AUSTRALIAN NATIONAL UNIVERSITY

COMP4560 ADVANCED COMPUTING PROJECT

Improving fuzzing performance using hardware-accelerated hashing and PCA guidance

Author:
Jorge Pinilla López
(u6759601)

Supervisors:
Alwen Tiu
Charles Gretton
Adrian Herrera and
Hendra Gunadi

May 31, 2019

AUSTRALIAN NATIONAL UNIVERSITY

Abstract

College of Engineering and Computer Sciences

Bachelor of Advanced Computing

Improving fuzzing performance using hardware-accelerated hashing and PCA guidance

by Jorge Pinilla López
(u6759601)

Fuzzing is growing software testing technique that is especially successful when applied to large packages of software as it is able to automatically find bugs. One of the most used fuzzers is American Fuzzy Lop (AFL). While being fast and simple, AFL suffers from a poor guidance do to it's simple approach to seed rating and a collision problem when tracing large programs. Some previous work demonstrate the advantages of using Principal Components Analysis (PCA) as a guidance. In this work, we extend the previous implementation by fully integrating PCA in AFL, improving and extending the previous implementation. We extend the previous testing by exploring the effectiveness of synchronized PCA guided fuzzers. In addition to this, we aim to reduce the collision rate by providing a new hardware-accelerated hashing function based on AES. The results shows that our PCA implementation outperforms the vanilla AFL and previous implementation. Although we tested our AES implementation in theoretical high collision targets, we are not able to validate the effectiveness of our implementation due to the lack of real collisions. However we demonstrated that the performance impact of the implementation is negligible.

Acknowledgements

I would like to express my gratitude to my supervisors Alwen, Charles, Adrian and Hendra, as well as the Computer Science department for providing this opportunity to do this research project, supporting and encouraging me to learn further and test new ideas. I also want to express my gratitude to my parents Lourdes and Felix, for giving me the opportunity to study so far from home in such a prestigious university. Finally I want to thanks the Institute of Bio-computation and Physics of Complex Systems for letting me use their high performance computer to run part of my experiments.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Fuzzing is a state-of-the-art technique for finding bugs in software, especially for large projects. Fuzzing consists of the automatic, or semi-automatic generation of program inputs in order to find unexpected behaviours in software. While there are multiple styles of fuzzers, the most widely used are *greybox fuzzers*. Greybox fuzzers use indirect analysis or lightweight indicators to perform semi-guided input generation, offering a balance between speed and accurate generation of inputs. One of the most used greybox fuzzers is American Fuzzy Lop (Zalewski 2016a).

We based our research on previous work by Martin 2018, who showed the potential of adding machine learning techniques to guide the fuzzing process. Specifically, this work focused on using Principal Components Analysis (PCA) and resulted in a tool called PCA^{2}FL. We aim to extend this work and implement a second version of PCA^{2}FL in C/C++ and fully integrated with AFL. Our new version of PCA^{2}FL fixes a number of bugs in the previous implementation and improves performance by adding the ability to relearn the PCA operator during a fuzzing campaign. Our implementation eliminates all on Python language and libraries. We test this new implementation of the fuzzer on the LAVA-M benchmark and we explore the impact of horizontal scaling on PCA performance by running tests with several fuzzers in synchronised mode. The results for synchronized mode outperform the results the previous PCA^{2}FL implementation by Martin 2018 and in synchronized mode, is able to find the same or more bugs than the vanilla AFL implementation.

A second part of this paper attempts to fix the problem in AFL's hashing algorithm. AFL uses a simple but fast hashing algorithm to compress information about execution traces. It is well known that this hashing algorithm is likely to have collisions by its design specially on high density targets for example libtorrent, vim or libav. We aimed to solve this problem by implementing a hardware-accelerated AES hashing algorithm that, while maintaining a similar number of executions per second, allows a better dispersion function that reduces the collision number. We evaluated this algorithm and show that the execution speed is not considerably affected. We tested our implementation on high collision targets reported by Gan et al. 2018. Unfortunately, our experiments could not replicate their results even after extensive testing and therefore we could not demonstrate an improvement in the number of collision due to the lack of high collision targets in the testing benchmarks. However we are able to demonstrate the our AES hashing implementation has a minimal impact on performance compared to the vanilla AFL hashing.

Chapter 2

Background and Related Work

2.1 Fuzzing

Fuzzing, as defined by Sutton, Greene, and Amini 2007, is an automatic or semi-automatic method for discovering faults in software by providing unexpected inputs. The main goal of fuzzing is the automatic discovery of unexpected behaviour of the program due to unexpected inputs. The technique consists of repeatedly generating inputs by mutating a set of initial input *seeds*, running the target with the mutated inputs, and analyzing the result as well as any possible crashes that occur.

There are three types of fuzzers as described by Sutton, Greene, and Amini 2007: *whitebox* fuzzers perform a static code analysis to perform a very accurate but slow input generation constrain by the code and possible execution paths. These fuzzers are highly effective as they generate accurate inputs but are mostly infeasible to execute due to the high computational requirements of static code analysis specially on large sources. *Blackbox* fuzzers perform random input mutations without any guidance of the source code or program execution behaviour. These are the fastest fuzzers generating inputs but are also less guided and therefore less accurate. *Greybox* fuzzers use lightweight code analysis or indirect program status and execute analysis to perform a guided generation of inputs. These fuzzers are the most common as they give a tradeoff between speed and input generation.

2.2 America Fuzzy Lop

American Fuzzy Lop (AFL)(Zalewski 2016a) is a greybox fuzzer designed by Michał Zalewski and written in C/Assembly. AFL uses compile-time instrumentation and genetic algorithms to automatically discover interesting test cases. It focus on C/C++ programs compiled with gcc or Clang, however multiple variants (e.g., go-fuzz, python-afl, etc.) extend the use of AFL for other programming languages.

2.2.1 Workflow

AFL starts by instrumenting the target binary. Once the target is instrumented it performs a *dry run* on the initial seeds, allowing AFL to collect some statistics about these base seeds. After that, AFL selects a single input to be fuzzed and performs mutations on this input and tests of the results. If any of the results seem "interesting"—e.g., they discover new program behaviours—the input is added to the queue and the process is started again by selecting the next input from the queue. (See Figure 2.1)

2.2.2 The AFL bitmap and interesting seeds

We now discuss how and what statistics AFL collects when running an instrumented binary on a seed. The bitmap is a shared memory space between the target binary and the fuzzer (see Zalewski 2016b). The bitmap allows AFL to collect information about the target execution. By default, the bitmap has a size of 64 Kilobytes. This size was chosen in order to fit within the L2 cache and avoid overheads when running the target. The bitmap contains 2^{16} 8-bit unsigned integers. Each of these integers represents the number of executions of a certain edge (between two basic blocks) in the target program. This allows the fuzzer to collect statistics on what paths of the program have been explored, without the large overhead of performing static code analysis. One of the problems that AFL faces is the possibility of hash collisions due to the bitmap size. Some researchers have experimented with changing the bitmap size, however the results showed a great slowdown on the target execution. For example, in Gan et al. 2018, the authors found that increasing the bitmap size from 64Kb to 1Mb slowed down executions up to 30% in targets like libtorrent.

AFL uses the bitmap to decide whether a seed is interesting based on the target's execution behaviour. The bitmap contains the information of the number of executions of each tuple, a tuple represents the transition between two *basic blocks* of the target. First, AFL checks if new tuples have been executed. If so, the seed is considered interesting. In addition to new paths, AFL considers a coarse tuple hit counts by splitting the path hits into several buckets: 1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+ hits. AFL considers the transition from one bucket to the other interesting. This makes the algorithm more robust to small changes and reduces the impact of some collisions.

2.2.3 Instrumentation

In order to guide input generation, AFL tracks the program execution by injecting instructions into the target program during compilation. AFL injects this instrumentation using a wrapper around the GCC or Clang compilers. AFL's instrumentation first maps the shared memory bitmap into the program memory. It then assigns a 16-bit random identifier to each *basic block*. This identifier, together with the identifier of the previously-executed basic block, then used to generate a hash representing the edge transition between these two blocks. The hashing function used by AFL is a simple but very fast function (see Algorithm 2.1). It calculates a hash by performing an xor between the current block identifier and the previous block identifier right shifted by one and then. The result is used as an index into the bitmap, where the count at that location is incremented by one.

Algorithm 2.1 Pseudo algorithm of AFL Vanilla Hashing function

```
1: procedure HASHING
2:     shared_mem[cur_location ⊕ prev_location]++;
3:     prev_location = cur_location >> 1;
```

2.2.4 Mutation

AFL mutates the input under three stages as explain in Lemieux and Sen 2017: the *deterministic stage*, the *havoc stage*, and the *splicing stage* (see Figure 2.2). Each step performs a series of mutation steps described below and explain in Zalewski 2014.

The deterministic stage works by performing mutations at sequential positions. These mutations include:

1. Flipping specific bits
2. Flipping specific bytes
3. Simple arithmetic (e.g., addition or subtraction of integers)
4. Replacing values with "interesting" values (e.g., -1 or `MAX_INT`)

After performing these deterministic mutations, the fuzzer enters into a havoc phase where it tries to perform random mutations of the input. The duration of the havoc phase is dictated by the score of the input and the results of the mutations. The havoc phase consists of:

1. Single-bit flips
2. Set interesting bytes, words, dwords, etc.
3. Simple arithmetic operations
4. Random bytes set
5. Block deletion
6. Block duplication by overwrite or insert
7. Block memset

As a last resort, AFL takes two distinct input files from the queue, and combines both by splicing them at random locations.

2.2.5 Score

The score is a metric used by AFL to determine the duration of the havoc phase. AFL calculates a scored based on execution speed, bitmap size, depth of the paths and how late the entry was found. All of these parameters are hard coded in AFL and produce a simple score. The full scoring algorithm is shown in Appendix 5. Entries with higher scores are allowed to be fuzzed longer during the havoc phase as they are considered to be more interesting.

The bitmap represents a full execution of an input. Therefore, being able to compare full bitmaps would give a much more detail information about the execution of the inputs and how interesting they are. However, comparing all bitmaps is slow and inefficient. Therefore, vanilla AFL only compares basic bitmap parameters like bitmap size or path depth.

2.3 Principal Component Analysis

Martin 2018's previous work demonstrated the ability to reduce the bitmap components to 2 dimensions by using PCA while maintaining the ability to visually classify the inputs that resulted in crashes. This indicates that a reduced 2 dimensional representation of bitmaps holds enough information to guide a fuzzer while still being efficient to compare.

Principal Component Analysis (PCA) is a statistical procedure used to reduce dimensions of dataset. It uses orthogonal matrix transformations to produced a reduced linear combination of variables that explains the most variance of the original dataset. PCA is achieved by performing eigen decomposition of the covariance matrix of the normalized data. (See Algorithm 2.2)

Algorithm 2.2 Algorithm of the PCA calculation and reduction

```
1: procedure PCA REDUCTION
2:     Normalized_data = Data - Average_Data
3:     Covariance_matrix = transpose(Normalized_data) * Normalized_data
4:     EigenValues, EigenVectors = eigen_decomposition(Covariance_matrix)
5:     Reduced_EigenVectors = Select k vectors with the highest EigenValues
6:     ReducedData = Normalized_data * Reduced_EigenVectors
```

2.4 AES

AES algorithm is a symetric block cipher algorithm for data encryption. It was created in 2001 and was specifically designed for cyber security in USA . AES algorithm tries to achieve 2 principles, confusion and diffusion. Confusion means that each bit of the output should depend on several parts of the key, obscuring the connections between the two, this is perform in AES by a key addition layer. Diffusion means that changing a single bit of the input, will change half of the bits in the output. This is performed in the AES by doing 3 operations: substitute bytes, shift rows and mix columns. The AES algorithm starts representing 128 input bits as a 4x4 matrix as described in Figure 2.3. Then 10 rounds are perform, a part of the key is added at the beginnin of the round. Then AES performs the diffusion layers containing byte substitution, row shifting and column mixing. Finally the next part of the key is added again. The last round of AES does not contain the mix columns transformation The algorithm is represented in Figure 2.4.

The wide use of this algorithm has lead to hardware efficient implementations of it. Intel offers an inbuilt hardware accelerated implementation of the AES algorithm in most modern CPUs (Gael 2012).

2.5 LAVA: Large-scale Automated Vulnerability Addition

Fuzzers rely on semi-random algorithms and comparing fuzzers can be really hard, especially on real applications as explain in Klees et al. 2018. One of the major problems in comparing fuzzers is finding the cause of a bug as multiple inputs can produce the same crash, the process of finding the cause of bugs is called Triage. In order to solve both of this issues some artificial datasets have been created providing artificially injected unique bugs and reporting the cause once found.

LAVA dataset (B. Dolan-Gavitt et al. 2016) is a ground-truth corpora by automatically injecting large number of realist bugs into programs. The original LAVA dataset consist of 69 versions of the same file, each of them with a single bug injected into it. The alternative version, LAVA-M, consists of four coreutils programs with a large number of injected bugs (See Table 2.1). Each bug has an unique identifier displayed when the bug is triggered succesfully before the program crashes. This allows the unique identification of bugs avoiding duplications when testing a

fuzzer. This dataset is used by many of the state of the art fuzzers to compare the results like Gan et al. 2018, P. Chen and H. Chen 2018, Li et al. 2017 or Aschermann et al. 2019.

TABLE 2.1: Total number of formal unique bugs in each test of LAVA-M benchmark

base64	**md5sum**	**uniq**	**who**
44	57	28	2136

FIGURE 2.1: Representation of the workflow of American Fuzzy Lop

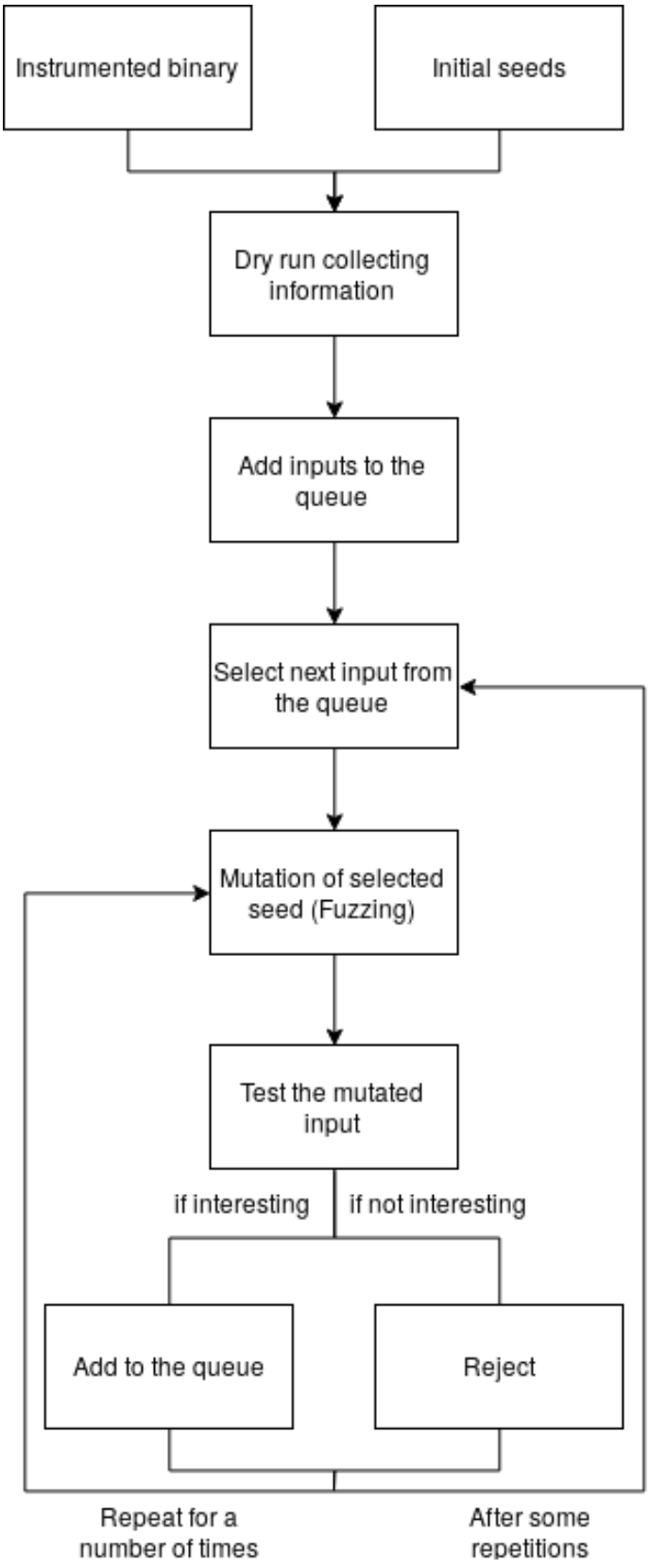


FIGURE 2.2: Representation of the mutating/fuzzing stage of American Fuzzy loop

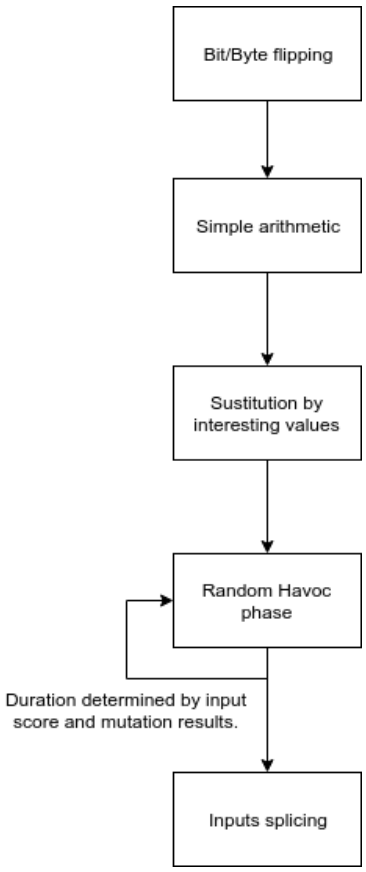
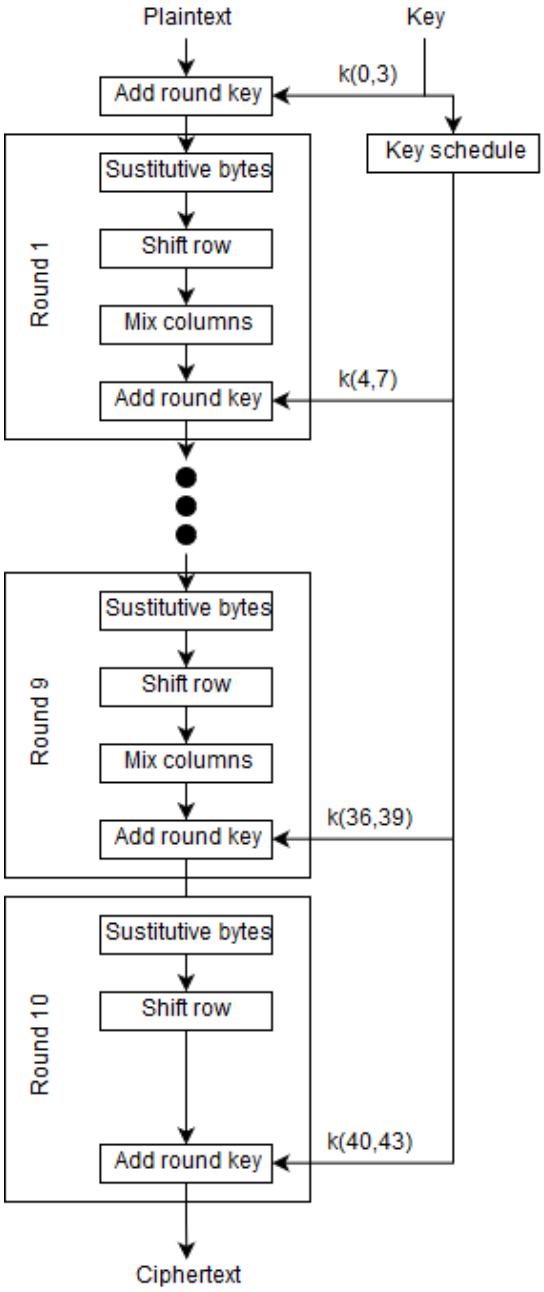


FIGURE 2.3: Representation of the AES input, each cell in the matrix represent a byte of the AES input. All operations of AES are done as cells, rows and columns operations

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

FIGURE 2.4: AES encryption algorithm, the AES formal algorithm performs 10 rounds of encryption in each round 4 operations are performed: a byte substitution, a row shifting, a column mixing operation and a key addition.



Chapter 3

Design and Implementation

3.1 PCA Guided Fuzzing

One of the major impacts in a fuzzer's success is to be able to recognize interesting seeds to fuzz. In AFL this is done in two ways: first, by adding seeds with interesting bitmaps as explained in Section 2.2.2, and second, by setting the time used to fuzz each seeds by the scoring function as described in Section 2.2.5. AFL uses its bitmap to hold the trace of a program execution and then analyses this trace to see how interesting a seed is. Comparing a full seed bitmap with the rest is inefficient and costly. This is why AFL relies on other measures like path depth, or bitmap coverage, to compare seeds, specially when setting the score associated with a seed. Previous work done by Martin 2018 shows how the seeds could be reduced from a 2^{16} dimensional array to a 2 dimensional array by using PCA and still identify bugs. This means that a reduced 2 dimensional array contains enough information about the seed to compare whether a seed can lead to new bugs. In Martin 2018 implemented a guided measure for AFL where it reduce the dimension of bitmap and it compares with the reduced versions of the previous 100 fuzzed bitmaps. The implementation was buggy and incomplete, nonetheless it showed a better performance than the Vanilla AFL scoring implementation.

One of the major issues of this implementation is relying on an external python code to perform the PCA reduction. This means that the fuzzing had to be done in 2 steps, first the fuzzer worked with vanilla score while collecting bitmaps, then the PCA operator was calculated and finally the fuzzer started again using the PCA calculations. This implementation suffered from a number of issues like a big performance impact due to having to stop the fuzzer to call external code and losing all the previous work done by the fuzzer when restarting the fuzzer with PCA guidance. This also means that when the fuzzer was restarted it needed to perform a dry run of initial seeds. This can be expensive, especially when large dictionaries are used or when there are a large number of initial seeds.

Another problem with the PCA guidance is that while the measure is particularly good at analyzing differences behaviors in heavy correlated tuples, for example loops, it is useless when a new tuple is discovered after the PCA operator is calculated. This is because the calculation of PCA relies on the correlation of the data available, and therefore it ignores any new information as the tuple correlation by the time when the operator was calculated was 0. To address this, our new implementation has the ability of relearning the PCA operator while consider necessary. When calculating a new PCA operator based on new bitmaps with different tuples, the operator will take into consideration these new tuples. This was infessible in the previous implementation when the fuzzer was externally stoped and relunched aftera fine period of 6 hours.

Also we also extended Martin 2018 by using multiple guidance measures like maximum distance, minimum distance and average distance between targets. After some initial testing we found that maximun distance dominates and it was the best measure to choose, so all further experiments were performed by using maximun distance as the score measurement.

For these reasons we consider a reimplementation of the PCA guidance fuzzer fully written in C/C++ and fully integrated with AFL. This enable us to improve performance by being able to calculate the PCA operator without stopping the fuzzer and it allows further improvements like being able to recalculate the PCA operator when necessary.

3.1.1 PCA implementation

Our implementation in C++ using the armadillo library version 9.400.3 developed by Sanderson and Curtin 2016, and OpenBLAS version 0.3.6 developed by Xianyi and Kroeker 2019. We developed a wrapper to be able to call these functions in C. First, when a new operator is calculated the fuzzer reads the bitmap files saved, then it subtracts the mean to the dataset and calculates the PCA operator. Both the mean and the operator are saved and share to other fuzzers using shared files. Calculating PCA using the covariance matrix is unfeasible due to the large matrix 65,0000x65,0000. In order to solve the computational problem we use a sparse matrix representation of the data, as most of our data are zeros. We calculate a PCA operator using an algorithm for singular value decomposition on sparse matrices. This allows us to calculate the full PCA component of the data instead of doing incremental PCA as performed in the first version of PCA^{2}FL. Also, after calculating the scores, the bitmap files except the 1000 more recent are removed so new bitmaps generated after would have more importance in the next calculation of the PCA score. We leave the 1000 latest bitmap files so the fuzzer can have enough data for the next calculation of the PCA in case really need bitmaps were discovered. (see Algorithm 3.1)

Algorithm 3.1 Algorithm of the PCA calculator implementation

```
procedure CALCULATEPCA
    for bitmap_file do
        dataset += read(bitmap_file) as uint8 vector
    DataMean = mean(dataset)
    dataset -= Datamean
    [U,S,V] = svd(dataset)
    V = first 2 columns of V with the highest values in S.
    save(DataMean, V)
    for bitmap_file except latest 1000 files do
        delete(bitmap_file)
```

After the PCA operator has been computed and saved successfully, the master fuzzer calls the other fuzzers by using the signal SIGUSR2. When a fuzzer receives the signal, it will load the PCA operator and the mean of the data from a shared file.

As we do not restart the fuzzer, the bitmaps of the seeds executed before the PCA operator is available need to be saved so the scores can be calculated if needed. In order to do that we save the bitmap of the seeds and when the operator is available we calculate the score and delete the bitmaps.

When the score needs to be calculated and the PCA operator is available for the fuzzer, the fuzzer calculates the reduced dimension of the data from 65,000 dimensions to 2 dimensions. For the first 100 seeds it saves the reduced data points in a round robin queue of the 100 latest data points. After the 100th seed then the fuzzer starts using the PCA metric by comparing the current seed reduced data point with the 100 previous data points. We used 3 different metrics, minimun, average and maximun euclidian distance. Our early experiments show that the maximun distance seems to perform better than the rest so we fill focus on it. (See Algorithm 3.2) The scored, as described in Section 2.2.5 is used to set the length of the havoc phase.

Algorithm 3.2 Pseudo algorithm of the maximum distance scoring

```
procedure CALCULATE_SCORE
    if PCA_compoents not loaded then return vanilla_score
    else
        data_point = reduce_dimensions(bitmap)
        if data_points_queue contains 100 elements then
            score = max_distance(data_point, data_points_queue)
        else
            score = vanilla_score
        data_points_queue = data_points_queue ⋃ data_point
        return score
```

3.2 AES

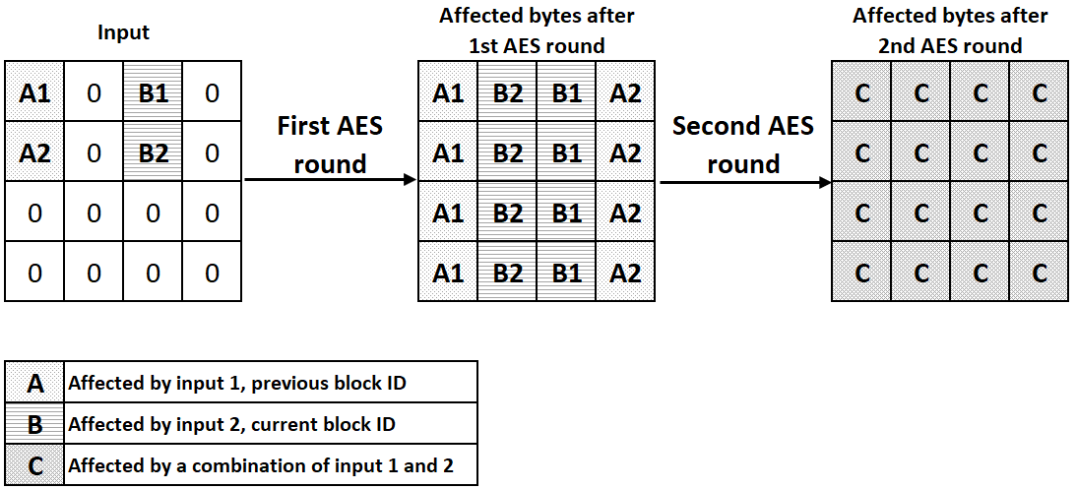
One of the biggest problems in vanilla AFL is the implementation of the hashing function for inserting tuples in the bitmap, as described in algorithm Algorithm 2.1 of Section 2.2.3. While extremely fast and simple, it relies heavily on the diffusion of the random code blocks IDs. One of the major issues is the use of the right shift to indicate the direction of the tuple as explain in Zalewski 2016b. This greatly reduces the space for the hashing as all the block ID, that differ only in the last bit hash to the same value. This drawback is usually acceptable because the number of code block ids is usually small and other hashing implementations would greatly affect performance.

We aim to solve this collision problem by implemented a hardware accelerate hashing function based on AES. AES implementation guarantees good cryptographical properties like diffusion. This means that changes on one bit change in the input will produce a change in half of the output bits. In the vanilla implementation this doesn't happen a single bit change in the input only produce a bit change in the output. It also does not need to encode the direction by right shifting as the direction is implicit in the position of the block ids when performing PCA . The most computational costly part of AES is the key schedule because we are not trying to develop a security scheme, we are able to omit that expensive step entirely using a fixed key. Also one of the major advantages of using AES is the intel hardware implementation of the AES functions as described in Gael 2012 and Akdemir 2010 that greatly increase performance over other hashing functions using only ≈ 1.3 cycles/byte.

3.2.1 AES-based hash function

Our algorithm described in Algorithm 3.3 works by inserting the prev block in the byte 0 and 1 and the current block in the byte 8 and 9. Then we perform 2 rounds of AES, as the first round with only data in 2 bytes will only change 2 of the output columns. Next we return the first byte of the result as our hash. This results in a hashing function that does not depend on the right shifting to preserve the direction of the edge and in theory should give less collision ratio. Our algorithm does not perform data encryption, therefore only 2 rounds of AES are used instead of the 10 used in the algorithm and a fixed key is used. Using 2 rounds of AES the algorithm can spread the changes of the input bytes to all the output bytes of the results as described in Figure 3.1. We tested a second but more costly version of the algorithm described in Algorithm 3.4, were all the hash depends on all the output by performing an xor of all the output bits. We realized that using a key of 0 if we tried to encoded paths to the same block the results were always 0, therefore a non zero key was used.

FIGURE 3.1: Description of the affected bytes by each input of the AES function with key 0. The left image represents the input, the middle image represent the altered bytes by each input after 1 round of AES. The final image represents the final altered bytes after 2 rounds of AES, in this stage the result is affected by both inputs.



Algorithm 3.3 Pseudo algorithm for first AES hashing function

```
procedure AESHASHING1
    key = 0;
    edge[byte 0,1] = previous_block
    edge[byte 8,9] = current_block
    edge rest of bytes until 128 are zeros
    hashaes = aes_round(edge, key)
    hashaes = aes_round(hashaes, key)
    previous_block = current_block
    return hashaes[byte 0,1]
```

We implemented the first algorithm in both, gcc/Assembly version of AFL compiler and clang llvm version of AFL compiler.

Algorithm 3.4 Pseudo algorithm for second AES hashing function

```
procedure AESHASHING2
    key = [0x00,0x01,...,0x0f];
    edge[byte 0] = previous_block
    edge[byte 8] = current_block
    edge rest of bytes until 128 are zeros
    hashaes = aes_round(edge, key)
    hashaes = aes_round(hashaes, key)
    previous_block = current_block
    return XOR all bytes of hashaes
```

Chapter 4

Experiments and Results

4.1 PCA Experiments

4.1.1 Single fuzzer experiments

We redo the experiments by running our implementation of PCA^{2}FL on the 4 targets of the LAVA-M dataset. Each target is run by an individual fuzzer, 30 times, during 24 hours with the input seeds provided by the LAVA-M dataset. These experiments follow the old approach of PCA^{2}FL of training for 6 hours and fuzzing for 18 hours as we consider that a single fuzzer can not probably generate enough bitmaps to learn new PCA operators specially with the really low bitmap coverage of the targets showed in Table 4.2. The sum of unique bug identifiers found by each fuzzer is show in Table 4.1.

TABLE 4.1: Number of unique bugs found by all the 30 individual fuzzers after running for 24 hours. PCA^{2}FL tests have been running 6 hours of collecting seeds with vanilla guidance and 18 hours with PCA guidance

	base64	**md5sum**	**uniq**	**who**
Vanilla	0	1	3	0
Previous PCA2FL	30	1	0	0
PCA2FL	**31**	**4**	**4**	**4**

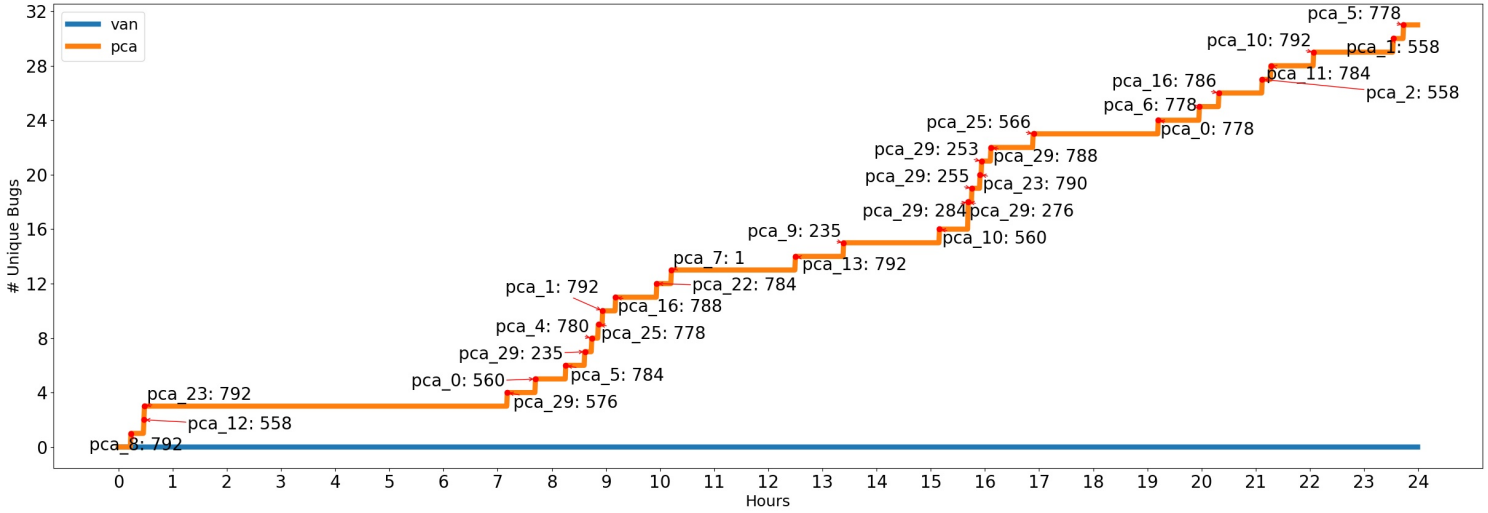
TABLE 4.2: Average statistics of 30 tests running a single fuzzer. PCA^{2}FL tests have been running 6 hours of collecting seeds with vanilla guidance and 18 hours with PCA guidance

	base64		**md5sum**		**uniq**		**who**	
	PCAFL	**Vanilla**	**PCAFL**	**Vanilla**	**PCAFL**	**Vanilla**	**PCAFL**	**Vanilla**
Number of Executions done	32,030,159	40,411,024	14,991,555	19,547,524	35,325,090	48,108,625	23,016,914	37,300,661
Number of Total paths	**347**	342	**413**	364	98.38	**104.3**	**262**	225
max depth found	11.5	11.36	10.52	9.3	5.98	07.06	13.21	16.53
bitmap coverage	0.55	0.55	0.92	0.89	0.38	0.38	20.59	20.59

TABLE 4.3: Number of unique bugs IDs founded by state of art fuzzers after 5 hours, the number in parenthesis indicate the number of unidentified bugs found by each fuzzer

	base64	**md5sum**	**uniq**	**who**
Steelix	43	28	7	194
Angora	28(+1)	48(+4)	57	1443(+98)
Redqueen	28(+1)	44(+4)	57(+4)	2134(+328)

FIGURE 4.1: Unique Bugs Identifiers found by all independent fuzzers for BASE64 in 30 experiments during 24 hours. PCA runs in seeds collection with vanilla guidance for 6 hours and 18 hours with PCA guidance. The labels indicates, pca or van depending on the guidance, followed by the fuzzer number and the bug identifier



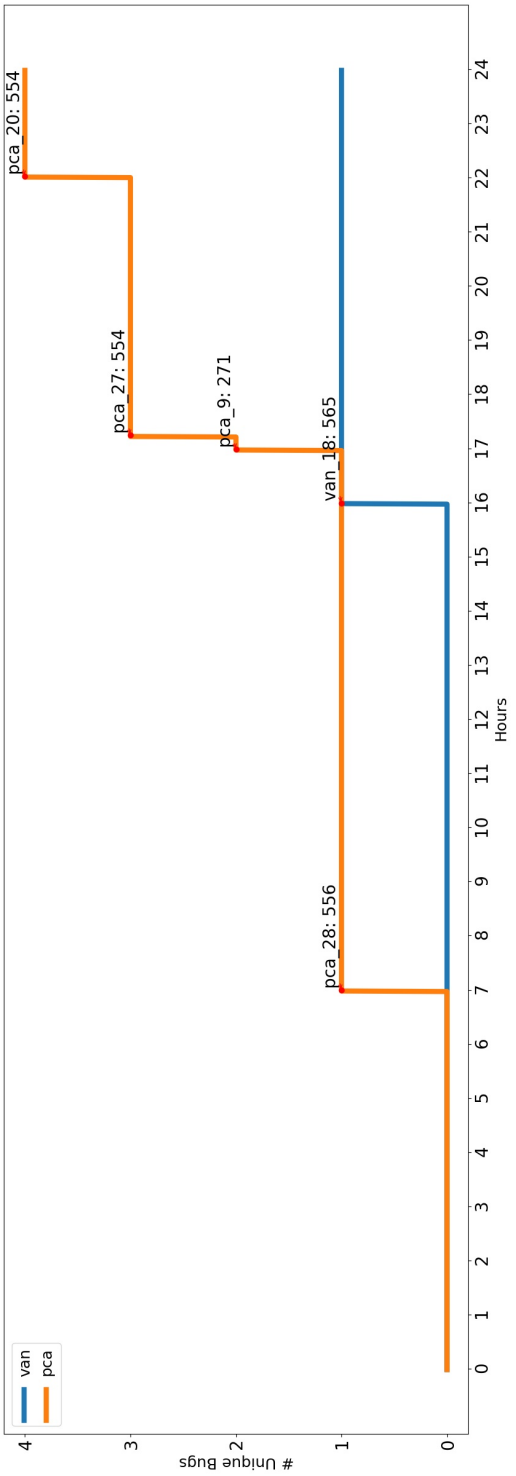


FIGURE 4.2: Unique Bugs Identifiers found by all independent fuzzers for MD5SUM in 30 experiments during 24 hours. PCA runs in seeds collection with vanilla guidance for 6 hours and 18 hours with PCA guidance. The labels indicates, pca or van depending on the guidance, followed by the fuzzer number and the bug identifier

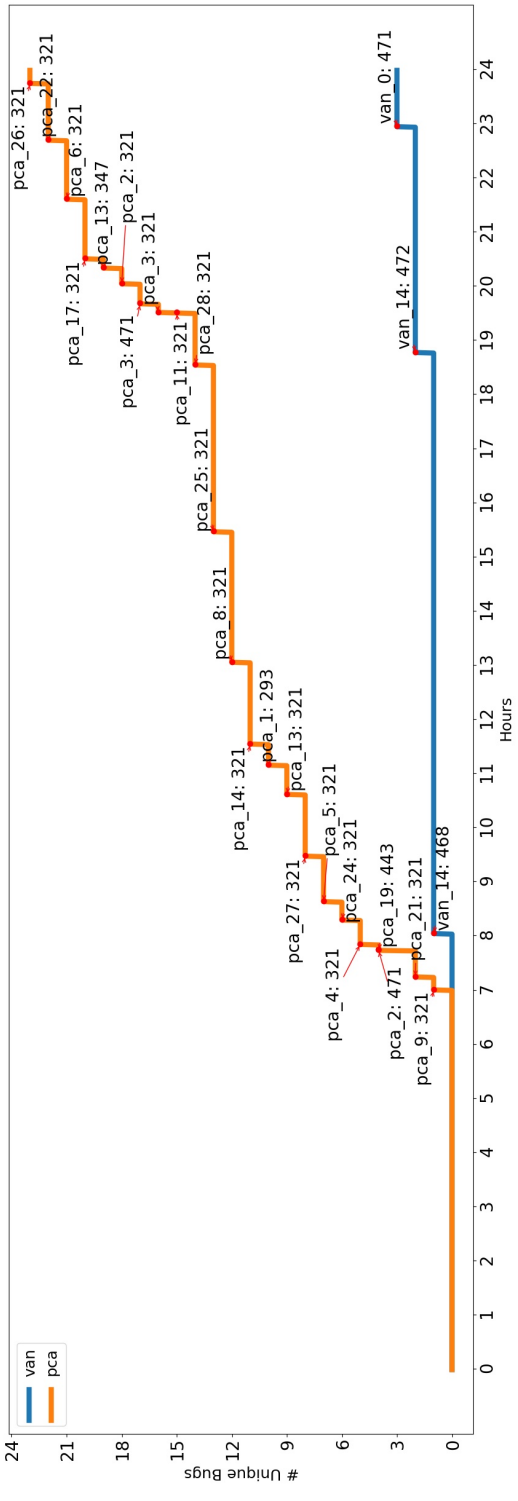


FIGURE 4.3: Unique Bugs Identifiers found by all independent fuzzers for UNIQ in 30 experiments during 24 hours. PCA runs in seeds collection with vanilla guidance for 6 hours and 18 hours with PCA guidance. The labels indicates, pca or van depending on the guidance, followed by the fuzzer number and the bug identifier

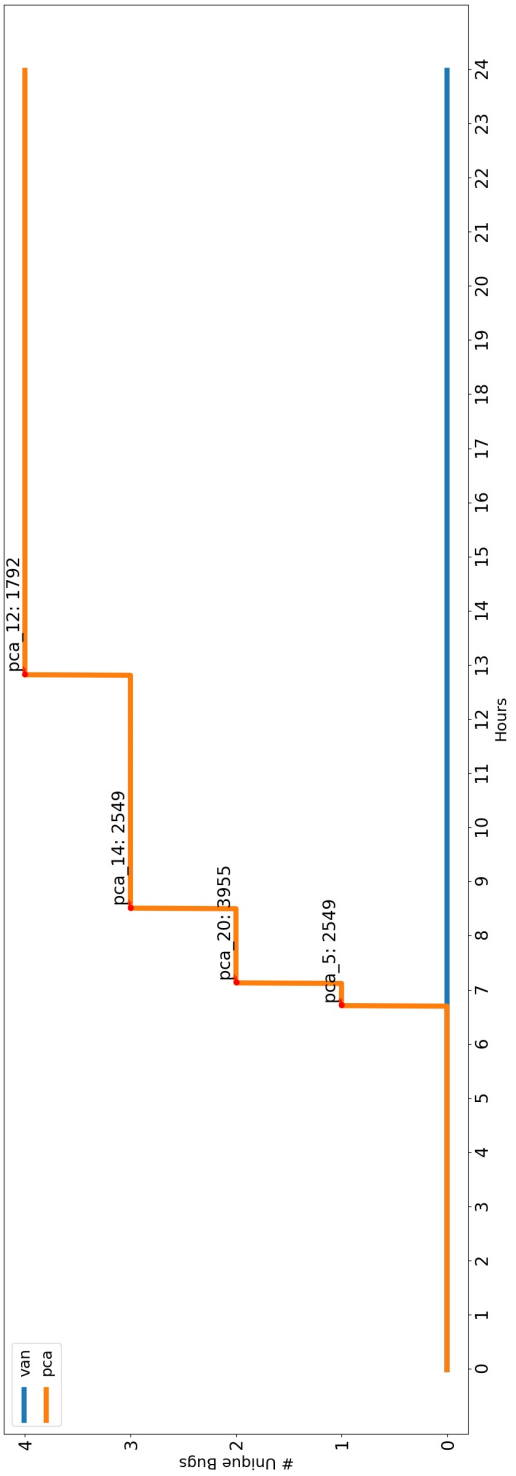


FIGURE 4.4: Unique Bugs Identifiers found by all independent fuzzers for WHO in 30 experiments during 24 hours. PCA runs in seeds collection with vanilla guidance for 6 hours and 18 hours with PCA guidance. The labels indicates, pca or van depending on the guidance, followed by the fuzzer number and the bug identifier

The results shown in Table 4.1 and the graphs shown in Figures 4.1, 4.2, 4.3 and 4.4 show how the PCA guidance greatly improves the tests results. However, as a drawback it also affects performance as shown in Table 4.2. Its remarkable how the WHO test resulted in such a lower bug found however the LAVA-M author mentions that multiple fuzzers have problems dealing with this target B. Dolan-Gavitt 2018. Although PCA^{2}FL performs better than vanilla AFL it can be not compared with some state of the art fuzzers like Steelix by Li et al. 2017, Angora by P. Chen and H. Chen 2018 or Redqueen by Aschermann et al. 2019, that use more advance techniques for fuzzing like program status or magic byte comparations. The results of those fuzzers in LAVA-M are described inTable 4.3.

4.1.2 Multiple synchronized fuzzer experiments

Due to the big success of PCA on single thread experiments we considered the possibility of horizontal scaling by running the same LAVA-M dataset on 15 tests of 8 fuzzers in syncronized mode for 18 hours. During theses experiments the PCA operator is calculated using the bitmap of all the fuzzers, therefore we consider that relearning the PCA operator every 6 hours could lead to a better score as new edges are found. Also, following the LAVA-M author notes B. Dolan-Gavitt 2018 and B. Dolan-Gavitt 2016, we improve the results of our fuzzers by providing a simple dictionary generated from the target binary.

TABLE 4.4: Average of the number of unique ID bugs found by the 8 fuzzers in 15 tests running on synchronized mode, the number in parenthesis indicates the total number of unidentified bugs found in all the tests

	base64	**md5sum**	**uniq**	**who**
Vanilla	48	42 (+20)	23	1384 (+22)
Full PCA	48	44 (+27)	23	1443 (+13)
Mix	48	43 (+17)	23	1462 (+116)
All bugs in dataset	**44**	**57**	**28**	**2136**

The results shows how PCA implementation either mix or full PCA behaves the same or better than the vanilla implementation. It is remarkable how in base64, all of the tests found 48 unique bugs ids when in the LAVA-M dataset B. Dolan-Gavitt et al. 2016 there are only 44 documented. Also during the runs of WHO some of the fuzzers found unexpected bugs: up to 90 undocumented bugs in a single tests.

4.2 AES

To test both algorithms and the vanilla version we implemented a tracing method in llvm mode. When a path is executed it writes the current block ID into a file. After all the traces from a program are obtained, we reduced the output file, only keeping unique paths, and lastly we run the 2 AES algorithms and vanilla algorithms on all pairs of unique previous and current block ID to check the number of collisions they would have produced. We tried to find high collision targets based on Gan et al. 2018 in order to prove which one is better, libtorrent/dumptorrent (75.29% collisions), libav (74.85% collisions) and vim (61.4% collisions). However after trying to reproduce the results as described by the paper by using their seeds and fuzzing the same targets with 1 fuzzer during 200 hours the results shows a really low bitmap

utilization and not enough collision ratio to be able to make any conclusion about which algorithm is better.

To test the performance we first run both implementations of AES on 4 fuzzers during 24 hours for OBJDUMP, we choose objdump as we were able to find a new not documented bug on the target. After testing AES implementation on objdump on 4 fuzzers during 24 hours, the number of executions done show how the AES1 execution speed is comparable to vanilla speed with a 3.7% of slowdown, however AES2 suffers big execution penalties with a 21.7% slowdown in number of executions compared with the vanilla implementation. We also tested the executions of Vanilla and AES1 on libtorrent following the testing described in Gan et al. 2018, 1 fuzzer during 200 hours. The results shows how AES1 also performs similar to Vanilla with only a 1.78% performance slowdown compared the the vanilla hashing implementation.

TABLE 4.5: Statistics on bitmap coverage, number of unique paths and collisions on the reproduced tests from Gan et al. 2018, run with a single fuzzer during 200 hours

	libtorrent	**libav**	**vim**
bitmap coverage	5.80%	1.18%	5.61%
Number of Unique paths	7965	1653	7336
Collisions Vanilla	388	19	355
Collisions AES1	480	22	391
Collisions AES2	444	14	408

Chapter 5

Conclusion and future work

5.1 Conclusion

Our implementation of PCA^{2}FL improves the fuzzer performance in finding bugs in the LAVA-M targets we considered. We observe that the number of executions using PCA guidance is less than the number using vanilla AFL. Although our implementation of PCA^{2}FL can not compete with some state of the art fuzzers like Angora by P. Chen and H. Chen 2018, Steelix by Li et al. 2017 or Redqueen Aschermann et al. 2019, it shows an improve over the vanilla seed scoring technique, this indicates that implementing this scoring along with more advanced techniques instrumentation techniques could be beneficial. After trying to reproduce the results from CollAFL by Gan et al. 2018, we were not able to reproduce the results for high collision targets LIBTORRENT, LIBAV and VIM and therefore, due to the low bitmap coverage and the lack of collisions we can not validate that our AES implementation reduce collisions compared to the vanilla implementation. After testing both vanilla and AES hashing functions we can demonstrate that our AES hashing algorithm has a minimal impact in the fuzzing performance, measured by number of executions.

5.2 Future work

Future work should perform further investigation about the effect of AES in the bitmap collisions by finding appropriate tests or creating a synthetic benchmarks in order to prove that our function reduces collisions. PCA guidance gives better results than the vanilla AFL guidance, therefore it would be intesting to test the results of a combination between AES and PCA, specially on real applications with high bitmap coverage. PCA guidance could not compete with state of the art fuzzers, however an implementation of PCA guidance with more advance instrumentation could benefit some of the edge fuzzers.

Appendix 1

Project description

In this project we will experiment with a couple of modifications to a popular fuzzer, AFL, and evaluate their performance, in terms of code coverage and the number of unique crashes. The first modification is based on applying a technique based Principal Component Analysis (PCA) to decide the ranking of seeds selected for testing. The second modification aims to modify the hashing function used in AFL using Intel hardware-accelerated AES encryption (using the AES-NI instructions). An earlier version of the PCA-base modification has been implemented by a student in a previous project; this project will improve on that implementation and perform a more extensive evaluation. The second modification (AES-based hashing) has not been implemented, and will be done in this project. Several testing scenarios will be designed and evaluated; these include evaluating each modification in isolation, and in combination. The student will also need to setup test environments in a high-performance computing platform.

Appendix 2

Independent study contract

- **UniID:** U6759601
- **SURNAME:** Pinilla López
- **FIRST NAMES:** Jorge
- **PROJECT SUPERVISOR:** Charles Gretton, Adrian Herrera, Alwen Tiu
- **FORMAL SUPERVISOR:** Charles Gretton, Alwen Tiu
- **CORSE CODE, TITLE AND UNITS:** COMP4650 Advanced Computing Project 12 units
- **SEMESTER:** S1
- **YEAR:** 2019

PROJECT TITLE: Improving Fuzzing Performance using Hardware Accelared Hashing

LEARNING OBJECTIVES:

- Understand the foundational concepts and implementation details of fuzzing-based security testing of software
- Learn how to apply knowledge in cryptography to real-world applications
- Understand how to perform sound scientific analysis on the performance of fuzzing techniques

ASSESSMENT:
In this project we will experiment with a couple of modifications to a popular fuzzer, AFL, and evaluate their performance, in terms of code coverage and the number of unique crashes. The first modification is based on applying a technique based Principal Component Analysis (PCA) to decide the ranking of seeds selected for testing. The second modification aims to modify the hashing function used in AFL using Intel hardware-accelerated AES encryption (using the AES-NI instructions). An earlier version of the PCA-base modification has been implemented by a student in a previous project; this project will improve on that implementation and perform a more extensive evaluation. The second modification (AES-based hashing) has not been implemented, and will be done in this project. Several testing scenarios will be designed and evaluated; these include evaluating each modification in isolation, and in combination. The student will also need to setup test environments in a high-performance computing platform.

Assessed project components	**% of mark**	**Evaluated by**
Repor	45%	examiner
Artefact	45%	supervisor
Presentation	10%	course convenor

Appendix 3

Artefacts description

3.1 AFL-PCA

The original version of AFL implementation version 2.52b from Zalewski 2016a. We modify the file AFL-FUZZ.C with the PCA implementations creating:

- **afl-fuzz-max.c:** AFL with maximun distance guidance
- **afl-fuzz-min.c:** AFL with minimun distance guidance
- **afl-fuzz-avg.c:** AFL with average distance guidance
- **Makefile:** Compile all the code with the new PCA files

We also added the files:

- **PCA.cpp:** Code for the PCA calculation and tranformations
- **PCA.h** Common header between C and C++ for the PCA implementation

For an easier compilation ande deployment we added in the folder extra a static version of the libraries armadillo library version 9.400.3 developed by Sanderson and Curtin 2016, and OpenBLAS version 0.3.6 developed by Xianyi and Kroeker 2019. By using a static version we can create a single executable that can be transfer to other computers without the need of having the library. This is done to facilitate the execution and testing of the program on multiple servers with similar characteristics like high-performance clusters.

3.2 AFL-AES

The original version of AFL implementation version 2.52b from Zalewski 2016a. We modify the files:

- **afl-as.h:** Definition of the instrumentation performed by AFL when compiling with gcc or g++
- **llvm-mode/afl-llvm-pass.so.cc:** Instrumentation performed by AFL when compiling with clang or clang++, we add AES hashing and program tracing
- **afl-llvm-rt.o.c** Function definitions for the code implementation, we add the definition for the tracing function.

We add extra functions in the folder AES-AUX to read and test the generated tracing of the program when compiled with the modified AFL-CLANG-FAST. This folder contains:

- **reduce.py:** Python program to read a tracing file and reduce it keep only the unique tuples
- **simulate.cpp:** Testing program to simulate random tuples to test the AES implementation on assembly
- **AES.asm:** Assembly implementation of the AES implementations and vanilla implementations
- **read_trace.cpp:** File to read a tracing without reduction and perform testing of the file and both vanilla and AES implementation
- **read_reduced.cpp** File to read a reduced tracing file and perform testing of the file and both vanilla and AES implementation

3.3 Testing Benchmarks

The LAVA-M folder contains the source code of the benchmark binaries tested in our fuzzing. No modification has been done. The seeds folder contains the seeds for the fuzzing provided by the LAVA-M benchmarks. The file MAKE_TESTCASES.SH from B. Dolan-Gavitt 2016 generate the dictionary test cases from a binary. The dictionaries used for the tests are also in the file DICS.ZIP. The build binaries with vanilla AFL instrumentation for the benchmark are provided in the folder BIN

3.4 Others

The folder PLOT contains the python code to perform the plotting of the results. The files PLOT_SINGLE.PY and PLOT_SYNC.PY plot the results of single and synchronized mode respectively. The rest of files are helpers to those functions. We provide a script SYSTEM_CONFIG.SH to perform the system configurations before runing AFL
To help with the testing we also give 3 running scripts:

- **run.sh:** Runs instances of AFL on single mode, all of them are independent from the others
- **sync-run.sh:** Runs instances of AFL on synchronized mode.
- **run-multiple.sh:** Runs tests of sync-run, each tests consist of 8 fuzzers running for 18h

3.5 Testing and Building

For the AES implementation the testing of the code has been done by using scripts mention in appendix 3.1 before integrating the code and using the tracing files after the code has been implemented.
For the PCA implementation, the testing is done by tracing the PCA score and the fuzzer score. Each fuzzer logs their scores into a file called pca_log.csv that can be read after to verify the implementation of the PCA operator.
The build is made is automatically in a dockers container using gitlab continous integration. The build compiles the AFL, the LAVA-M benchmarks and produce a zip file with only the resulted binaries and the scripts to run the tests.

Appendix 4

README file

```
# pca2fl: an extension of afl

## Compilation:
The source code of pca2fl is located in the afl-PCA and afl-PCA folders
    .

There will be 4 different versions of afl-fuzz:
- afl-fuzz : Vanilla from afl official source
- afl-fuzz-avg : PCA version with avg distance
- afl-fuzz-min : PCA version with min distance
- afl-fuzz-max : PCA version with max distance

Instalation tested in:

Debian 9
Kernel 4.9.0-9-amd64


Compilation of AFL

Packages:
    build-essentials
        -gcc: 6.3.0
        -g++: 6.3.0
Instructions:
    cd afl
    make clean
    make
Notes:
    A static version of Armadillo and OpenBLAS libraries are available
        in afl/extra

Compilation of LLVM mode (with tracing):

Packages:
    llvm: 3.8
    clang: 3.8
    libz-dev
Instructions:
    build AFL as explained in Compilation of AFL
    cd afl/llvm-mode
    make
Note:
    This will build afl-clang-fast and afl-clang-fast
    This version incorporate tracing of the program.


Compilation of LAVA-M test
```

```
Packages:
    libacl1-dev: 2.2.52-3+b1
    automake: 1-15
    texinfo: 6.3
Instructions:
    cd LAVA-M/$test/coreutils-8.24-lava-safe/
    make clean
    ./configure --prefix=`pwd`/lava-install LIBS="-lacl"
    make
    make install
Note:
    - The result buggy binary is in LAVA-M/$test/coreutils-8.24-lava-
      safe/lava-install/bin folder
    - To verify the number of bugs in the binary use LAVA-M/$test/
      validate.sh script
    - To compile it with afl instrumentation add CC=../../afl/afl-gcc
      before ./configure like
       CC=../../afl/afl-gcc ./configure --prefix=`pwd`/lava-install
          LIBS="-lacl"


## Running the fuzzer

Before you run the AFL, you may need to change some system
    configuration. Run the system_config.sh script for this.

Create a folder call afl and copy your fuzzer binaries (afl-fuzz, afl-
    fuzz-max...) to that folder

- To run AFL in stand alone mode, use the run.sh script.
For example: assuming the binary to be fuzzed is base64 (located in the
     same directory as run.sh), to run 30 instances of AFL-max with
    relearning time of 6h, use the following:

./run.sh -t base64 -n 30 -o . -s max -p 360

- To run vanilla pca use -v
- To run on empty seeds use -e
- To run with a dictionary use -x $DICT_FOLDER:
- To specify the reload time use -p $RELOAD_TIME

- There are 3 metrics for the pca_score function: min, max, avg

- To run AFL in sync mode (processes can import queues from each other)
     use the sync-run.sh script. (Same syntax as run.sh, just replace
    run.sh in the above example with sync-run.sh).

- To run multiple tests of AFL sync mode use run-multiple.sh


Optiosn added to afl-fuzz-max, afl-fuzz-min, afl-fuzz-avg:

-K collect bitmaps, this will make the fuzzer to start collecting
    bitmaps
-p $TIME Recalculate PCA operator in $TIME minutes
-I $PIDS Signall the pids $PIDS after calculating a new PCA operator
-P $PATH When the signal to load the new PCA operator is received, the
    fuzzer will load the operator from the files in $PATH.
     Usually $PATH is the same as the output path of the fuzzer.

Compiled versions of the binaries with vanilla hashing are provided in
    the bin folder. Compiled version of afl-fuzz are provided in the
    afl folder
```

Appendix 5

Vanilla score calculation

```
/* Calculate case desirability score to adjust the length of havoc
    fuzzing.
   A helper function for fuzz_one(). Maybe some of these constants
       should
   go into config.h. */

static u32 calculate_score(struct queue_entry* q) {

  u32 avg_exec_us = total_cal_us / total_cal_cycles;
  u32 avg_bitmap_size = total_bitmap_size / total_bitmap_entries;
  u32 perf_score = 100;

  /* Adjust score based on execution speed of this path, compared to
      the
     global average. Multiplier ranges from 0.1x to 3x. Fast inputs are
     less expensive to fuzz, so we're giving them more air time. */

  if (q->exec_us * 0.1 > avg_exec_us) perf_score = 10;
  else if (q->exec_us * 0.25 > avg_exec_us) perf_score = 25;
  else if (q->exec_us * 0.5 > avg_exec_us) perf_score = 50;
  else if (q->exec_us * 0.75 > avg_exec_us) perf_score = 75;
  else if (q->exec_us * 4 < avg_exec_us) perf_score = 300;
  else if (q->exec_us * 3 < avg_exec_us) perf_score = 200;
  else if (q->exec_us * 2 < avg_exec_us) perf_score = 150;

  /* Adjust score based on bitmap size. The working theory is that
      better
     coverage translates to better targets. Multiplier from 0.25x to 3x
         . */

  if (q->bitmap_size * 0.3 > avg_bitmap_size) perf_score *= 3;
  else if (q->bitmap_size * 0.5 > avg_bitmap_size) perf_score *= 2;
  else if (q->bitmap_size * 0.75 > avg_bitmap_size) perf_score *= 1.5;
  else if (q->bitmap_size * 3 < avg_bitmap_size) perf_score *= 0.25;
  else if (q->bitmap_size * 2 < avg_bitmap_size) perf_score *= 0.5;
  else if (q->bitmap_size * 1.5 < avg_bitmap_size) perf_score *= 0.75;

  /* Adjust score based on handicap. Handicap is proportional to how
      late
     in the game we learned about this path. Latecomers are allowed to
         run
     for a bit longer until they catch up with the rest. */

  if (q->handicap >= 4) {

    perf_score *= 4;
    q->handicap -= 4;

  } else if (q->handicap) {
```

```
    perf_score *= 2;
    q->handicap--;

  }

  /* Final adjustment based on input depth, under the assumption that
     fuzzing
     deeper test cases is more likely to reveal stuff that can't be
     discovered with traditional fuzzers. */

  switch (q->depth) {

    case 0 ... 3:   break;
    case 4 ... 7:   perf_score *= 2; break;
    case 8 ... 13:  perf_score *= 3; break;
    case 14 ... 25: perf_score *= 4; break;
    default:        perf_score *= 5;

  }

  /* Make sure that we don't go over limit. */

  if (perf_score > HAVOC_MAX_MULT * 100) perf_score = HAVOC_MAX_MULT *
      100;

  return perf_score;

}
```

Bibliography

[1] Kahraman et al. Akdemir. "Breakthrough AES Performance with Intel AES New Instructions". In: *Intel white paper* (2010). URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/10TB24_Breakthrough_AES_Performance_with_Intel_AES_New_Instructions.final.secure.pdf.

[2] Cornelius Aschermann et al. *REDQUEEN: Fuzzing with Input-to-State Correspondence*. 2019. URL: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/.

[3] Peng Chen and Hao Chen. *Angora: Efficient Fuzzing by Principled Search*. 2018. eprint: arXiv:1803.01307.

[4] Brendan Dolan-Gavitt. *Fuzzing with AFL is an Art*. July 2016. URL: http://moyix.blogspot.com/2016/07/fuzzing-with-afl-is-an-art.html.

[5] Brendan Dolan-Gavitt. *Of Bugs and Baselines*. Mar. 2018. URL: http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html.

[6] B. Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition". In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 110–121. DOI: 10.1109/SP.2016.15.

[7] Gael. *Introduction to Intel AES-NI and Intel Secure key instructions*. July 2012. URL: https://software.intel.com/en-us/node/256280.

[8] S. Gan et al. "CollAFL: Path Sensitive Fuzzing". In: *2018 IEEE Symposium on Security and Privacy (SP)*. May 2018, pp. 679–696. DOI: 10.1109/SP.2018.00040.

[9] George Klees et al. "Evaluating Fuzz Testing". In: (Oct. 2018), pp. 2123–2138. DOI: 10.1145/3243734.3243804.

[10] Caroline Lemieux and Koushik Sen. "FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage". In: (2017). DOI: 10.1145/3238147.3238176. eprint: arXiv:1709.07101.

[11] Yuekang Li et al. "Steelix: program-state based binary fuzzing". In: Aug. 2017, pp. 627–637. DOI: 10.1145/3106237.3106295.

[12] Jonathon Martin. *AI Guided Fuzzing*. Oct. 2018.

[13] Conrad Sanderson and Ryan Curtin. "Armadillo: A template-based C++ library for linear algebra". In: *Journal of Open Source Software* 1 (July 2016), p. 26. DOI: 10.21105/joss.00026.

[14] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. ISBN: 0321446119.

[15] Zhang Xianyi and Martin Kroeker. *OpenBLAS*. Apr. 2019. URL: https://www.openblas.net/.

[16] Michal Zalewski. *american fuzzy lop (2.52b)*. 2016. URL: http://lcamtuf.coredump.cx/afl/.

[17] Michal Zalewski. *Binary fuzzing strategies: what works, what doesn't*. Aug. 2014. URL: https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html.

[18] Michal Zalewski. *Technical "whitepaper" for afl-fuzz*. 2016. URL: http://lcamtuf.coredump.cx/afl/technical_details.txt.