs. But if we look more closely at the box plots, we directly see a different story in terms of time to be *triggered*.

Indeed, we already can rule that *FairFuzz* is clearly not the best fuzzer in this target. When looking more closely at the graphs, we can see a similar story for *HongFuzz*. But in this case, two bugs are found later than *FairFuzz*. Thus, we can also rule out *HongFuzz*.

We also see that *Mopt-AFL*, *AFL++*, *AFL* and *AFLFast* have striking similarity. This is not necessarily as surprising as it may seem as all of those fuzzers work almost identically [4], [5], [6], [7]. This is because all of those fuzzers are extensions of *AFL*. Only the method to generate new inputs may differ and be different over all those fuzzers. Thus, it is not really surprising that after about 8 minutes all of them would have striking resemblance. After only a few minutes, they may all still be in the same iteration. Even though fuzzers work with random input, Magma makes sure that all of them start with the same seed for fairness reasons.

FairFuzz doesn't have the same trend as the *AFL* family based fuzzers, even tough *FairFuzz* is a member of the *AFL* family. The difference lies in the fact that *FairFuzz* is not an extension of *AFL* but a

modification of *AFL* [8].

In conclusion, we can only say that all fuzzers from the *AFL* family give similar results even though more research and tests would be needed for this particular *target* as all of them use the same seed. We can also say that in the first few minutes all of them have similar behavior as not many iterations would have passed. To confirm this we would need different types of graphs, such as a coverage graph or even an iteration graph so that we can see exactly when all of them start to have different behaviors.

4.6 Poppler

For *HongFuzz* (as seen in Fig. 16b), we can see that three bugs can be found before the 10,000 mark. Every other fuzzer can only find two bugs before the 10,000 mark and those two bugs are the exact same as two of the three bugs that *HongFuzz* finds. Furthermore, *HongFuzz* finds one bug (*AAH048*) around the 40,000 mark and the bug isn't found for any other fuzzers. Though *HongFuzz* only finds it once. This may only be due to chance and *HongFuzz* may not find it anymore in other campaigns. Thus, we would need to run more benchmarks for that fuzzer and see if that particular bug still can be found or if it was only pure chance.
One bug (*JCH201*) can be found by all fuzzers except *HongFuzz*. That bug is found by all fuzzers after the 40,000 mark which tells us that this particular bug is difficult to find by all fuzzers and especially those from the *AFL* family. This bug could be difficult to find for any type of fuzzer. Thus, new benchmarks would be needed to see if other types of fuzzers can find it faster or if that bug is difficult to be found by any type of fuzzer.

Similar to *PHP*, all the fuzzers from the *AFL* family have similar behaviors with the bug *JCH207*. We can make the same conclusion as with *PHP*.

In conclusion, *HongFuzz* finds more bugs and faster than other fuzzers for this *target*. Even though two bugs that *HongFuzz* finds was also found in most fuzzers around the same time mark.

4.7 Sqlite3

HongFuzz has more *triggered* bugs but does not have the best ratio of *triggered* by *reached* bugs as seen in Fig. 19a. *HongFuzz* can find three bugs before the 20,000 mark (as seen in Fig. 19b). But other fuzzers don't except *Mopt-AFL* which can find all three bugs before and during the 15,000 mark (as seen in Fig. 20a).

Mopt-AFL finds all its three bugs between the 10,000 and 15,000 mark. As opposed to *HongFuzz* which finds two bugs before the 5,000 mark. The last one is found around the 20,000 mark which is barely 8 minutes after the last bug that *Mopt-AFL* finds.

In conclusion, both *Mopt-AFL* and *HongFuzz* are quite close to each other. *HongFuzz* finds more *triggered* bugs and it also finds more bugs than *Mopt-AFL* before the 5,000 mark. Though if we take the comparison with the 15,000 mark, then *Mopt-AFL* clearly wins.

5 Conclusion

We have talked about Magma, what it is and how we can add new bugs. We have also talked about the different fuzzers and targets that was incorporated in Magma. After this we went on talking about the different new bugs that have been implemented under the guidance of that report.

Magma has now implemented webpages and we can make comparison between different fuzzers and different targets such that anyone running campaigns can launch that website. It has now the following graphs: the bar plot, the heatmaps plots and the box plots. It is essential to have different graphs and compare different graphs because different graphs give different perspectives. We can also make

comparisons on how different fuzzers or even different bugs fare against each other.

As we have seen, chosing the best fuzzer usually depends on the case. As with everything in Computer Science there is not a fuzzer that is the best in every occasion. It depends on what we want and can do. For most of our targets the best fuzzer was *HongFuzz*. Even though, many bugs could still be found by different fuzzers. In the industry, we have to choose the best fuzzer and find bugs as fast as possible so that no security risks can be exploited by ill-intentioned people. Even if *HongFuzz* has better performance over the different targets than all the other fuzzers. Different fuzzers can find other bugs that *HongFuzz* itself can't find. On other occasions, different fuzzers find bugs faster than *HongFuzz*.

Magma had the goal of adding more targets and bugs. It has reached that goal but it still has to do more of them. We have found earlier that *HongFuzz* was the best fuzzer in many occasions but does that trend continues with different programming techniques, with different targets.
One of the thing Magma has to do is to add more diverse targets and more diverse techniques of programming. For now, it has an interpreter with the *PHP target* but how would fuzzers work with different interpreters or even with different compilers as both interpreters and compilers are quite similar in how they work. Does the same fuzzer keeps getting consistent results if there is different ways of programming for lexers, parsers, loaders, assemblers ?
Even how would fuzzers fare on machine learning targets (e.g *Tensor Flow*) or even other *Artificial Intelligence* targets.
For now, all targets belong to either the *C* or *C++* programming language but what would happen with different languages such as *Java*, *Scala* or *Python*. At first we could think that since we have very different languages., different fuzzers would give different results. Could we see a trend on a family of programming languages ?

There is also the possibilty of adding more types of graphs. Such as graphs that displays the coverage of different fuzzers and how they can find different bugs. Does their coverage actually improve their technique of bug findings ?

For now, Magma only implements mutational grey-box fuzzer but it should increase its number and have different types of fuzzers and compare different fuzzers with each other. There could also be targeted benchmarks for each fuzzers and see how the claims the fuzzer usually makes about a technique can actually find new bugs.

6 References

[1] S. McConnell, *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2004.

[2] HexHive Laboratory, "Magma." `https://github.com/HexHive/magma`, Last accessed on June 14, 2020.

[3] M. Payer and A. Hazimeh, "Magma: A ground-truth fuzzing benchmark,"

[4] "Mopt-AFL." `https://github.com/puppet-meteor/MOpt-AFL`, Last accessed on June 14, 2020.

[5] "AFL++." `https://github.com/AFLplusplus/AFLplusplus`, Last accessed on June 14, 2020.

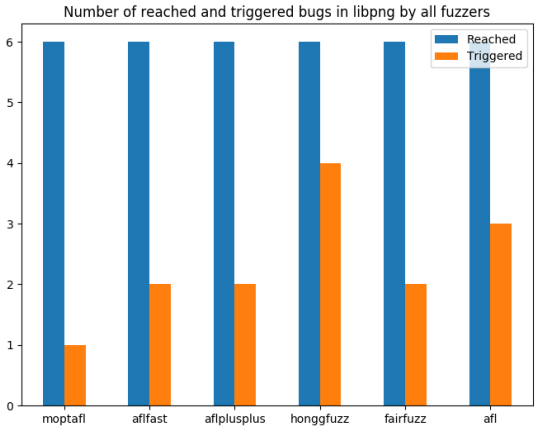
[6] Michał Zalewski, "American fuzzy loop," 2019, stable release (2.56b). `https://lcamtuf.coredump.cx/afl/`, Last accessed on June 14, 2020.

[7] "AFLFast." `https://github.com/mboehme/aflfast`, Last accessed on June 14, 2020.

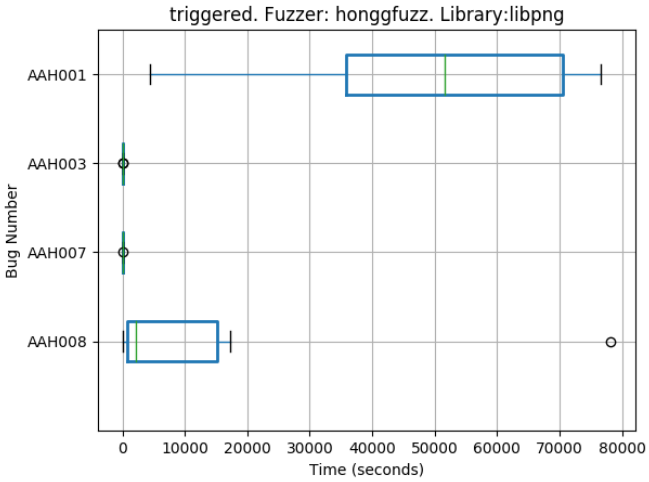
[8] "FairFuzz." `https://github.com/carolemieux/afl-rb`, Last accessed on June 14, 2020.

Appendix A Graphs

A.1 Libpng

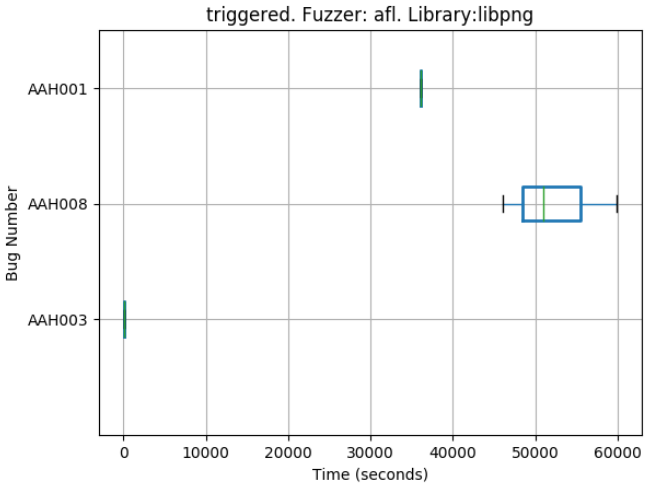


(a)

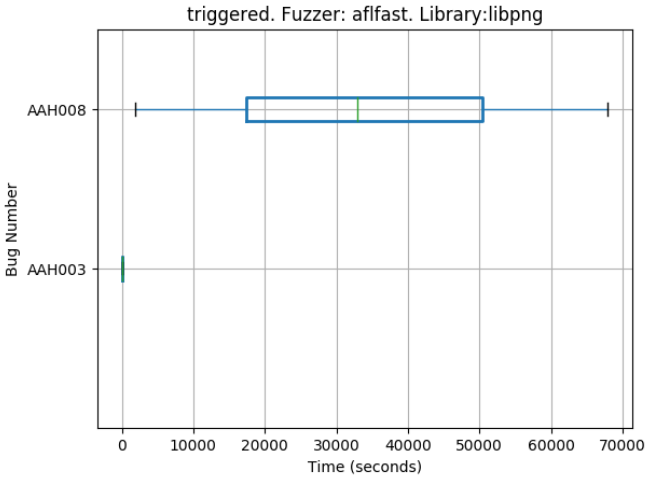


(b)

Figure 1

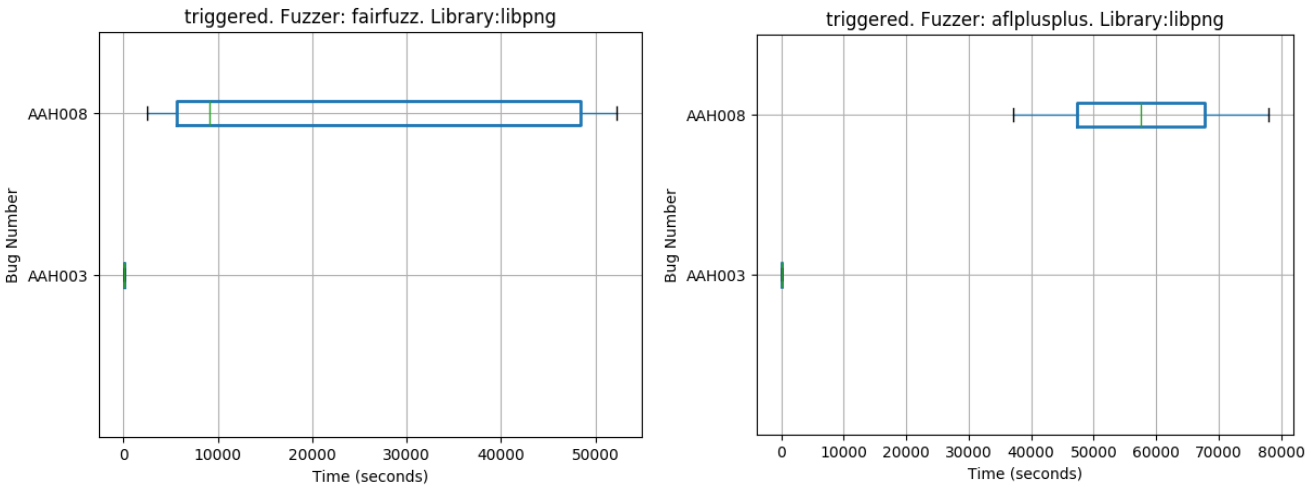


(a)



(b)

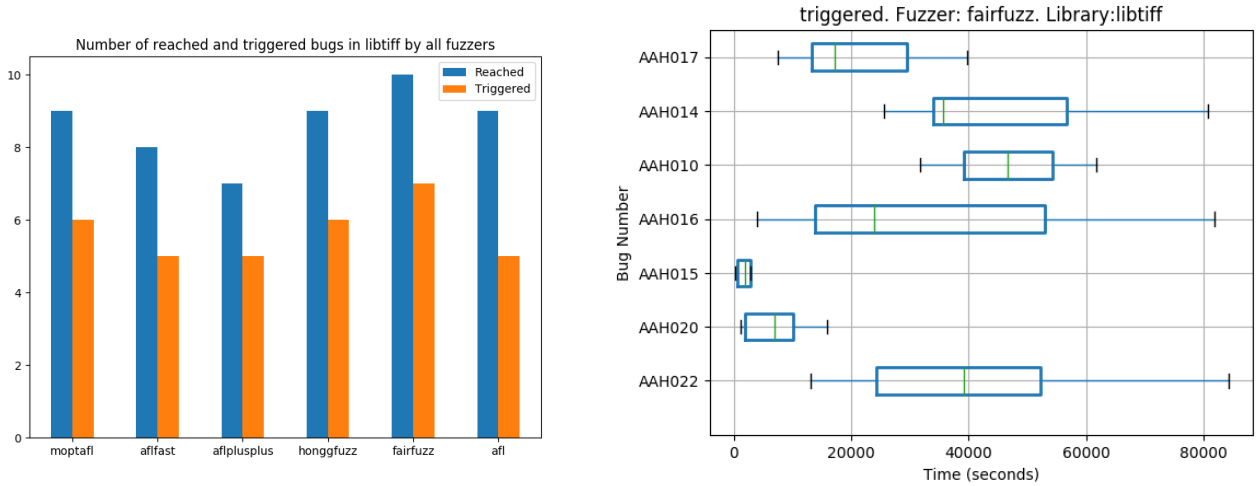
Figure 2



(a) (b)

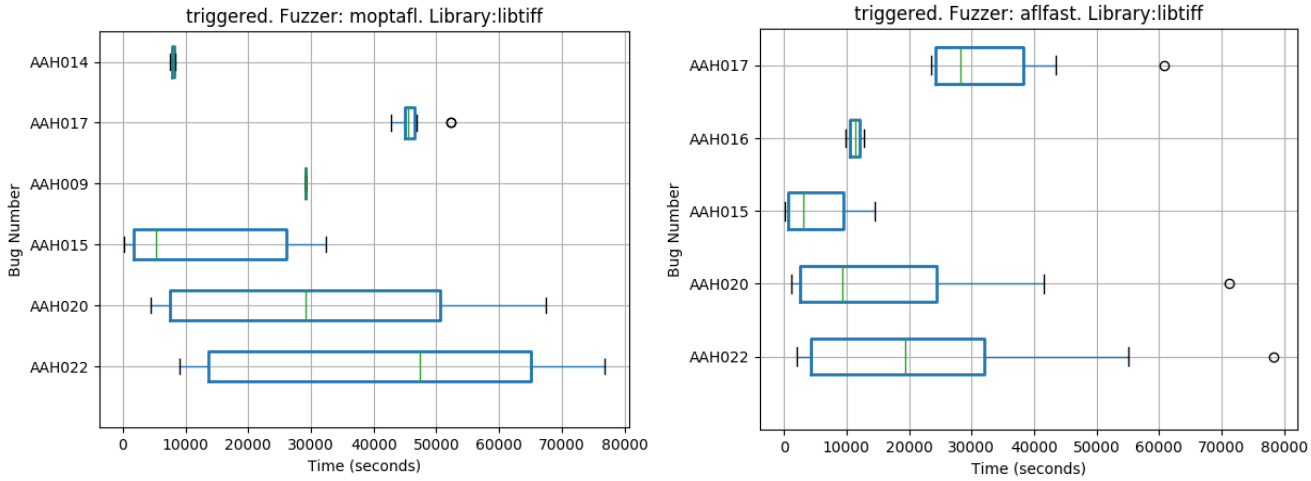
Figure 3

A.2 Libtiff



(a) (b)

Figure 4



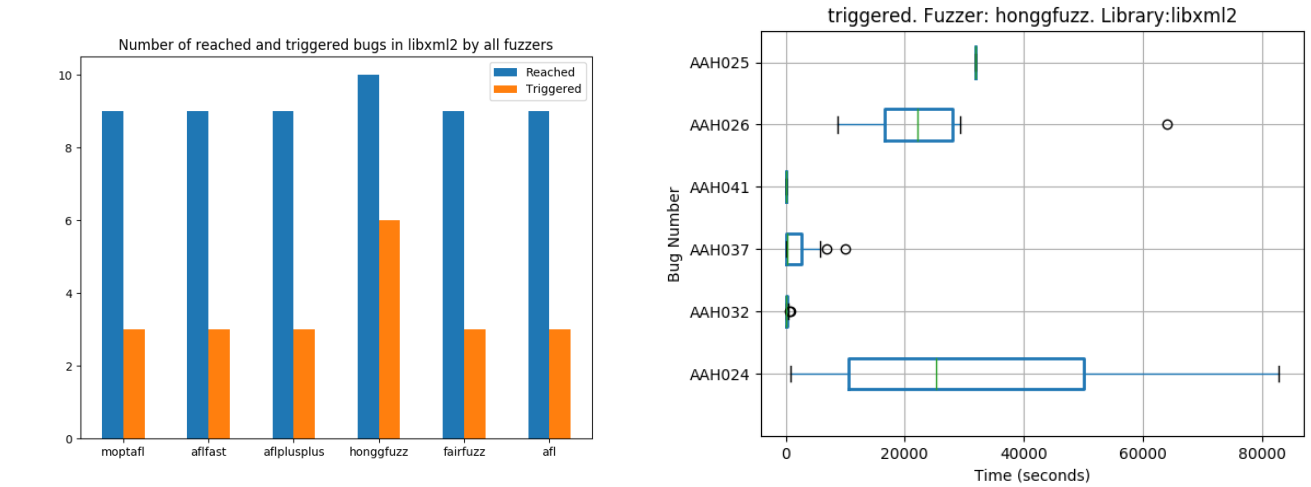
(a) (b)

Figure 5

(a) (b) (c)

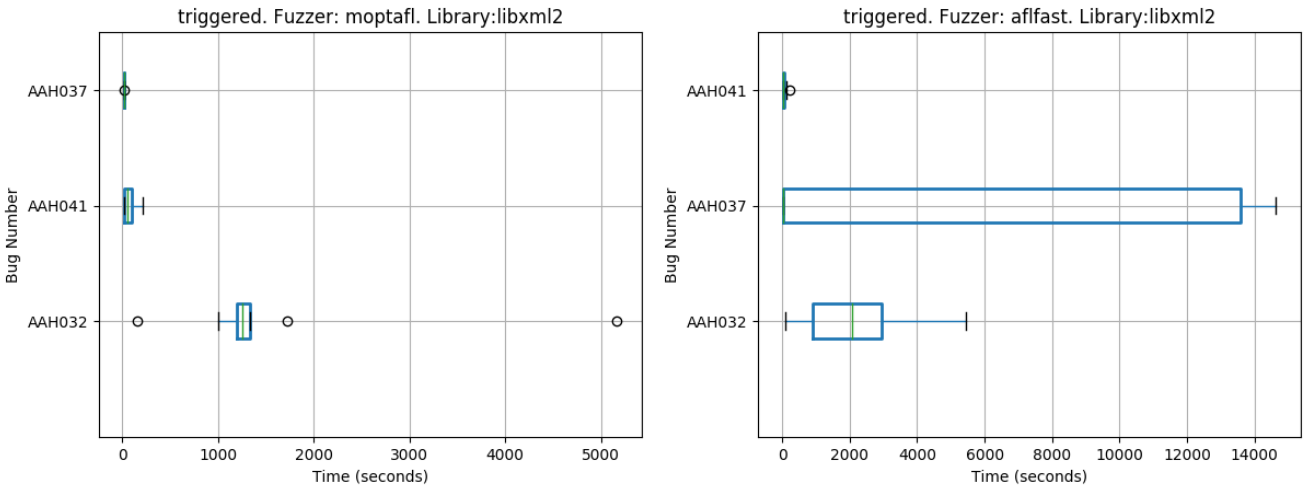
Figure 6

A.3 Libxml2



(a) (b)

Figure 7



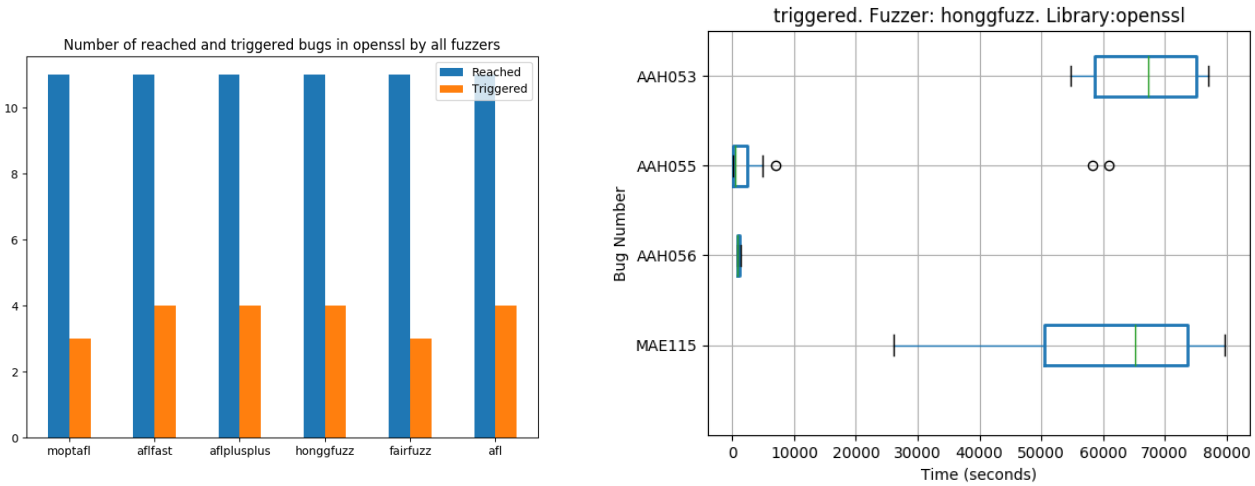
(a) (b)

Figure 8

(a) (b) (c)

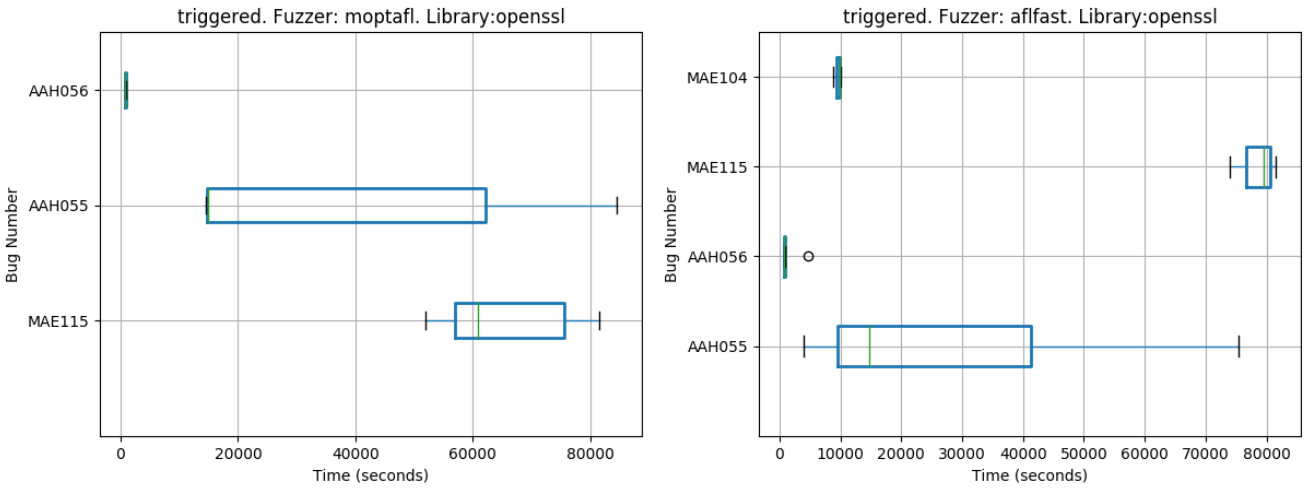
Figure 9

A.4 Openssl



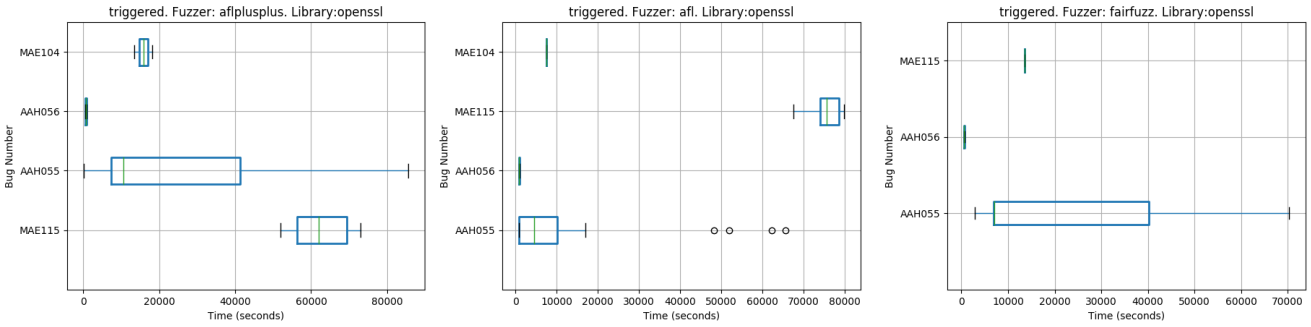
(a) (b)

Figure 10



(a) (b)

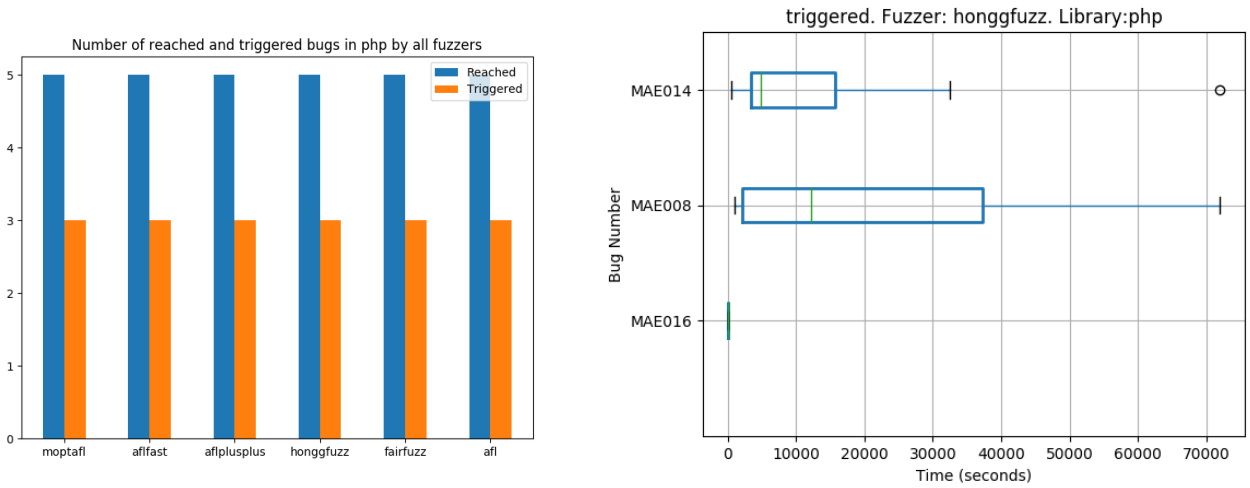
Figure 11



(a) (b) (c)

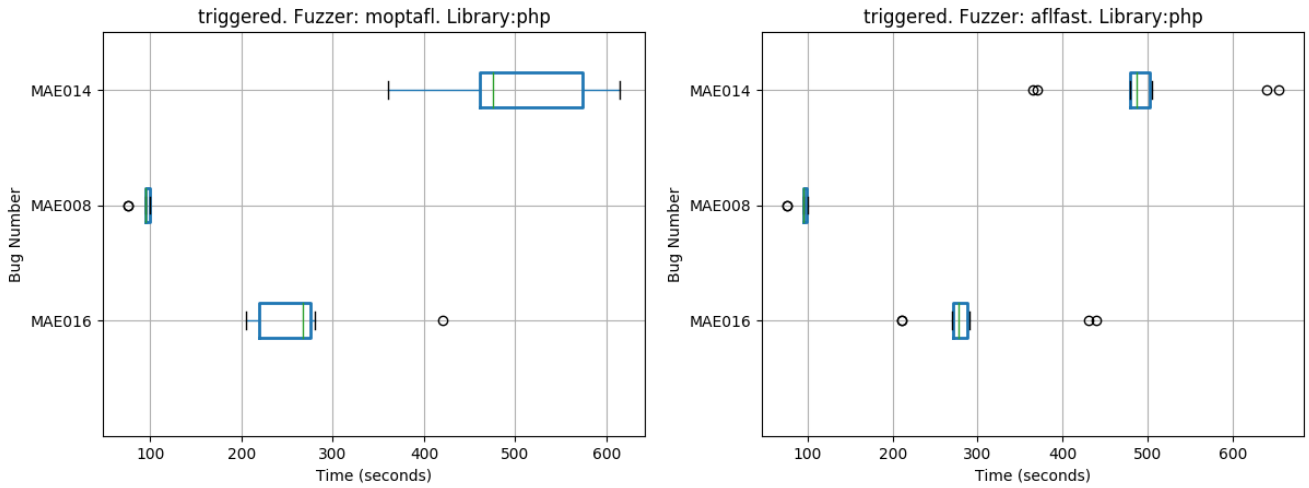
Figure 12

A.5 Php



(a) (b)

Figure 13



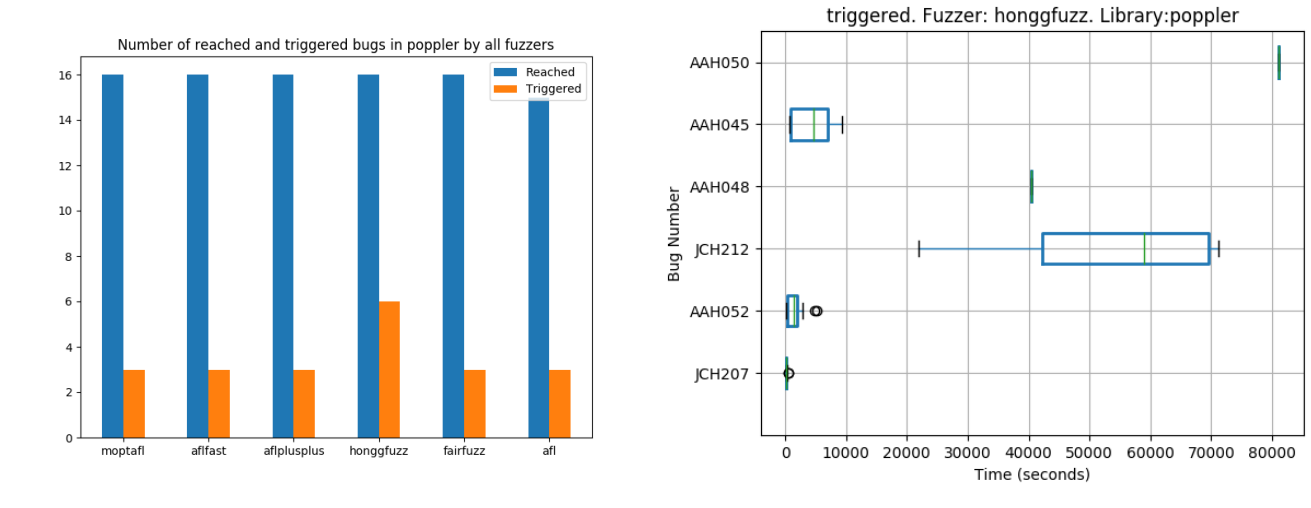
(a) (b)

Figure 14

(a) (b) (c)

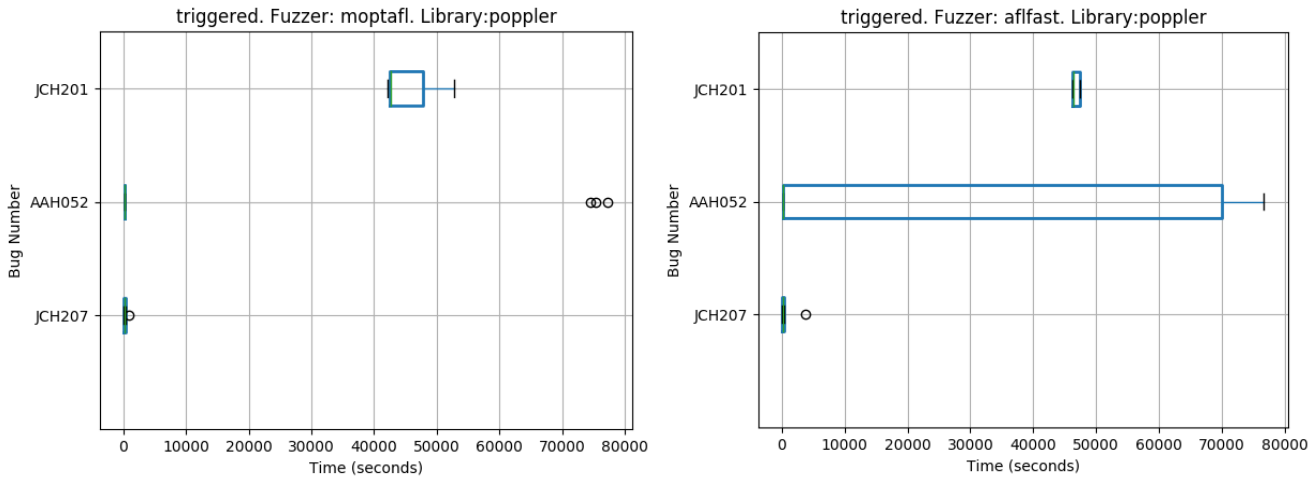
Figure 15

A.6 Poppler



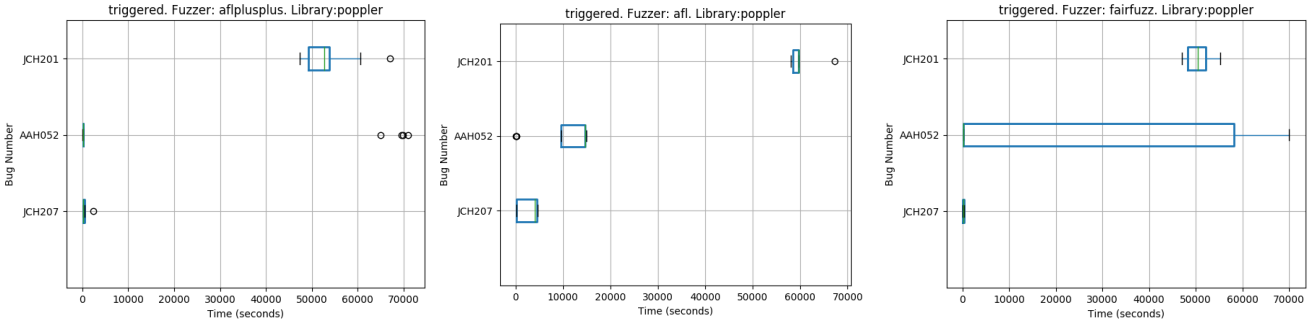
(a) (b)

Figure 16



(a) (b)

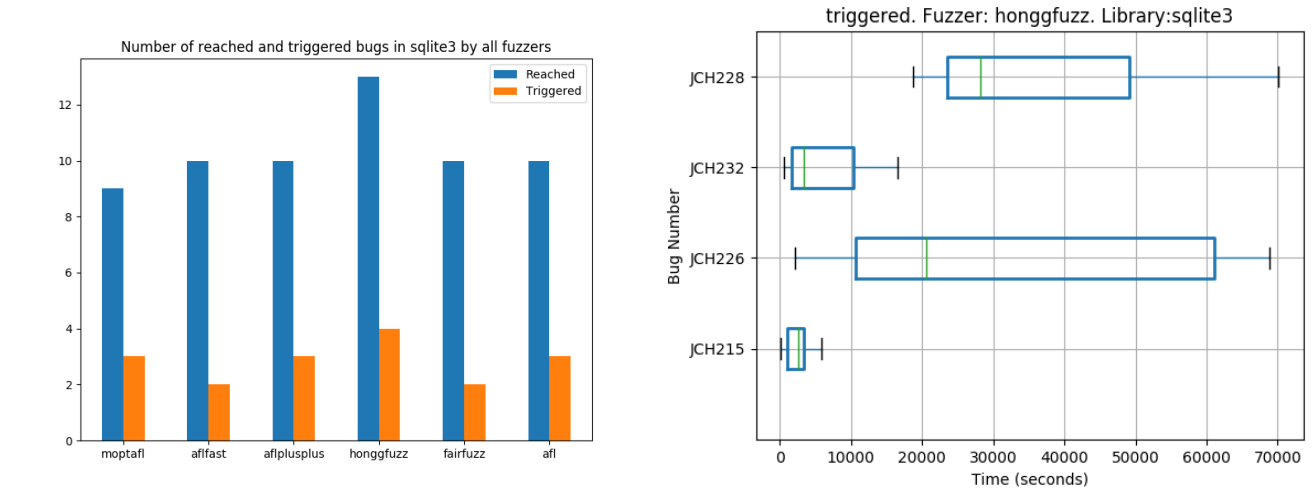
Figure 17



(a) (b) (c)

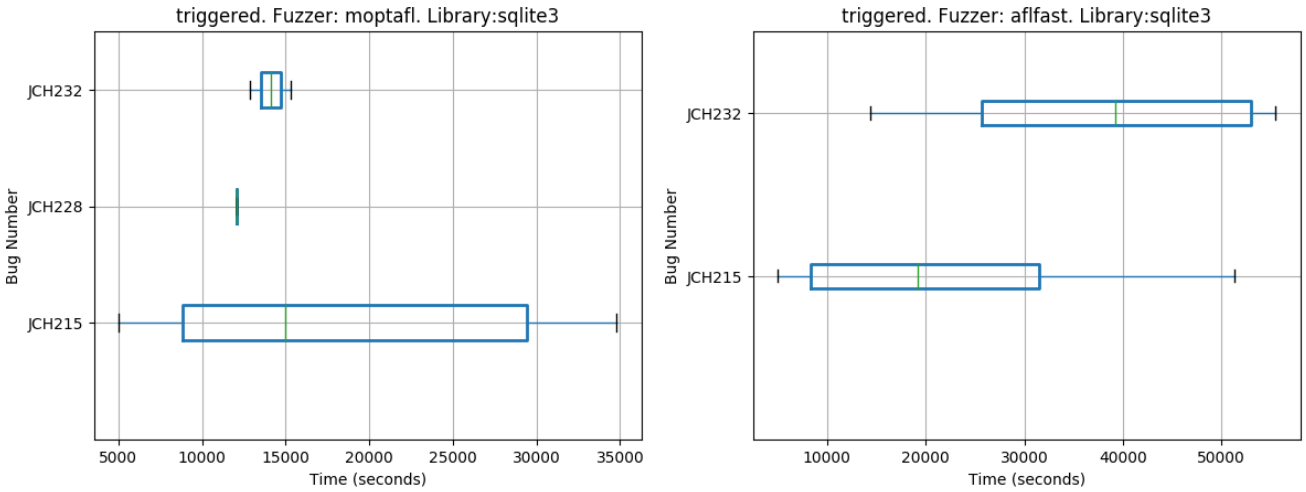
Figure 18

A.7 Sqlite3



(a) (b)

Figure 19



(a) (b)

Figure 20

(a) (b) (c)

Figure 21