UNIVERSITÀ DEGLI STUDI
DI TRENTO

UNIVERSITY
OF TWENTE.

Department of Information Engineering and Computer Science

Master's Degree in Computer Science

Final Dissertation

A QBDI-based Fuzzer Targeting Magic Bytes Comparisons

Supervisor

Prof. Fabio Massacci

Student

Elia Geretto

Academic year 2017/2018

Acknowledgements

I would first like to thank my supervisor at Quarkslab, Cédric Tessier, whose invaluable guidance and instructive chats allowed me to overcome all the bumps on the road towards the results presented in this document.

My gratitude goes also to the other members of my team, who bore my endless babbling about my problems and were always able to provide valuable suggestions.

I would also like to thank my friend Filippo Bigarella, who helped in smoothing all the rough edges of my writing.

Finally, I would like to express my gratitude to my advisor, prof. Fabio Massacci, who provided valuable guidance and allowed this whole experience to take place.

Contents

1 Background **8**
1.1 Fuzzing . . . 8
1.1.1 Historical overview . . . 8
1.1.2 Classification . . . 9
1.1.3 Binary targets . . . 10
1.2 Implementation details of AFL . . . 10
1.2.1 Execution trace . . . 10
1.2.2 Input handling . . . 11
1.2.3 Mutation operators . . . 11
1.3 Instrumentation . . . 12
1.3.1 Source instrumentation . . . 12
1.3.2 Static binary instrumentation . . . 12
1.3.3 Dynamic binary instrumentation . . . 13
1.4 QBDI . . . 14
1.4.1 Overview . . . 14
1.4.2 Execution cycle . . . 14
1.4.3 Architecture . . . 15

2 Problem statement **16**
2.1 Challenges in fuzzing . . . 16
2.2 Proposed solutions . . . 17

3 Steelix approach **19**
3.1 Approach overview . . . 19
3.2 Limitations . . . 21
3.2.1 Static binary instrumentation . . . 21
3.2.2 Static analysis . . . 21
3.2.3 Taint analysis . . . 22
3.2.4 Function return values . . . 22

4 Implementation **23**
4.1 Pre-existing tool . . . 23
4.1.1 Instrumentation technique . . . 23
4.1.2 Library targets . . . 23
4.1.3 Architecture . . . 24
4.1.4 Comparison unrolling . . . 24
4.2 Steelix integration . . . 25
4.2.1 Comparison progress reporting . . . 25
4.2.2 Queue handling . . . 25
4.2.3 Steelix mutation operator . . . 26
4.3 Other improvements . . . 26
4.3.1 Support for executables . . . 26
4.3.2 Dislocator . . . 27

5 Evaluation **30**
5.1 Target binaries . . . 30
5.1.1 LAVA-M . . . 30
5.1.2 ImageIO framework . . . 31
5.2 Evaluation setup . . . 31
5.2.1 LAVA-M . . . 32
5.2.2 ImageIO . . . 32
5.3 LAVA-M Evaluation . . . 33
5.3.1 Steelix comparison . . . 33
5.3.2 Bloom filter size comparison . . . 35
5.3.3 Skip probability comparison . . . 39
5.4 ImageIO evaluation . . . 40
5.4.1 Heuristic evaluation . . . 40
5.4.2 Bloom filter size comparison . . . 42
5.4.3 Skip probability comparison . . . 44

6 Conclusion **45**

Summary

Context

In recent times, the automated software testing technique that has enjoyed the most widespread success is probably fuzzing. Indeed, despite the fact that its first application dates back to 1989, its modern variations have been integrated in the software development process of various large companies [10] and open source projects [1].

In addition, fuzzing was also able to gain widespread adoption in the information security community thanks to its good scalability and low requirements regarding the knowledge of the target software. In this context, one of the most influential projects is American Fuzzy Lop [25], which also drove the adoption of grey-box fuzzers. These tools employ lightweight instrumentation in order to obtain information on the executions of the program under test. This information is then used to guide the fuzzer towards a precise goal, which is usually obtaining the highest possible coverage, but it can also be reaching a specific point in the program.

However, fuzzing has significant limitations that might hinder its effectiveness. The most important one is that, while trying to explore a program, fuzzers might not be able to reach portions of code that are protected by conditions which have a low probability of being guessed randomly. This is commonly due to the fact that, without applying specific modifications, these tools generate the test cases that are fed to the target program using only random or deterministic mutation operators; these, however, do not take into account the structure of the code.

The problem of reaching code protected by complex conditions translates into a variety of less abstract limitations; one of the most common ones is that of matching magic byte sequences and tokens, which will be treated as one and the same for the rest of this document. These conditions are commonly found in parsers and are constituted as follows: they require that a sequence of adjacent input bytes matches a specific sequence of values in order to access a given code branch.

Goal and approach

The goal of the work presented in this document was to improve the coverage obtained by a specific grey-box fuzzer, called AFL/QBDI, when exploring code containing magic bytes conditions without sacrificing performance.

The important aspect of this particular tool is that it is the first, and currently the only one, to use a dynamic binary instrumentation framework called QBDI to perform the instrumentation of the target code. This is relevant since dynamic binary instrumentation provides advantages in terms of the accuracy of the callback placement, but has a considerable performance impact and thus imposes different speed constraints on the instrumentation code.

In the literature, several solutions were proposed in order to solve the issue of branch conditions that are difficult to solve for fuzzers. Most of them can be reduced to solve the problem of magic byte conditions but, as in the case of Driller [24] and T-Fuzz [20], they employ techniques that are unnecessarily computationally expensive for this particular problem. The tools that provide solutions specifically designed to overcome magic byte comparisons are three: AFL-lafintel [2], VUzzer [22] and Steelix [16].

The last was selected for the implementation since it seemed to provide the best results while using the lightest instrumentation code. Its approach is based on instrumenting every comparison on

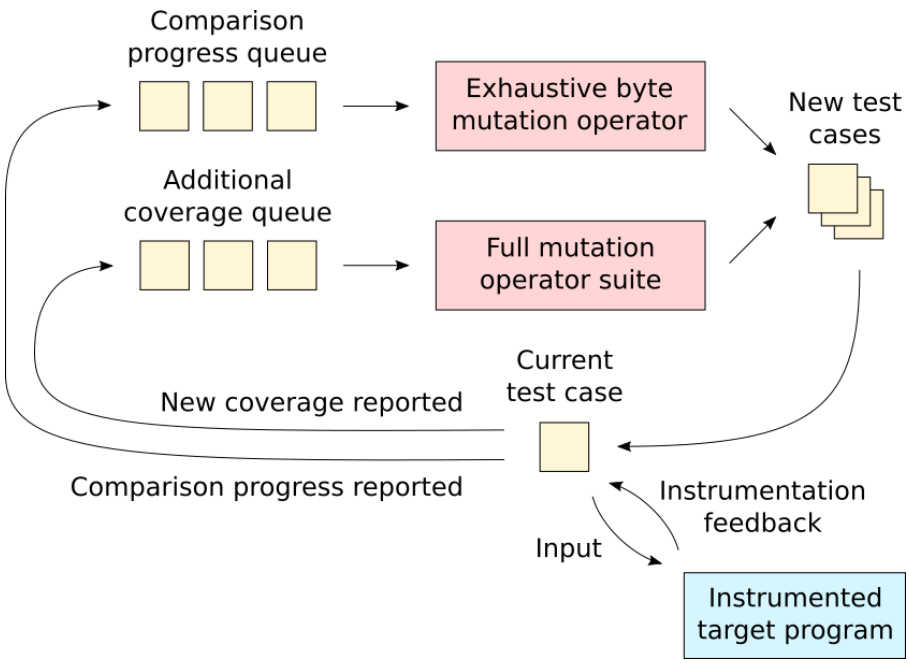

Figure 1: Scheme of the fuzzing process in AFL/QBDI

multiple bytes so that it appears to the fuzzer as a sequence of single byte comparisons, which can be more easily matched with random mutations. In addition, the matching is accelerated through a heuristic that deduces which is the next input byte that is being compared and mutates it exhaustively.

The implementation outlined by the authors of Steelix relies on modifying the program under test through static binary instrumentation, with the aid of a static analysis procedure. While this choice might bring several limitations with regard to the accuracy of the instrumentation, it has the advantage of providing better performance than an approach based on dynamic binary instrumentation. Therefore, one of the goals of the implementation was to adapt the heuristic in order to preserve the original performance as much as possible, despite the lower execution speeds imposed by the use of a different instrumentation technique.

Implementation and results

The implementation of the heuristic was performed in two parts: at first, new instrumentation callbacks were added in order to unroll all the comparisons performed by `CMP` and `TEST` instructions as required by Steelix; secondly, the heuristic that deduces the next byte to mutate was added to the fuzzer. In order to improve its efficiency at lower speeds, the algorithm proposed by Steelix was modified to separate the test cases that improve the coverage from the ones that generate new progress in an unrolled comparison. The latter were assigned a higher priority of being fuzzed since the Steelix heuristic makes them faster to process.

The structure of the fuzzing process in the final implementation is shown in Figure 1 and evolves as follows:

1. A test case is selected from the comparison progress queue or from the additional coverage queue; higher priority is given to the former.

2. If the test case comes from the coverage queue, it is mutated using all the operators available; if instead it comes from the comparison queue, it is mutated using the exhaustive byte mutation operator introduced by Steelix. In both cases, a series of new test cases is produced.

3. Each new test case is fed to the program under test whose instrumentation will report if the cumulative coverage was incremented or if an additional byte within a comparison was matched.

4. If the test case does not generate any of these events, it is discarded; otherwise, it is appended to the appropriate queue.

The results obtained when comparing the original Steelix implementation to AFL/QBDI show that, despite the lower execution speeds achievable using a dynamic binary instrumentation framework, it is still possible to obtain comparable performance. In addition, the overhead introduced by the instrumentation code is acceptable for longer runs since the heuristic is able to produce additional coverage that compensates for the time delay.

Outline and contribution

The rest of this document is structured as follows:

Chapter 1 begins with a section that provides an overview of fuzzing focusing on its application in the security context. It continues with a thorough description of American Fuzzy Lop, which was used as the basis for AFL/QBDI. The next section provides instead an analysis of the three main instrumentation techniques, their advantages and disadvantages. The chapter ends with a description of QBDI in order to clarify why it can provide better performance than other DBI frameworks when fuzzing.

Chapter 2 examines the main limitations of fuzzing, subdividing them in several categories. It then provides an overview of the literature on grey-box fuzzers, associating each tool with the issue it tries to solve.

Chapter 3 contains a complete explanation of the heuristic proposed by Steelix providing additional details on the algorithm. In addition, it examines the limitations to this approach that were found during the development phase.

Chapter 4 begins with a description of the pre-existing tool which was then improved as a result of this work. The rest of the chapter provides a thorough explanation of the implementation of the Steelix heuristic and the additional modifications that were performed in order to simplify the evaluation process. These include the ability to fuzz standalone binaries, removing the requirement of writing a library wrapper, and the re-implementation of the Dislocator library in order to integrate it with QBDI.

Chapter 5 presents the results obtained on the two selected targets, LAVA-M and ImageIO. The chapter begins with a description of the two pieces of software and of the experimental setting; it then continues with the examination of the results obtained, considering also the possibility of changing some parameters of the implementation.

This document is the result of my internship at Quarkslab where QBDI and the original version of AFL/QBDI were developed. I have contributed to the AFL/QBDI project through the review of the literature on grey-box fuzzing, the selection and the re-implementation of the Steelix heuristic. In addition, I have also designed and implemented the support for testing standalone executables and re-implemented the Dislocator library in AFL/QBDI, partially exploiting the original design proposed by AFL.

Chapter 1

Background

1.1 Fuzzing

This section is meant to provide an overview of fuzzing both from a historical and a theoretical point of view.

1.1.1 Historical overview

Fuzzing is an automated software testing technique which was first introduced in 1989. The study [18] that first applied it was meant to assess the reliability of UNIX utilities against possibly corrupted inputs. Examining its most primitive form, this technique consists in providing randomly generated inputs to the target software hoping to trigger errors like, for example, memory corruptions.

Over the course of time, the basic technique presented in that initial study was improved in several different directions and, nowadays, fuzzing has become probably the most used automated software testing technique. Indeed, many of the major software vendors are currently testing their products using fuzzing: one of the most famous examples is Microsoft SAGE, which reportedly found one third of all the bugs corrected during the development of Windows 7. [10] Another illustrious example is the Google OSS-Fuzz initiative [1], which provides continuous fuzzing for open source software, thanks to which more than 7000 bugs were found and fixed in software that is used every day by a considerable portion of the world population.

The success fuzzing has enjoyed in the recent years, as compared to other competing techniques, like symbolic execution, is probably due to the following characteristics: firstly, fuzzing is probably one of the easiest automatic testing methods to use, since it usually requires only a recompilation of the software under test and, in some cases, the writing of a simple library wrapper. Secondly, it is a very scalable technique, allowing to conduct successful testing campaigns even with limited resources. Lastly, since it is based on real executions of the software being tested, using concrete inputs, it does not produce false positives and it allows for the easy reproduction of the issues it has found, thus making the debugging process easier.

Moreover, additional success has been brought by the fact that the IT security community has taken a particular interest in this technique. The reason for this is that some of its characteristics adapt very well to the security evaluation setting: these are, among all others, simplicity of the setup and scalability. The motivation behind the first claim is that the possibility of being able to test software without having to understand it in depth and without being forced to write a large amount of code make fuzzing cost-effective. The motivation behind the second is that the possibility of conducting successful fuzzing campaigns even on small server machines or personal computers contributes to the resource requirement asymmetry that attackers can take advantage of when targeting a particular software.

In addition, another important factor that led to the widespread adoption of fuzzing in the security community is the lack of a valid competitor. In concrete, the technique that is more likely to represent an alternative to fuzzing is symbolic execution. This technique, however, despite giving better guarantees on paper, lacks one of the components that led to the success of fuzzing: it is not scalable, due to the path explosion problem, and it is not practical when applied to real life software,

requiring several compromises to be made [27]. In detail, the path explosion problem is generated by the fact that the number of possible paths in a program grows exponentially with the program size requiring an exponentially growing amount of resources to keep track of all the possible paths.

1.1.2 Classification

Given the success of fuzzing as a testing technique, the amount of different enhancements that have been introduced in order to increase its efficiency is considerable. As a consequence, it is useful to express the canonical classification [7] which subdivides fuzzers in smaller subgroups; this classification is based on three orthogonal directions.

Input reuse The first direction is relative to the method with which new inputs are generated. If the fuzzer always creates inputs from scratch, then it is considered to be generation-based; if instead it creates new inputs starting from existing ones, it is considered to be mutation-based.

This difference, as a consequence, changes also the way in which the fuzzing campaign is set up: indeed, in the case of mutation-based fuzzers only, some working sample inputs need to be provided by the user. It is important to notice that these initial inputs greatly influence the success of the campaign.

Input structure awareness The second direction deals with the knowledge the fuzzer has of the internal structure of the inputs it is generating. When the fuzzer is provided with a grammar or a protocol specification that describes how the input is composed, it is defined as smart; in the case in which no specification is provided, the fuzzer is instead defined as dumb.

While the awareness of the input structure may always seem superior to the blindness of a dumb fuzzer, it is important to notice that this first approach also has strong limitations. The issue, in the case of smart fuzzers, lies in how to interpret the grammar in such a way that allows the fuzzer to violate it: it is often the case that crashing inputs are malformed from a specification perspective, but are nonetheless interesting from a security one.

Program structure awareness The third, and last, direction considers instead the amount of knowledge the fuzzer has relatively to the internal structure of the program being tested. In this case, three categories have been specified:

Black-box This category contains fuzzers which do not have any particular knowledge about the internal structure of the target program and, indeed, treat it as a black-box. This is the most simple approach and it is usually employed when external observation is not possible, for example when targeting a program running on a proprietary appliance through a network connection.

White-box Fuzzers falling in this category rely on powerful program analysis techniques, as symbolic execution, in order to improve both the mutation and the input scheduling process. They can usually explore deeper paths within the program, but present the risk of introducing a significant execution overhead. In particular, the overhead may slow the fuzzing process to such a great extent that the use of a simple black-box fuzzer, which grants extremely high execution speed, may provide better results.

Grey-box This category falls in between the previous two since it comprehends fuzzers that have less insight on the internals of the target, but grant higher execution speeds. In order to achieve these results, grey-box fuzzers employ instrumentation techniques instead of the program analysis ones implemented in white-box fuzzers.

Within this classification, the subgroup that has enjoyed the greatest amount of success in the IT security community is that of grey-box, mutation-based dumb fuzzers, in which American Fuzzy Lop [25] is the most famous example. The amount of fuzzers presented in the literature that rely totally, or in large part, on such techniques is also significant. [24, 3, 22, 16, 6, 20]

The strongest points that can be made in favour of this approach are the following: firstly, mutation-based dumb fuzzers make the set up of the fuzzing campaign easier, since they do not require the user to express an input grammar and thus to be completely aware of all the inputs the target program can handle. Secondly, the grey-box approach has proven to be a good middle ground between the heaviness of program analysis and the limits imposed by an agnostic observation.

1.1.3 Binary targets

In addition to what was stated in the previous subsection, which holds in every scenario, there are some considerations that need to be made from an IT security perspective. The most important thing to be noted is that relying on the assumption that the source code for the target software is available can be considered reasonable only when a developer is testing his own software; unfortunately, when conducting an independent evaluation, this is likely not to be the case. As a consequence, approaches that allow the efficient fuzzing of closed source binaries are way more valuable in the security community as compared to those that require source code.

Most of the techniques used to aid fuzzing can usually be implemented both relying on source code or on the assembled binary, but their efficiency and accuracy can vary greatly between the two scenarios. This is the case, for example, for techniques that rely on instrumentation: when possessing the source code, it can be inserted through a compiler pass and thus optimized but, when targeting binary, it can be inserted either statically or dynamically but it always provides worse performance as compared to the compiled version.

For this reason, from a security perspective, it is more interesting to evaluate the quality of a certain technique targeting binary instead of source code so that the trade-offs in this first scenario are immediately clear.

1.2 Implementation details of AFL

This section is meant to give an overview of the implementation details of AFL that are significant in order to understand how the project presented in this document changed the upstream version. For this reason, some important parts of the original design, for example the algorithms used to evaluate and prioritize the test cases in the queue, will not be examined in depth since they were left untouched. A complete overview can be obtained reading the documentation available online. [26]

1.2.1 Execution trace

The first aspect of AFL that should be examined in depth in order to comprehend the rest of this document is the technique used to provide the fuzzer with information regarding the execution trace. This information is necessary since AFL is using genetic algorithms in order to maximize the edge coverage of the target software and thus it uses the information about the CFG edges that have been traversed as a quality metric for a generated test case.

The solution implemented in AFL to collect this information from the fuzzed program is based on a shared memory area, which is used as a bloom filter to record the edges traversed during the execution. When the target process starts, the shared memory area, already allocated and zeroed by AFL, is retrieved and then, each time an edge is encountered, the value in the corresponding cell is incremented by one.

The mapping between a certain CFG edge and the corresponding cell in the bloom filter is obtained as shown in Listing 1.1; `shared_mem` is the bloom filter itself and `cur_location` is assigned a random value that is either determined at compile time, when using source instrumentation, or it is derived from the basic block address, when targeting binaries. The important thing to note is that, in either case, the index corresponding to the current edge can be retrieved in a constant, and quite reduced, amount of time; the number of accesses to the bloom filter, instead, scales with the size of the program.

Once the execution of the target process terminates, the bloom filter is examined by the fuzzer, which observes if edges not seen before were traversed and if the same edges were traversed a significantly different amount of time as compared to previous executions; in this case, significantly different

Listing 1.1: AFL basic block instrumentation

```
cur_location = <BASIC_BLOCK_ID>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

is defined with a special logarithmic heuristic. Using this information, the fuzzer decides whether the test case is useful or it should be discarded.

As stated before, the main advantage of this implementation is that it grants good performance; its main drawback, instead, is that it can produce false negatives. In particular, multiple edges of the CFG can be mapped to the same bloom filter position, creating the possibility for a known edge to mask a new one, making it invisible to the fuzzer. In order to provide a good trade-off between performance and false negatives, the size of the bloom filter is set to 64 KB, hoping for its used part to fit in the processor cache.

1.2.2 Input handling

The second important feature of AFL that will be examined is the way in which the generated test cases are handled during the execution of the fuzzer.

Given that AFL is a mutation-based fuzzer, an initial set of valid inputs is provided by the user. These inputs are evaluated and then added to the test case queue. During the fuzzing process, generated test cases that produce additional coverage are also added to that same queue. In addition to storing significant test cases, the queue also collects a small group of them in what is called the preferred set. This set comprehends all the test cases that are likely to produce additional coverage when fuzzed while also providing the best performance according to metrics based on execution time and size.

When the fuzzer decides to start mutating a new test case, it selects the current head of the preferred set with a very high probability, even if selecting a non preferred one is still allowed. It is important to notice that, when a test case has been fuzzed or skipped, it is not removed from the queue, it is kept to be fuzzed in the next queue cycle, which starts when all the test cases in the queue have been examined; this means that the longer the queue is, the longer a single cycle will last.

1.2.3 Mutation operators

The third feature that is necessary to examine is how the fuzzer mutates the test cases in the queue to generate new ones.

Once the next test case to be fuzzed is selected from the queue, it is mutated exploiting a series of operators, which can be divided in two categories: deterministic and non-deterministic. The obvious difference is that the deterministic ones applied to the same test case produce always the same results, while the non-deterministic ones produce different results each time; a consequence of this is that deterministic operators are run on a certain test case only the first time it is fuzzed, while non-deterministic ones are run during every queue cycle.

The deterministic operators implemented in AFL are the following:

Bitflips This operator flips bits in groups of 1, 2, 4 or 8 in every position within the test case. It also collects information regarding markers that can be part of the file format, as “IHDR” for PNG.

Arithmetic operations This operator increments or decrements groups of 8, 16 and 32 bits interpreting them as integer values. These operations are repeated in every position of the input buffer.

Interesting values This operator replaces bytes in the input buffer with a series of predefined interesting values; these usually include minimum or maximum values for integers, null bytes and similar. Also in this case, the replacement is done in groups of 8, 16 or 32 bits in every position of the input buffer.

Dictionary operations This operator exploits a list of values specified by the user that are either used to overwrite portions of the input buffer or inserted in it; these values should be markers commonly found in the target format. The list is also expanded while fuzzing through a heuristic that extracts markers during the bitflip step.

The non-deterministic operators are instead only two, but they produce more extensive modifications:

Havoc This operator combines several of the deterministic operations presented before to produce a single test case, modifying the input buffer quite extensively. The positions in which the deterministic operations are applied are selected randomly and even the choice of the operations itself is random, despite the fact that some have a higher probability of being selected. Apart from those presented before, this operator performs also the insertion of random bytes at randomly selected positions and the deletion of random portions of the input buffer; these operations are needed to support the genetic-based approach to input generation.

Splicing The last operator listed, which is the last one needed to support the statement that AFL uses genetic algorithms in order to generate new inputs, is the one that splices together two of them. At a randomly selected location, two test cases are split and then the initial portion of the first is combined with the final portion of the other, effectively producing the result of a crossover operation.

1.3 Instrumentation

Given that the project described in this document relies on instrumentation, this section is meant to provide an overview of the three main instrumentation techniques stating their advantages and disadvantages.

1.3.1 Source instrumentation

The first option to instrument a binary is to use a source instrumentation technique. These techniques, as the name suggests, rely on the presence of source code and are normally implemented through a modification of the behavior of the compiler so that it produces additional assembly code at specific locations. This usually implies writing a compiler plug-in or, in the worst case, writing an assembly patcher.

In the case of AFL, both options are available: the project was launched with an assembly patcher which simply added code modifying the assembly source generated by GCC and Clang. As a later addition, an LLVM pass was added to the project so that the instrumentation process could happen inside the compiler. The advantage of this second approach is that optimization passes can be run after the instrumentation is inserted, thus obtaining faster instrumented code, as reported in the official documentation. [25]

1.3.2 Static binary instrumentation

Considering instead the case in which the source code is not available, one of the viable options is static binary instrumentation. This procedure involves the rewrite of some instructions in the binary file in order to make the program jump to the inserted code and then jump back. It is important to notice that this procedure is performed offline, before the execution, bringing both advantages and disadvantages.

The most important advantage is that the performance of the execution, even if not equal to that obtained with source instrumentation, is at least comparable. This is due to the fact that the overhead introduced by the instrumentation is almost only due to the callbacks themselves, no additional code is required apart from some jump instructions.

The disadvantages come instead in terms of consistency of the injection: first of all, the position in which the instrumentation needs to be inserted is established through static analysis. This is convenient for instruction based instrumentation, for example to instrument every comparison instruction,

Listing 1.2: Branch instruction that cannot be instrumented

```
1 mov rax, [rip + rbx]
2 jmp rax
```

but it is less reliable when considering instrumentation based on program structure, for example for every basic block or function: in this case, indeed, static analysis is subject to false positives and false negatives. An example of this are functions called only through indirect jumps that are notoriously difficult to identify statically.

An additional drawback is that it is not always possible to inject code in every position of the executable since some specific constructs, present in all binaries, prevent such an injection or make it really difficult to perform correctly. An example of this is presented in Listing 1.2, which shows that it is not possible to insert an instrumentation callback between line 1 and line 2 since the `rip`-relative memory access prevents the instruction from being moved around without patching its content. This kind of code is particularly common for switch statements in position independent executables.

An example of a framework that allows this kind of modification is Dyninst [5], which provides an API that allows to parse a binary and instrument it at specified locations with arbitrary code that has been previously compiled.

1.3.3 Dynamic binary instrumentation

The last alternative to instrument an executable is dynamic binary instrumentation. This technique, as suggested by its name, is able to target compiled binaries, as it happens for static binary instrumentation, but it grants higher reliability at the expense of execution speed.

The main difference when compared with static instrumentation is that no patching is required before the execution and thus no modification of the original binary is necessary. A DBI framework controls the execution of the program through the following procedure: first, it disassembles the next basic block, starting from the current position, and then it assembles the patched version in memory just-in-time. The patching includes the insertion of the callbacks requested by the user, the modifications necessary to make the control of the execution return to the framework and other fixes required by the relocation. After that, the state of the execution of the target program is restored and the assembled block is executed; when the control is returned to the framework, the execution state is saved and the cycle restarts.

This mechanism grants a precise view on the direction the execution is taking, making it easy to distinguish, for example, the next basic block that will be executed, even through an indirect call or jump. As a consequence, while the patching by instruction type is as reliable as the one for static instrumentation, the instrumentation based on layout of the CFG is significantly more reliable. In addition, a DBI framework will never incur in the problem presented with Listing 1.2, since assembling just-in-time allows to insert instrumentation at truly arbitrary locations without using excessively complex heuristics. One last advantage is that DBI frameworks can instrument every piece of code that is loaded in the executable memory, including shared libraries; in the case of static binary instrumentation, each shared library would have to be patched singularly.

The price to pay for this added precision in the instrumentation is worse performance, which is mostly due to the fact that every new basic block requires a context switch between the target program and the DBI framework; this is a costly operation since it requires to back up all the CPU registers in memory including AVX and floating point registers, which are slow to manipulate.

The most used, and best performing, DBI framework is probably PIN [17], developed by Intel and currently supporting only x86 and x86_64 architectures. Other famous examples are Valgrind [19], which probably supports the largest number of architectures, and DynamoRIO [4], which grants better performance than Valgrind, but only allows assembly level manipulation; DynamoRIO, as compared to PIN, also supports ARM.

The DBI framework used for the project presented in this document is still in its development phase and will be described thoroughly in Section 1.4.

1.4 QBDI

The purpose of this section is to thoroughly examine the dynamic binary instrumentation framework that was used for the implementation of this project. This analysis is needed in order to give a clear overview of the foundations the project relies on, thus helping to understand why certain design choices were made. Most of the information presented is taken from the presentation [12] the authors of QBDI gave in occasion of the first public release of the software.

1.4.1 Overview

As already discussed in Section 1.3, statically inserting code in a binary file can be dangerous since it is not always possible, not always complete and it can lead to a corruption of the original code. The only solution to these problems is to insert the instrumentation at runtime, modifying the code while it is being executed.

In concrete, code is examined only until the end of the next basic block to be executed, thus until the first branching instruction starting from the current instruction pointer. As a consequence, a DBI framework modifies only one basic block at a time, alternating the execution of the guest program with the operations needed to instrument the binary code. This method gives the certainty that all the basic blocks that will be executed are recognized correctly.

Given that most of the problems related to static binary instrumentation are generated by the fact that patches are inserted in place, modifying the original code, in DBIs another approach is taken: each basic block is moved to a different position in memory, where it can be easily modified. It is important to note that this process, defined as relocation, requires extensive modifications to the basic blocks: all the relative references to code and data need to be corrected and the control of the execution needs to be returned to the DBI framework at the end of each basic block.

1.4.2 Execution cycle

Going further in detail in the implementation of QBDI, a precise cycle can be established when a basic block that has never been observed before is encountered. It is important to note that, for every step in this cycle, it is evident how the goal of building a multi-architecture framework strongly influenced the whole design of the project.

The first step is to disassemble the binary code until the first branch instruction in order to retrieve and decode the whole basic block. QBDI completes this step relying on the LLVM framework [15], which provides an intermediate representation for instructions, called MC. This representation is architecture independent and can be easily reassembled once modified as desired. In this way, QBDI offloads the task of assembling and disassembling to an external and extremely stable library that already offers support for multiple architectures.

The second step is to patch the MC representation in order to allow the relocation of the code. This task can be considerably complex and, more importantly, it tends to be dependent on the architecture that is being used. In order to avoid duplicating patches, QBDI implements a component, called patch engine, that interprets a special domain specific language which, in turn, allows to write architecture independent patches, when possible. The patching process also allows for the insertion of calls to the callbacks registered by the user, allowing it to get control of the execution at any location within the binary.

The third step is to assemble the MC representation that was patched in the previous step. In order to do this, as stated before, QBDI relies again on LLVM, which provides a JIT assembler that is able to assemble, in memory, the instructions expressed in the MC intermediate representation. This allows, again, to offload a difficult task to an external library allowing QBDI to support a large number of architectures with reduced effort.

The last step is simply to execute the basic block that was just assembled relying on the fact that the branch instruction terminating the block will have been substituted with a jump that returns the control to the DBI framework. It is important to note that basic blocks that have already been processed are also cached so that, when encountered again, they can be reused directly.

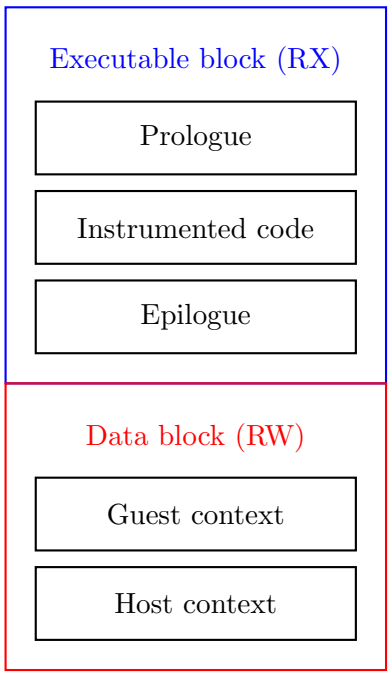

Figure 1.1: ExecBlock structure used in QBDI

As a variation on this loop, QBDI also gives the possibility to the user to specify a series of address ranges that should not be run through the DBI, but instead natively on the original code. This is quite useful if only a particular area of the binary is being targeted and the performance of the execution is critical. The control of the execution is usually recovered through heuristics, such as substituting, on the stack, the return address of the function that is not being instrumented with the entry point of the DBI.

1.4.3 Architecture

From an architectural point of view, it is worth examining how QBDI manages the assembled code in order to keep the execution control activities as transparent as possible for the guest program.

First of all, it is important to note that the DBI framework, also named host, and the guest program are supposed to run in two different contexts; this is necessary since the guest should be unaware of the code relocation process and the host should be able to manage the guest without damaging its state. These two contexts, however, both reside within the same process and thus there is no feature in the kernel or in the CPU that helps separating them. It is thus necessary to manually switch between them.

In order to perform this context switch, QBDI exploits a structure called ExecBlock, shown in Figure 1.1, which is composed by two pages: the first one has read and execution permissions, while the second one has read and write permissions. The first page is used to contain the assembled code, while the second one is used to backup the state of the host during the execution of the guest and to store host data necessary to handle the instrumentation code. In order to save space, a single ExecBlock can be used to store several basic blocks and the relative data; the multiplexing between these blocks is then managed through a common prologue and a common epilogue that run the appropriate code through a selector, specified by the host.

The introduction of this structure was necessary since the authors wanted the DBI to behave correctly even when in the presence of the W^X protection mechanism, which is becoming the norm on macOS, and wanted to support ARM architectures: these architectures, indeed, cannot reference data that is too far away from the current instruction pointer due to how memory accesses are handled.

Chapter 2

Problem statement

2.1 Challenges in fuzzing

As every other automated testing technique, fuzzing is also affected by some intrinsic limitations, which are difficult to overcome without precise techniques targeting them.

In order to illustrate such limitations, it is useful to refer to the model presented in the paper about Driller [4] which describes software as a set of communicating "compartments": the authors argue that branch conditions that check for a very restricted set of values in a specific input split the application into two different compartments. As an example, checking for a specific byte sequence in a header positions the code that is run when the sequence is matched in a different compartment as compared to the one that is run when the sequence is not matched.

This model is useful to describe the kind of exploration fuzzing is capable of: it is able to thoroughly explore the code within the same compartment, covering all the branches protected by conditions that are easy to guess providing random inputs; this is the case, for example, of parsing code, which translates the input in a set of data structures that is then used by the rest of the program.

The point in which fuzzing struggles the most, however, is making the execution flow from one compartment to another. This is simply due to the fact that conditions that have a large search space and a really small accepted range are nearly impossible to guess only randomly mutating the input test case. In the case of generation-based fuzzers, this difficulty can be overcome if those same branch conditions are embedded in the grammar definition but, in the case of mutation-based fuzzers, the only way to explore a certain container is to provide a seed input that traverses it; containers that are not touched by the seed inputs are really unlikely to be explored after.

Examining in detail the branch conditions that usually block grey-box fuzzing, it is possible to group them in different categories and try to define why each one of them represents an obstacle to the exploration process.

Magic bytes The first, narrow, category that can be identified is related to magic byte sequences. This particular category is quite relevant in the context of fuzzing since it is the main problem that this method faces while testing parsers, one of the targets it is most commonly used for.

In detail, these branch conditions check that a series of adjacent bytes in the input corresponds to a precise sequence and, if this is not the case, make the program take another path. This extends to both sequences that are required to be present in a certain position and to sequences that can be placed anywhere in the input, which are commonly defined as tokens.

Common grey-box fuzzers, like AFL, are unable to overcome this kind of condition since they do not have any notion of what is composing a particular branch condition, they receive feedback only regarding whether it is met or not. In the case of white-box fuzzers, however, conditions themselves may be observable and thus a mutation of the input can be used to satisfy them quite easily.

Complex conditions The second, and probably largest, category comprehends instead conditions that include complex aggregations of the input bytes, or even metadata regarding them, like the length

of the input. Having such conditions in the program does not constitute a problem in itself, but it becomes an issue when the search space is large and the accepted range is really small.

As for the previous case, grey-box fuzzers have no clue regarding how to satisfy the condition and can only keep generating random samples hoping to hit the accepted range. Also in this case, white-box fuzzers may be able to mutate the input to satisfy the conditions, but only in the case they are easy enough to be solved with a SMT solver.

Unsolvable conditions The last category comprehends instead all the pathological conditions that are impossible to solve with a SMT solver and are really difficult to guess just through random generation. Concrete examples are conditions that construct complex non-linear relations between the input bytes; an extreme case is a branch condition requiring that the cryptographic hash of the input is equal to a precise value.

In this case, not even white-box fuzzers can do anything to solve the condition, the only hope to explore those branches is that the user is providing a sample already satisfying them and that the related bytes are not modified during the mutations.

2.2 Proposed solutions

Several attempts have been made in order to solve the challenges related to difficult branch conditions for grey-box fuzzers described in Section 2.1. Being able to solve these issues would allow to enjoy the advantages of white-box fuzzing, illustrated in Section 1.1, while obtaining higher execution speeds and better scalability.

Within the literature, there are several examples of grey-box fuzzers that have been integrated using a technique, or a heuristic, aimed at overcoming one or more of the challenges presented before. These projects are going to be presented in the following paragraphs, roughly in chronological order.

The fuzzer which is the most relevant for this study, Steelix [16], will be dealt with in Chapter 3, and thus is not reported in this list.

Driller This tool was developed with the purpose of being used in the Cyber Grand Challenge competition created by DARPA to push the development of automated detection and automated patching tools for software. The goal of the developers was to combine the benefits of two automated testing techniques, symbolic execution and fuzzing, in order to get the best of both worlds.

The main idea behind the tool is to employ a normal grey-box fuzzer, plain AFL in this case, and use it to explore the program until it cannot find any new basic blocks. The common reason for the exploration to be blocked in this phase is that difficult branch conditions protect the remaining part of the program. At this point, a symbolic execution engine is used to create new test cases: it first examines the queue and records all the branches that have never been explored during the fuzzing phase; after that, it generates new inputs simply by negating the conditions protecting those branches.

This heuristic allows Driller to overcome branch conditions that fall in the first and the second category of challenges. However, it has the significant problem that the use of a symbolic execution engine is quite expensive in terms of resources since it is subject to the path explosion problem. [24]

VUzzer This tool was developed with the goal of trying to prove that, through the use of heuristics, it is possible to overcome a good part of the problems for which symbolic execution was previously employed. This substitution should help improving the overall performance of a fuzzer due to the lower amount of resources required.

In order to achieve this goal, several heuristics have been introduced: some are targeted at improving the management of the queue containing the test cases to be fuzzed, like basic block weighting, and some at overcoming the problem of difficult branch conditions. The most relevant, given the breakdown made in Section 2.1, is the one targeting magic bytes: this heuristic exploits dynamic taint tracking for `cmp` and `lea` instructions to point out sequences of input bytes that taint the same operand in every execution of the seed inputs; when this happens, it considers those sequences as magic bytes. Another relevant heuristic is the one that tries to target markers, for example section separators, that

may be positioned at different offsets in the input: for every `cmp` instruction that compares a certain sequence of input bytes with an immediate, VUzzer places that immediate at the offset which tainted the register operand of the target instruction during the following mutations.

These two heuristics help to solve the first challenge presented, the one related to magic bytes. However, the first heuristic relies on the fact that the user will provide a diverse enough set of inputs, possibly introducing false positives, and the second one does not work when only register operands are used, so the problem is not completely solved. [22]

T-Fuzz This tool takes a completely different approach when compared to the ones that were presented up to now. Instead of trying to overcome difficult branch conditions through heuristics or improvements to its fuzzer component, it patches them in the binary.

In detail, the tool is composed by two components: a grey-box fuzzer, which is just plain AFL, and a program transformer, which is able to analyze and patch the program. When the fuzzer gets stuck, the program transformer examines its queue and isolates all the branches that it was not possible to take. At that point, it patches their conditions one by one, creating several different patched programs, and then it restarts the fuzzer.

Such a procedure allows this fuzzer to virtually overcome all the types of challenges previously illustrated without recurring to heavyweight techniques, but it introduces several drawbacks. First of all, it breaks the reproducibility guarantee that fuzzers usually have since it is not possible, in the general case, to produce an input that is able to reach the location of a crash; the authors proposed a symbolic execution based technique in order to do so, but it does not always work. In addition, producing several different programs which need to be fuzzed again introduces a problem similar to path explosion in symbolic execution, which was reported in the paper. [20]

Angora This tool introduces several improvements, some of which change the architecture of the fuzzer itself, like the queue based on target branches and not on inputs to be fuzzed. However, considering the challenges previously presented, the most important heuristic introduced is the one that cracks branch conditions combining byte-level taint tracking and the gradient descendent technique.

In detail, Angora is able to keep track of all the bytes that contribute to the value of an operand in a certain branch condition. Using that information, it treats the condition as a black-box and uses the gradient descendent technique to solve it; when several different conditions are present for the same branch, Angora is able to separate them.

The technique presented is able to solve both the first two challenges illustrated before, but it is not useful against conditions that cannot be cracked even with a SAT solver. In addition, it presents another criticality: the possibility of applying the gradient descendent technique when targeting a binary is likely to be impossible or considerably difficult, since the logic of a particular condition is split over several instructions, probably requiring static analysis to reconstruct it. [6]

Chapter 3

Steelix approach

3.1 Approach overview

The project presented in this document heavily relies on the ideas that supported the development of Steelix [16]. For this reason, this section is meant to give an overview of the approach the authors of that tool took for the development of their heuristic.

The main goal targeted by Steelix is to create a fuzzer that is able to overcome the problem of magic byte sequences and, at the same time, keep the execution speeds as high as possible.

In order to reach this goal, the basic idea is to transform each condition involving magic byte sequences into a series of conditions involving single bytes, effectively unrolling the comparison. This technique, which was first presented by the authors of AFL-lafintel [2], transforms a single condition with $2^{8 \cdot l}$ possibilities and 1 accepted value into l conditions with 2^8 possibilities and 1 accepted value, where l is the size of the magic sequence in bytes. Since the fuzzer can keep track of each matched comparison, it can crack the general condition byte by byte; this process is easier since the search space is reduced. An example of this code modification is presented in Figure 3.1.

In addition to this, Steelix introduces a form of guided mutation meant to speed up the process of cracking magic byte sequences: it uses a heuristic to understand which is supposed to be the next input byte to be matched in the targeted comparison and, based on that, it executes a complete mutation, exploring all the possible 256 values, of that byte.

In order to avoid compromising the performance of the fuzzer, the individuation of the next byte to mutate is not made through dynamic taint analysis, as one would expect. It is instead achieved simply supposing that the next byte to be matched is either the one following, or the one preceding, the byte that was last modified when the last comparison progress was reported. This information is already available to the fuzzer, so no complex communication is needed in order to report comparison progresses, it is sufficient to report that one occurred. A more thorough description of the implementation is present in Section 4.2 and a complete example that explains how the heuristic works can be found in the original paper. [16]

The original implementation subdivides the fuzzing process in three main steps, which are illustrated in detail in the following paragraphs.

Static analysis The first step is based completely on static analysis which is performed using IDAPython [11] as an analysis framework.

As far as the Steelix heuristic is concerned, all the comparison instructions present in the binary, in particular all the variations of `cmp` and `test`, are located and recorded together with information about their operands. The recorded locations are then used as instrumentation targets in the next step.

The authors introduce also two arbitrary heuristics in order to discard some of the instrumentation locations and thus improve the overall performance of the binary. The first rule forbids the instrumentation of single byte comparisons, while the second one forbids the instrumentation of comparisons on function return values.

In addition to the information necessary to properly handle the heuristic, a list of all the branch

```
if (input == 0xabad1dea) {
  /* buggy code */
} else {
  /* good code */
}
```

(a) Before the compiler pass

```
if (input >> 24 == 0xab) {
  if ((input >> 16) & 0xff == 0xad) {
    if ((input >> 8) & 0xff == 0x1d) {
      if (input & 0xff == 0xea) {
        /* buggy code */
        goto end;
      }
    }
  }
}
/* good code */
end:
```

(b) After the compiler pass

Figure 3.1: AFL-lafintel comparison unrolling

instructions is also collected in this step. This is necessary since, in order to produce the AFL-style execution trace that is necessary for the fuzzer to work, those locations need to be instrumented as well.

Instrumentation The second step consists in the instrumentation of the target binary, exploiting Dyninst [5] as an instrumentation framework.

The instrumentation is performed on the locations selected during the static analysis step and consists of AFL-style instrumentation for branch instructions and unrolling instrumentation for comparison instructions.

In detail, the unrolling instrumentation reports to the fuzzer how many bytes were matched within the target comparison, using the same shared memory area that is used to report the AFL execution trace. The unrolling always starts from one side of the operands and taints a new byte in the shared memory area for each byte that is equal for both of them. Unfortunately, no further information is provided regarding how the comparison information is reported to the fuzzer within the memory area.

The instrumentation for branch instructions is instead exactly the same as the one performed by AFL in QEMU mode, exploiting the same mechanism based on basic block identifiers, as described in Section 1.2.

Fuzzing loop The last step in processing a target binary is, as one would expect, the execution of the fuzzer. In the case of Steelix, this piece of software is a modified version of AFL, which has been patched to support the heuristic.

In particular, the patches allow the fuzzer to distinguish between comparison progresses and branch hits reported by the instrumentation which was generated in the previous steps. In both cases, the sample is appended to the fuzzing queue but, when dealing with a comparison progress, the last mutated byte is also recorded in order to be used in the mutation phase.

A different behavior is also introduced when processing a new sample extracted from the queue, since the ones that generated comparison progresses need to be dealt with in a different way. The important point is that, differently from what happens in AFL-lafintel, comparison progress inputs undergo only local exhaustive mutation as described previously, they are not fuzzed using all the

mutators available in AFL; this, instead, is done only if the local exhaustive mutation fails in generating any kind of additional progress. Moreover, the queue handling is also modified so that comparison progress test cases are eliminated from the queue when they generate additional progresses.

Both the fact that comparison progress test cases are fuzzed using all the mutators only when necessary and the fact that they are removed from the queue once they generate additional progress helps in mitigating one of the largest problems present in AFL-lafintel, defined as queue flooding. This problem consists in the fact that comparison progress test cases that are inserted in the queue and never removed will quickly become a considerable proportion of the whole queue, delaying the processing of test cases with additional coverage; these are instead likely to produce results that are more useful to the general process. In addition to that, the fact that comparison progress test cases undergo a full fuzzing cycle, while instead their sole purpose is to help cracking a single comparison instruction, makes the fuzzer unnecessarily waste a considerable amount of time on them. Both these problems are addressed through the patches that have been added to AFL.

3.2 Limitations

Both while reading the paper and while creating the alternative implementation of the Steelix heuristic presented in this document, several limitations of this technique became evident. Since these limitations guided the design of the reimplementation and of the improvements that were made on the original version, they are analyzed one by one in this section.

3.2.1 Static binary instrumentation

As already discussed in Section 1.3, static binary instrumentation has one main intrinsic limitation: it does not always allow to instrument every location and, if the instrumentation is performed even when not possible, it can break the binary that is being instrumented. In concrete, this can reflect into a corrupted binary that may behave differently from the original. In the worst case scenario, the binary can even be corrupted in such a way that it does not crash and thus the user cannot realize that the behavior is not consistent.

The only way around this problem is to use dynamic binary instrumentation instead of the static version, so that callbacks can be inserted in every position. Indeed, no actual modification of the original code is performed: the framework that is managing the execution will take care of executing the callbacks on its own.

The only drawback brought along with this change is in terms of execution speed, which may be degraded; understanding the magnitude of this degradation, and thus the feasibility of this change, was also one of the goals of this study.

3.2.2 Static analysis

In addition to the previous limitation, the position of the instrumentation, when patching statically, has to be necessarily determined before the execution of the binary; this means that static analysis has to be used, introducing a new limitation.

Static analysis, given that it cannot execute the binary, may produce incomplete results when trying to extract specific features of the code, such as a complete CFG. It is sufficient for the analysis to encounter an indirect jump whose target is computed at runtime, as it happens for C++ virtual methods, for it to fail. This is a well known problem and, while it never happens targeting single instructions, it is common when trying to isolate all the basic blocks in a program, as it is required by AFL-style instrumentation.

Again, the only way around this issue is to use dynamic binary instrumentation, so that locations to be instrumented can be determined at runtime. This, in turn, allows the dynamic binary instrumentation framework to extract the control flow features at runtime, allowing to correctly report every basic block. The drawback is obviously still related to speed.

3.2.3 Taint analysis

The previous issues were generated mostly by how the authors decided to implement the heuristic, more than by the heuristic itself. This issue, instead, is created by how taint analysis is implemented to make the Steelix heuristic work.

In particular, no canonical taint tracking mechanism is put in place since the authors argue that every other solution present in the literature is too heavyweight for a fuzzer. As a consequence, they simulate the result of taint tracking making the following assumption: when a comparison progress is reported, they suppose that the byte that was last modified in the input generated it and that its neighbors are candidates to be part of the same comparison.

As it is evident, this taint tracking heuristic relies on the concept of last modified byte. However, despite the fact that the fuzzer is always the one mutating the input, it may be impossible to keep track of which byte is the last modified one. This is unfortunately the case for all the mutation operators that modify more than one byte at a time, since it is impossible to know which is the one that generated the comparison progress. Even worse, when using operators based on genetic algorithms, such as those that insert or delete bytes, it is not possible to make any assumption on the origin of the comparison progress, since a large part of the buffer is shifted. The authors of Steelix do not deal with this problem in the paper and thus it is unclear which is the behavior of their implementation when a comparison progress is reported after a multibyte mutator was run.

When patching AFL, this issue is significantly disruptive since it breaks all the non-deterministic operators listed in Section 1.2 and it does not allow to import test cases between different instances. There are two possible solutions to this problem: one of them is to use actual dynamic taint tracking, but it would go against the assumptions made by the authors regarding execution speed, while the other is to disable the heuristic for unsupported operators. In the reimplementation presented in this document, the second option was selected since shifting to dynamic taint tracking just to crack magic byte sequences would have probably been unwise in terms of performance.

3.2.4 Function return values

This issue is related to one of the rules used to prohibit instrumentation for certain comparisons in order to limit the amount of instrumented locations. The problematic rule is the one prohibiting instrumentation of comparisons on function return values. This rule, while making sense in the context of the example presented in the paper, introduces one main issue: it breaks simple parsing functions, such as `atoi`, and completely prevents the heuristic from working on LAVA-M binaries, since the input is always parsed using the `lava_get` function, as explained in Section 5.1.1.

Given the consequences presented above and the fact that the addition of this rule would have not been straightforward when using dynamic binary instrumentation, it was considered preferable to avoid applying it. Possible solutions would have been to introduce taint tracking during execution just for function return values or to add a static analysis step to generate a blacklist, taking the risk of it being incomplete.

Chapter 4

Implementation

4.1 Pre-existing tool

This section is meant to describe the structure of the pre-existing project, called AFL/QBDI, on which the work described in this document is built.

The original version of AFL/QBDI is really similar to the AFL project: in fact, it uses the upstream version of that software for the fuzzer executable, the only difference is in how the instrumentation is inserted in the binary. As a consequence of this similarity, the following paragraphs will only deal with the differences regarding the instrumentation process, the rest of the fuzzer can be assumed to be exactly the same as AFL, which was already described in Section 1.2.

4.1.1 Instrumentation technique

The most important difference that distinguishes AFL/QBDI from AFL consists in how the instrumentation is performed: AFL can perform source instrumentation, when the source code is available, or instrument binaries through a patched version of QEMU; AFL/QBDI, instead, exploits QBDI [21], a dynamic binary instrumentation framework, in order to achieve the same result. Despite this difference, however, the behavior of the instrumentation is fully compatible with AFL.

The consequence of this design choice is already discussed thoroughly in Section 1.3, but an additional note has to be made. While the performance obtained using QBDI is quite distant from what can be obtained with source instrumentation, it represents a good improvement when comparing it with what is obtainable with QEMU. This is possible thanks to the fact that, while QBDI has a structure that is similar to PIN, and thus faster than QEMU on a single run, it also supports the same pre-caching mechanism that was added to QEMU by the authors of AFL; this allows to get the best of both approaches. The pre-caching mechanism will be described in detail later in this section.

4.1.2 Library targets

The second important difference is that, from a user interface perspective, AFL/QBDI was developed to target library functions instead of complete binary programs. This observation stems from the fact that, in order to insert the instrumentation required, it is necessary to write a small library wrapper that takes care of setting up the environment: in particular, it starts the dynamic binary instrumentation framework and handles the translation of the input provided by the fuzzer into the arguments that the target function is supposed to receive.

This design choice, however, compromises the possibility to compare AFL/QBDI with other grey-box fuzzers which, in general, respect the interface AFL exposes. As a consequence, the possibility to target binary programs was added as part of the work to perform the evaluation presented in Chapter 5.

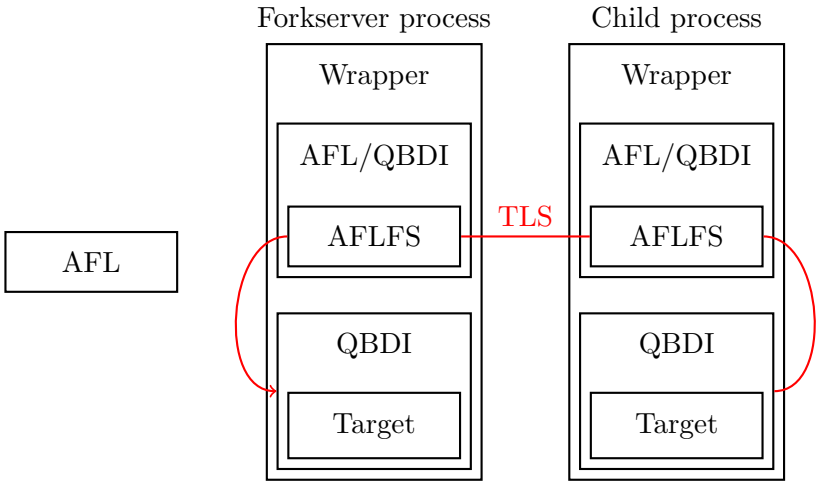

Figure 4.1: Basic block pre-caching

4.1.3 Architecture

The last important difference that AFL/QBDI possesses is constituted by how the handling of the communication between the instrumentation code and the fuzzer happens. This is mostly due to the use of QBDI to perform the instrumentation, which required some changes in the original design taken from AFL.

The most important thing to note is that every communication between the fuzzer and the library wrapper is handled through a shared library called AFLFS. This library is also used to support both the fork server and the persistency functionalities, which allow to achieve better performance in specific scenarios.

In detail, a fork server is used to avoid forking and then `exec`-ing each new library wrapper process directly from the fuzzer process; indeed, this is an expensive operation because it does not take advantage of the memory copy-on-write features implemented in modern kernels. In practice, this technique consists in having a copy of the library wrapper process, called indeed fork server, that is used to fork the wrapper processes necessary for fuzzing, thus eliminating the need for the `exec` system call. In AFL/QBDI, this mechanism is even more important since it allows to exploit the pre-caching mechanism present in QBDI: when a new basic block is found and assembled in one of the forked child processes, the location of the block is sent back to the fork server which passes it through to its QBDI instance. This allows QBDI to assemble that same basic block in the fork server too, so that every new forked child will benefit from that new cache entry. An overview of the architecture on a process level and of the information flow for the pre-caching process, marked as TLS, is presented in Figure 4.1.

The support for persistency, instead, consists in reusing the same library wrapper process for multiple calls to the target function; this works only for stateless libraries since they do not change the global state of the process. The advantage of this technique is that it allows to fork a new process only once for a fixed number of executions and not every time; this is useful mostly on macOS, where the time taken for a `fork` system call is considerably higher than Linux, where the difference is only noticeable for fast targets. [9]

The advantage of supporting these two features through an external library is that it allows to hide all the complexity from the user that is writing the library wrapper.

4.1.4 Comparison unrolling

In addition to all that was listed before, a first attempt to comparison unrolling had already been made in the original implementation. It consisted in a simple callback that tried to simulate the effects that the instrumentation shown for AFL-lafintel in Figure 3.1 would have had on the execution trace transmitted to the fuzzer. It consisted simply in reporting a new branch hit for every new byte matched in a single comparison.

The problem with this implementation was that, while working on small examples, it did not scale on larger programs. First of all, queuing comparison progress test cases as inputs with additional coverage leads to the queue flooding effect that was discussed in Section 3.1; this is a considerable loss of time due to the fact that all mutation operators are used when a single targeted modification is usually sufficient to crack a condition. In addition, reporting comparison progresses on the same shared memory map used for coverage information makes collisions, and thus masking interesting branches, more likely.

4.2 Steelix integration

This section is meant to illustrate the modifications that were made, starting from the original tool, in order to integrate the Steelix heuristic in it. These comprehend changes both in the instrumentation code and in the fuzzer, since the heuristic supposes that the fuzzer is able to distinguish comparison progresses and coverage progresses. This last point implied a complete rewrite of the comparison unrolling instrumentation present in the original version.

Each of the following paragraphs presents a, more or less, self contained modification made on the instrumentation code or on the fuzzer.

4.2.1 Comparison progress reporting

The first thing that was necessary to change was how comparison progresses were reported to the fuzzer. The previous method of mixing them with the execution trace was inefficient since it could create collisions in the shared memory map used to record the trace and would have required an additional mean of communication in order to distinguish between comparison progresses and coverage progresses.

In order to solve both problems at once, it was decided simply to duplicate the size of the memory shared between the fuzzer process and the wrapper process: this allowed to have enough room for an additional bloom filter to be used for comparisons exclusively. A discussion on whether the fact that the size of the map was doubled is significant from a performance perspective is discussed in depth in Sections 5.3.2 and 5.4.2.

In order to implement this modification, both the instrumentation and the fuzzer code were modified making them, respectively, write and read on the right portion of the shared memory area. An additional modification was introduced in the source code of the fuzzer program in order to allow for the separate processing of comparison progress inputs, distinguishing them from coverage progress ones.

An important technical limitation that needs to be highlighted is that QBDI, while allowing to access the runtime value of the operands of a certain instruction, it does not currently allow to access it when this value is obtained through a memory access. Instructions with such accesses are less common in optimized code, but they are still present; when encountered, they can only be ignored.

4.2.2 Queue handling

The second modification that was implemented is related to how test cases are processed during a single run. It is important to note that this modification slightly deviates from what the Steelix heuristic described, hoping to obtain better performance.

In detail, instead of queuing comparison progresses in the main queue after marking them, they are appended to a separate queue, which can be handled separately from the main one. This comparison progress queue is then always processed before the main one, so that comparison progresses discovered while fuzzing a particular test case are processed right after it.

This change was made since, appending the test cases to the main queue as in the original heuristic, the time required to crack a single condition is very long. This is due to the fact that, given a condition formed by four bytes, it is necessary to process all the test cases already in the queue four times before being able to reach the target comparison progresses. This happened despite the fact that processing a single comparison progress takes way less time than using all the mutation operators on a single

test case. It was thus decided to fuzz comparison progresses, that are reduced in number and fast to process, before extracting a new test case from the main queue.

The priority given to comparison progresses is not absolute, there is a certain probability that the fuzzer will jump directly to the main queue. This probability was introduced in order to avoid starvation in edge scenarios; several probability values were tested and the results can be seen in Sections 5.3.3 and 5.4.3.

It is important to remember that, while coverage progress test cases are never removed from the main queue, the heuristic states that comparison progresses should be removed if they produce an additional comparison progress or additional coverage. This same rule is applied to the double queue architecture presented above: when a comparison progress is processed, it is always removed from the comparison progress queue; if it produces an additional progress, it is simply discarded, if instead it does not produce any progress, it is appended to the main queue. The behavior of queueing sterile comparison progress test cases to the main queue can, in the worst case scenario, lead to the same queue flooding problem present in AFL-lafintel, but this also holds for what Steelix does and this is, in practice, quite unlikely.

Resume support was also implemented for the comparison progress queue so that stopping and then restarting the fuzzer does not lead to a loss of the current queue state.

4.2.3 Steelix mutation operator

The last change that was introduced is related to the implementation of the mutation operator required by the heuristic. This operator is able to produce test cases with every possible value of a specified target byte. Its behavior is equivalent to the one presented for Steelix.

In detail, when a comparison progress is produced, the last modified byte is recorded, if this is supported by the mutation operator that is being used. The position of this byte is important since one of its neighbors will become the target for the Steelix mutation operator when the comparison progress will be processed.

Which of the two neighbors of the last modified byte will be mutated is determined through the use of a mechanism that deduces the direction in which the magic byte sequence is encoded in the input. If the comparison progress is generated while using an AFL mutation operator, no direction can be deduced and thus both neighbors are targeted. If instead the comparison progress is generated while using the Steelix mutation operator, it is possible to know which of the two neighbors generated the new match; using this information, the direction of the magic byte sequence can be deduced and recorded. Once the direction is known, only the neighbor to the left or the one to the right needs to be fully mutated, allowing to process the test case faster.

Figure 4.2 illustrates an example of how the direction deduction technique works. In the graphics, the squares represent the input bytes, while the circles denote the bytes that are being targeted in the current step. The cells containing an “X” are the ones that contain a portion of the magic sequence that is yet to be matched; the sequence is “MAZE” in the example.

4.3 Other improvements

This section is meant to give an overview of the other improvements that were introduced in the original design, but that are unrelated with the Steelix heuristic presented in Section 3.1.

4.3.1 Support for executables

As stated in Section 4.2, AFL/QBDI was designed with the goal of testing a function belonging to a library; as a consequence, it provides the user with an API that can be used to write library wrappers, but it is not compatible with the interface offered by the upstream version of AFL.

The issue with the lack of AFL compatibility is that a good part of the state of the art fuzzers provide an AFL-like interface and test suites for fuzzers, such as LAVA-M which will be presented in Section 5.1.1, are designed around it. As a consequence, in order to aid the evaluation process, it was necessary to add the support for the instrumentation of a program through the injection of a shared

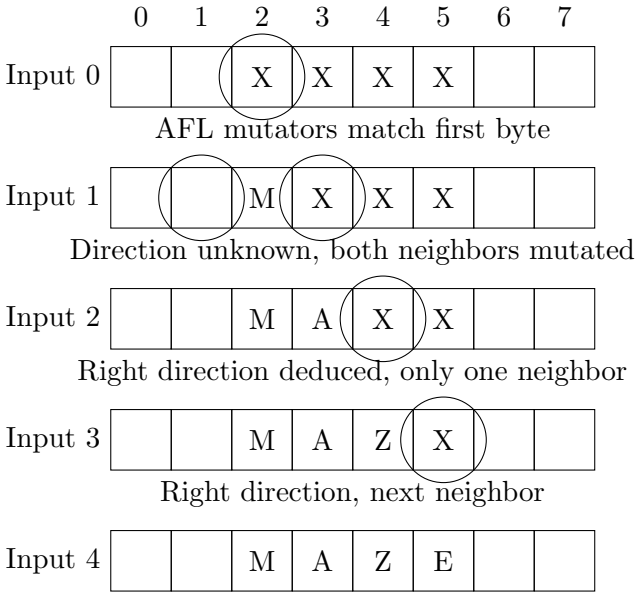

Figure 4.2: Steelix heuristic direction example

library. In this way, no modification is required on the original executable, the injection of the library is sufficient to produce the appropriate instrumentation.

Fortunately, QBDI provides a library, called QBDIPreload, that is built on purpose for this kind of application. It works by setting, from a library constructor, a fake breakpoint at the beginning of the `main` function; this breakpoint is then intercepted thanks to a signal handler. After the interception, the library provides a series of callbacks that allow the user to easily take control of the execution and initialize QBDI.

Using the API that is provided by AFL/QBDI to write library wrappers, it was possible to implement a generic shared library that, once injected in a process, takes control of the execution and sets up the communication with the fuzzer. After that, it starts running the `main` function through QBDI, effectively executing the program under the control of the instrumentation framework.

The injection of the library is obtained exploiting the standard features provided by the default linkers, thus using `LD_PRELOAD` on Linux and `DYLD_INSERT_LIBRARIES` on macOS. These two methods are supported by AFL and thus, simply by setting the `AFL_PRELOAD` environmental variable, it is possible to specify which libraries should be loaded by the fuzzer in the target process.

4.3.2 Dislocator

A problem that is generally encountered while fuzzing is that of determining a valid testing oracle: it is necessary to have a mechanism in place in order to understand when the execution of a certain input produced an unexpected behavior. This is quite easy for certain categories of bugs, for example assert violations, since the program always crashes. However, one of the most significant categories of bugs from a security perspective, that of memory corruptions, does not always exhibit such an evident behavior.

In order to solve this issue, there are several libraries that allow the instrumentation of programs, through compiler passes, in order to check for such problems. An example of this is AddressSanitizer [23], which is exploited by AFL when the source code is available.

The solution, however, is not that simple when targeting binaries: the instrumentation process is, in itself, more complex; moreover, the amount of information available in order to establish the constraints that should be checked is reduced when compared with having access to the source code. For this scenario, AFL provides another solution, called `libdislocator`, which is very expensive in terms of memory usage, but gives good results on small programs.

The general idea behind this library is to hook the standard allocator calls, like `malloc` and `free`, and guard the limits of the allocated buffers in order to catch out of bounds accesses. In detail, each

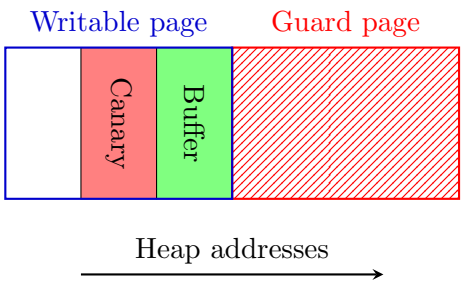

Figure 4.3: Dislocator allocation layout

buffer is allocated aligning it to the end of a memory page and then setting the protection flags of the following one to `PROT_NONE`, so that even a buffer overread causes a segmentation fault. Regarding the lower bound, instead, a simple canary is placed at the beginning of the buffer; this allows to check, on `free`, if there has been a buffer underflow. In addition to this, `free` does not really release memory, it just changes the permission of the page on which the buffer was to `PROT_NONE`, so that use-after-frees can also be captured. Figure 4.3 shows the memory layout of the allocation of a buffer smaller than one page, which is commonly 4 KB.

The problem with this solution is that the minimum allocation size is two pages, even for really small buffers, resulting in a large amount of wasted memory. In addition to this, the change of the permissions through `mprotect` and the allocation of the two pages through `mmap` is orders of magnitude slower than an allocation using the standard allocator, in particular on macOS.

Despite all this, given that the library performs quite well on small programs, it was decided to adapt it to work with AFL/QBDI too. The main issue, however, was that QBDI uses the default allocator in order to perform heap allocations, potentially leading to an unnecessary loss of performance if `libdislocator` was always active: it is thus desirable to intercept only the calls made by the target binary and forward those made by QBDI to the library allocator.

In order to implement this solution, however, it is necessary for the dislocator library to keep a state so that it can know when it is enabled and when it is not; the switching of this state is quite easy to do since QBDI already provides the possibility to register callbacks that are run when switching between QBDI and its guest. Keeping the library in a consistent state, however, represents a complex issue: first of all, it was necessary to give the client of the library the possibility to switch it on and off for every single thread, in order to avoid unwanted interactions. In addition to this, it was necessary to handle various special cases, like the possibility of allocating a buffer in one thread and deallocating it in another, where the library is potentially disabled.

It is also important to notice that even the forwarding to the standard allocator, due to the logic that the library needs to run for every allocation, can be considerably slower then a direct call. Through extensive benchmarking it was understood that the best solution with regard to performance was an implementation that resorted to atomic operations in critical sections using locks only when really necessary.

Further adjustments were needed to mimic the behavior of the standard allocator present in macOS. In particular, it was necessary to introduce support for aligned allocations and zone support, both of which are features on which macOS frameworks strongly rely.

The final result is a library that is able to handle even the most complex multithreading scenarios without excessively compromising the performance. It can be used simply by injecting it in a process with the standard preload mechanisms and then enabling it through environmental variables or the exposed API. It was successfully tested on several programs, including the Python interpreter on both Linux and macOS. Table 4.1 shows the results of the benchmarks for the final version on macOS, compared against the standard allocator; the sizes of the allocations used for the benchmark fall in the "tiny" and in the "small" pools, respectively [1B – 992B] and [993B – 127KB], larger sizes are just allocated page by page.

Allocator	Size	Function	Mean time (ns) ± std (%)
Original allocator	tiny	`malloc`	$44.66 \pm 0.34\%$
		`free`	$42.08 \pm 0.79\%$
	small	`malloc`	$42.06 \pm 3.08\%$
		`free`	$41.94 \pm 2.21\%$
Dislocator disabled	tiny	`malloc`	$132.77 \pm 2.39\%$
		`free`	$137.39 \pm 0.91\%$
	small	`malloc`	$153.70 \pm 0.83\%$
		`free`	$136.19 \pm 6.72\%$
Dislocator enabled		`malloc`	$6082.87 \pm 0.12\%$
		`free`	$2512.23 \pm 0.30\%$

Table 4.1: Dislocator benchmark results on macOS

Chapter 5

Evaluation

5.1 Target binaries

This section is meant to give a general description of the software that was used for the evaluation of the fuzzer presented in this document. The characteristics that are significant for the evaluation will also be examined in depth.

5.1.1 LAVA-M

This dataset was selected since it became a constant in the evaluation of fuzzers after its publication; all the fuzzers presented in Section 2.2 have been evaluated against it.

The first thing that is important to note about this dataset is that it was generated automatically so, even if it resembles real software, it is instead synthetic. It was produced running a fault injection tool called LAVA [8] on four programs belonging to the GNU coreutils project, which were modified at a source code level. These binaries are `base64`, `md5sum`, `uniq` and `who`; their sources were made available by the authors.

In order to perform the evaluation of the project presented in this document, the sources were compiled for x86_64; this is different from what was done for the evaluation of Steelix, which was tested on x86, but QBDI does not support that architecture at the moment. In addition, a small modification to the sources was needed since, in `md5sum`, the injection performed by LAVA introduced the use of an uninitialized pointer, which had to be explicitly initialized to `NULL`.

One of the factors that make LAVA-M distant from real software is that the bugs introduced by LAVA tend to be similar. The injection process starts by running an instrumented version of the target program that uses taint analysis to point out locations in which dead, uncomplicated data are available (DUA). These simply represent the possibility to directly access, at a certain location, sequences of input bytes that, up to that point, have not been modified and do not influence the control flow. After that, LAVA looks for attack locations close to where DUA are located; these are selected according to the type of bug that the user decides to inject and are usually array accesses, or bulk copies. Once these attack locations are selected, for each of them a number of bugs is injected. These bugs make use of the DUA, for example by adding it to the argument of a target bulk copy function; the bug, however, is triggered only when the selected sequence of input bytes matches a particular magic sequence of four bytes, which is selected randomly at compile time. It is important to note that the sequence of input bytes that are checked in a guard statement is always retrieved through a function called `lava_get`, which extracts the four selected bytes from the input and parses them as an integer; this value is then returned by the function and checked in the guard statement.

The construct consisting in having a magic bytes sequence protecting every bug makes all the faults similar from the perspective of the fuzzer. Once it is able to reach the condition, it only needs to crack it to trigger the bug. For this reason, a heuristic like the one proposed for Steelix is likely to provide good results in such a benchmark, since it helps in solving exactly the problem of cracking the guard. This statement, obviously, assumes that the exploration of the code is already good without the heuristic.

	Reproduced crashes	Expected crashes
`base64`	44	44
`md5sum`	52	57
`uniq`	20	28
`who`	2104	2136

Table 5.1: Results of LAVA-M test suite in QBDI

It is also important to note that the authors of LAVA-M provide a series of test cases in order to verify that at least a portion of the inserted bugs can be triggered on the compiled version of the binary. This test suite was really useful since it pointed out a problem that was not noticed until the evaluation phase: the use of a dynamic binary instrumentation framework is more likely to introduce artifacts in the execution as compared to static instrumentation since the change in the execution process is more extensive. In particular, it is important to note that QBDI was created in order to transparently replicate the defined behavior of a program, but it gives no guarantees on the undefined one. For this reason, running the test suite that was supposed to trigger the bugs in the LAVA-M binaries gave different results from the ones expected: a small portion of the bugs was not reproducible when running the program in QBDI, without any kind of instrumentation. As a consequence, observations on the reduction in the number of bugs found by the fuzzer need to take into account the fact that some of them may be masked by QBDI itself, which may prevent some of the crashes. The results of this test suite are presented in Table 5.1.

5.1.2 ImageIO framework

The second target selected for the evaluation is the ImageIO framework provided by Apple as part of macOS. It was selected since it constitutes a good benchmark to study the performance difference the Steelix heuristic introduces on large real software, both in terms of structure and in terms of dimensions. In addition, it allows to test the fuzzer on a different platform, macOS, and on a library written in a different language, C++, as compared to LAVA-M.

In concrete, this library provides opaque data types to read and write images from a specified source or to a specified destination. In addition, it hides from the user the complexity of handling the image format by deducing it directly from the data provided: according to the header of the file, the framework selects the plugin that is supposed to handle the decoding of an image and forwards the request. As many frameworks written at Apple, it relies on a framework called CoreFoundation, which is written in C and provides a common ground for the exchange of data between components. The range of supported formats is considerable, ranging from classic PNG images to obscure Apple icon formats; RAW camera formats are also supported, usually as a variation of TIFF or even through a parser generated using Yacc. Some of the format plugins rely on external libraries that are loaded together with the main module; these libraries were also run through the DBI framework while testing this software.

The source code for the framework and the libraries linked against it are not available, the only information on how to use it comes from the official documentation [13]; the information about the implementation, instead, comes from reverse engineering work done on the build of the framework being tested.

Since there were not any bugs known before the beginning of the evaluation, the interesting point in testing this framework was to evaluate the improvement in the ability of the fuzzer to explore it more in depth, thus increasing the likelihood of encountering bugs. In addition to this, it was important to verify the ability of the heuristic to scale on software with a large number of basic blocks.

5.2 Evaluation setup

This section is meant to give a description of the setup that was used to conduct the tests presented in the rest of the document. Given that the two targets used in the evaluation, LAVA-M and ImageIO,

run on different platforms, one of which is proprietary, the tests were conducted on different machines.

5.2.1 LAVA-M

The LAVA-M dataset was tested on a machine running Fedora 28, updated at the end of July 2018, with an Intel Core i7-2600 CPU and 16 GB of RAM. Given that LAVA-M is formed by four binaries, four fuzzer instances were run in parallel for every single configuration that it was decided to test. The compilation of the LAVA-M binaries was made on that same machine using the default development tools.

As it is usually done for LAVA-M evaluations, each single test on the four binaries was run for 5 hours. The data on the test was collected processing the output that AFL appends to the `plot_data` file which is normally used to create plots showing the behavior of the fuzzer; some of the output directives were modified in order to print more precise information.

One additional script was also created in order to collect information regarding the unique LAVA-M ids produced by the crashes. These ids are printed by the binaries right after the guard statement protecting a bug is cracked, helping to distinguish unique crashes more easily and with certainty. Counting the crashes found in this way gives information that is more precise than what AFL shows on its interface.

As already discussed in Section 4.3, the introduction of the support for full programs and not just for libraries allowed to run the tests following exactly the same method used for the LAVA-M tests in the literature. In particular, `base64` was run with the `-d` option, `md5sum` with the `-c` option, `uniq` and `who` without any additional options. The reliance on this functionality, however, prevented the use of persistency which cannot be relied upon if the program keeps a global state, as it almost always happens.

It is important to note that the LAVA-M dataset also provides fixed seeds for each binary; those were used for consistency, even if it is likely that with better crafted inputs it may have been possible to achieve better performance, for example by shrinking the input for `uniq`, which is unnecessary long.

5.2.2 ImageIO

Since ImageIO is a macOS framework that runs only on proprietary Apple hardware, it was tested on two MacBook Pros 13-inches produced in early 2015. The version of the framework being tested was the one provided in macOS High Sierra 10.13.6. The compilation of the library wrapper was also done using the development tools released with the same version of macOS.

In this case, given the larger dimension of the software, it was necessary to give the experiments more time, as compared to LAVA-M, for the exploration to properly stabilize. Each experiment was run for 24 hours, with a maximum of 2 instances per machine. Most of the results were collected, as before, from the `plot_data` file created by AFL.

Since in this particular case the most interesting information is not the number of crashes but the coverage the fuzzer is able to obtain, a separate tool was implemented in order to extract this information. Using QBDI, it was possible to record all the edges encountered by the program while running the samples in the queue without resorting to a bloom filter. In this way, even if the execution time for each sample was slower, it was possible to obtain precise data regarding the cumulative edge coverage obtained after the execution of each sample.

In this case, the selected target was a library function named `CGImageSourceCreateImageAtIndex`, which was targeted through the use of a library wrapper. This allowed the use of the persistency functionality provided by AFL/QBDI, which helps enormously when fuzzing on macOS since the `fork` system call is really slow.

With regard to the seed inputs used, it was decided to provide the fuzzer with a corpus of 11 images with different formats, each one of them containing a single orange pixel. The image formats selected were ATX, BMP, GIF, JP2, JPG, KTX, PBM, PNG, PSD, TGA and TIFF; they were chosen to try to cover as many formats as possible between those supported by ImageIO.

5.3 LAVA-M Evaluation

5.3.1 Steelix comparison

This section provides a comparison between the results obtained by the fuzzer presented in this document and those obtained by the implementation presented in the original Steelix paper. While the results of multiple runs, around 10, are available for the evaluation of AFL/QBDI, the authors of Steelix published the results of only one run; this is likely to be their best one, even if it is not explicitly stated in the paper.

Every plot shows the best run obtained by Steelix, as a single line, and aggregates instead the results obtained by AFL/QBDI following the guidelines presented by Klees et al. [14]: given that it cannot be assumed that the results have a normal distribution, the median needs to be used in place of the arithmetic mean. As a consequence of this, the plots show a 95% confidence interval for a median, calculated following the same guidelines. In addition to that, the plots contain a lighter area that is delimited by the maximum and minimum results obtained.

An evaluation between AFL/QBDI with and without the Steelix heuristic had also been planned; however, AFL/QBDI without heuristic was not able to find any LAVA crashes in any of the runs. In all likelihood, this happened due to the fact that it was not capable of cracking the guard statements protecting the bugs inserted by LAVA.

base64 Figure 5.1 shows the results obtained on `base64`, which are far better using Steelix than using AFL/QBDI. This is probably due to one of the limitations of the current implementation, most likely related to the fact that QBDI does not allow the unrolling of comparisons including memory accesses in their operands. This hypothesis is supported by the fact that, observing the plateau generated after the first few minutes, it seems impossible for the fuzzer to explore additional portions of the program.

It is significant to notice that the discovery of bugs during the first few minutes is faster using AFL/QBDI, probably due to the scheduling of the test cases based on the double queue model presented in Section 4.2.

md5sum Figure 5.2 shows instead the results obtained on `md5sum`. In this case, the progression obtained using Steelix shows better performance, but the final result, comparing the best runs, is in favour of AFL/QBDI.

Observing the plot for AFL/QBDI, there is a high variability in the progression after 3 hours. This is probably due to the fact that, in order to overcome the limit of 20 bugs, the fuzzer needs to explore a section protected by a condition which can be matched only through a random mutation; this mutation can happen starting at 1.5 hours, but the precise moment in which it happens is different for every run. Given this hypothesis, it is likely that the higher execution speeds that Steelix obtains, roughly double, help in generating the required random mutation faster than AFL/QBDI.

uniq Figure 5.3 shows the results obtained by the two fuzzers on `uniq`, which are strongly in favour of AFL/QBDI. This large difference in performance is probably due to some kind of limitation in the implementation of Steelix; a possibility is that the static instrumentation process did not work properly but, as already discussed in Section 1.3, this is difficult to assess.

It is worth to observe the peculiar ladder pattern generated by Steelix: given the fact that the crashes are triggered at equal intervals of time, it is reasonable to assume that the 6 bugs found are clustered together, on the same statement. The fact that there is a delay in their discovery is probably due to the need to process other test cases which, in the main queue, are located before the ones that help cracking the guard conditions.

The variable behavior starting at 1.5 hours can again be attributed to a condition that can be satisfied only through a random mutation, which randomly happens after 1.5 hours.

who Figure 5.4, the last for this section, shows instead the results obtained on `who`. In this case, despite the higher execution speeds, Steelix is worse than the best run for AFL/QBDI in discovering

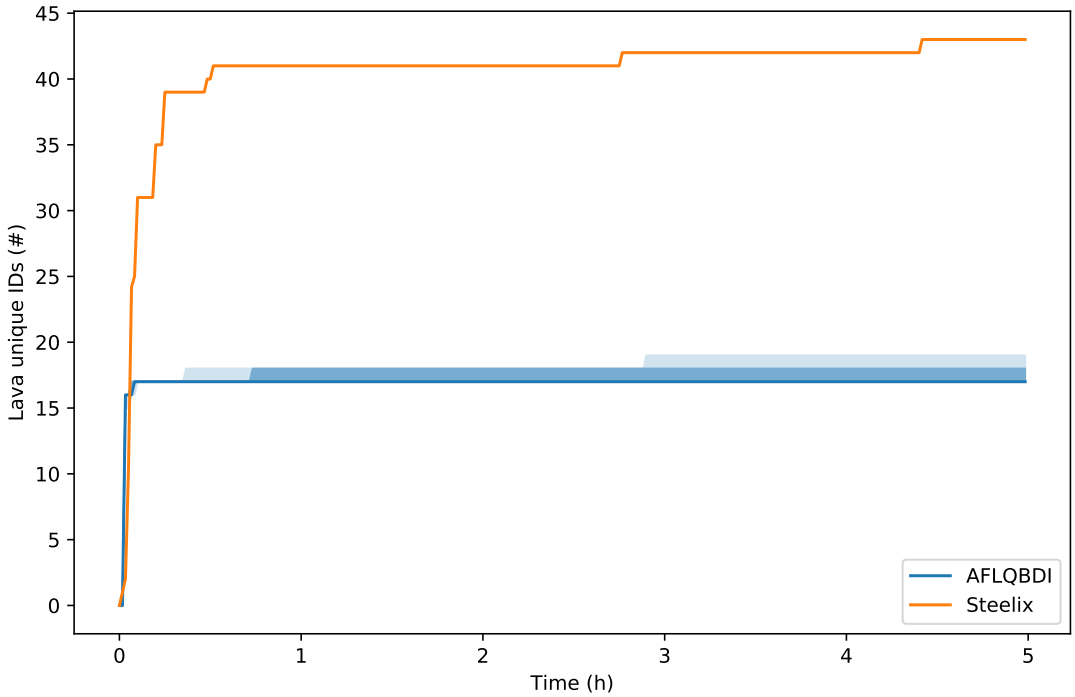

Figure 5.1: Cumulative crash count over time on `base64`

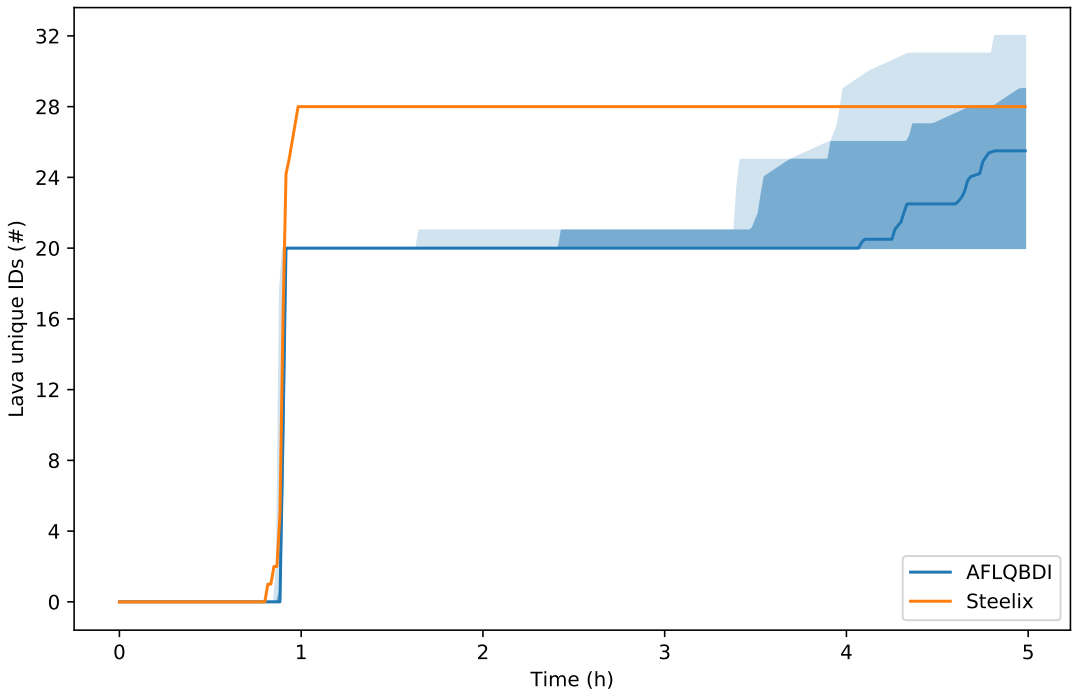

Figure 5.2: Cumulative crash count over time on `md5sum`

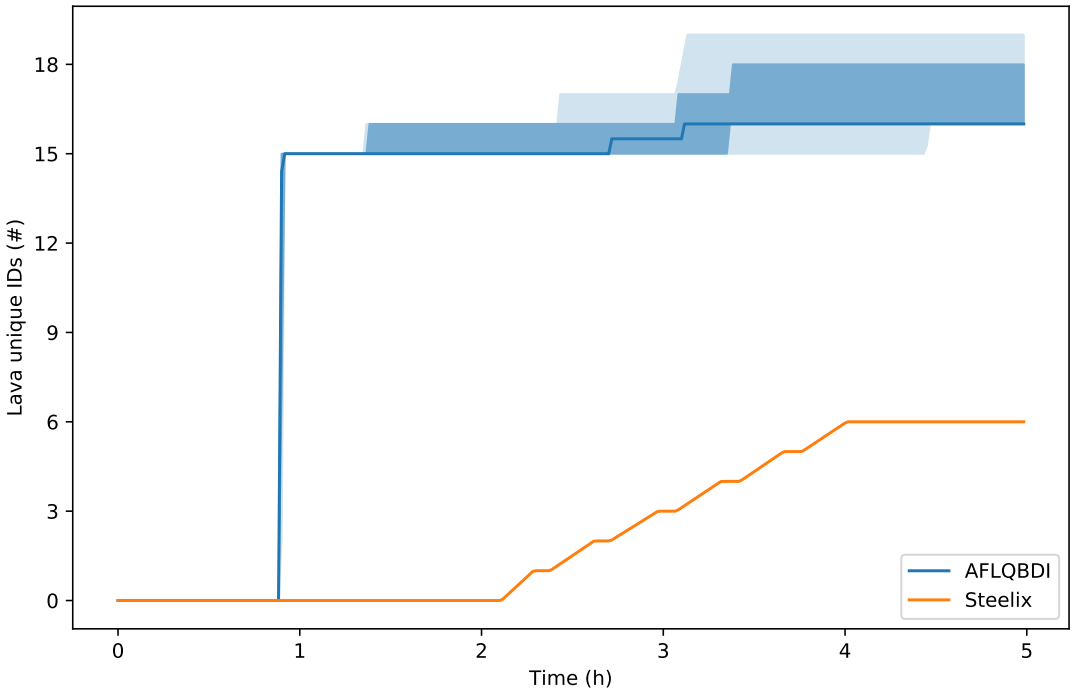

Figure 5.3: Cumulative crash count over time on `uniq`

bugs during the majority of the five hours. However, when exploring additional portions of the program requires random mutations to produce precise results, Steelix has the obvious advantage of being able to make more attempts.

In this case, it is likely that both fuzzers require additional time to find all the bugs that the heuristic allows to target, the five hours limit is likely to strongly affect the results.

5.3.2 Bloom filter size comparison

The plots presented in this section are meant to justify the choice of doubling the memory area shared between AFL/QBDI and the instrumentation code. As already discussed in Section 4.2, a half is used for the execution trace and a half is used to report progresses on comparisons.

The expectations are that, diminishing the size of the shared area, the execution speed should be higher since the shared memory area can be parsed in a lower amount of time and it can better fit in the processor cache. As a trade-off to this behavior, a smaller shared area also creates a higher chance of hash collisions in the bloom filters for the execution trace and the comparisons; as a consequence, a lower amount of crashes can be expected.

It is important to notice that only one run per size of the shared area is available due to constraints in the computational resources available, which were focused on more important experiments. As a consequence, the plots presented do not hold statistical significance by themselves, but only when compared to ones presented in the previous section.

As a general information, the default bloom filter size in AFL is 64 KiB, which translates in a 128 KiB shared memory area for AFL/QBDI since it uses two bloom filters. This size is the one that has been used for the runs presented in the previous section when comparing with Steelix. It is important to note that the size reported in the plots is the size of a single bloom filter, not the size of the whole memory area. The usage of the bloom filters recording the execution trace for the various sizes are presented in Table 5.2; execution speeds are not reported since they are influenced too strongly by early exits in the program to provide valuable information.

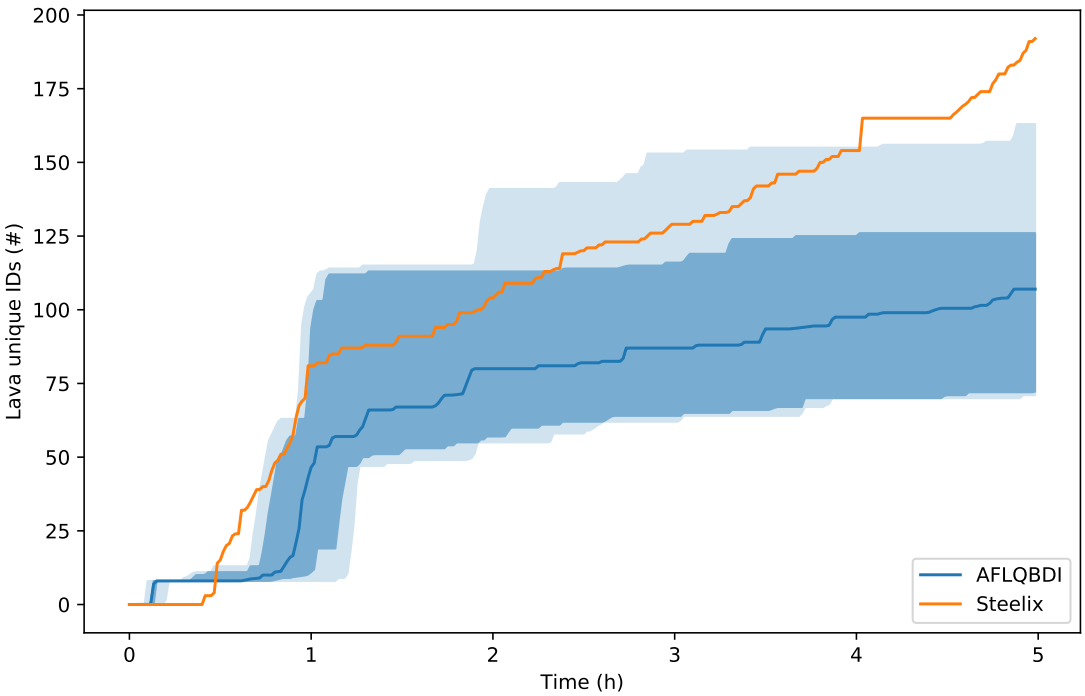

Figure 5.4: Cumulative crash count over time on `who`

base64 Comparing the runs shown in Figure 5.5 with the ones that were presented in Figure 5.1, it is quite evident that they match perfectly the normal behavior. This, in all likelihood, is due to the fact that `base64` is a really small program which exploits only a small portion of the shared area. As a consequence, the used portion of the bloom filters is likely to fit in the processor cache. The likelihood of collisions is also low for every tested size, given the low usage of the shared memory.

md5sum Considering the runs on `md5sum` shown in Figure 5.6, it would appear that an increase in the dimension of the bloom filters could lead to better performance in terms of the possibility of finding more crashes. However, when comparing this plot with the one in Figure 5.2, it is evident that a good part of the runs falls in the expected behavior; in addition, this hypothesis is quite unlikely given the reduced size of the binary. The observed behavior can thus be attributed to a condition that needs a random mutation to be matched; by chance, the mutation required has happened in that particular order, which is completely unrelated to the bloom filter size. Additional runs would be required to provide a more precise answer.

uniq The comparison between Figure 5.7 and Figure 5.3 provides essentially the same results as those given by `base64`. Given the small size of the binary, the variation of the size of the bloom filters does not influence the behavior of the fuzzer even if, as for `md5sum`, there is a subtle increase in the count of the used cells with larger filters which could indicate the presence of collisions.

who In this case, it is important to note that there is a strong reduction in the usage of the bloom filters when considering sizes of 16 KiB and 32 KiB. This reduction, however, does not translate in lower performance for the fuzzer since, even if the run with 16 KiB filters shows poor performance when compared to the others, as seen in Figure 5.8, it fits the distribution shown in Figure 5.4. In addition, the reduction in terms of usage does not reflect on the rest of the sizes, so it is simply advisable to avoid using 16 KiB bloom filters for a program the size of `who`, as also suggested by the author of AFL.

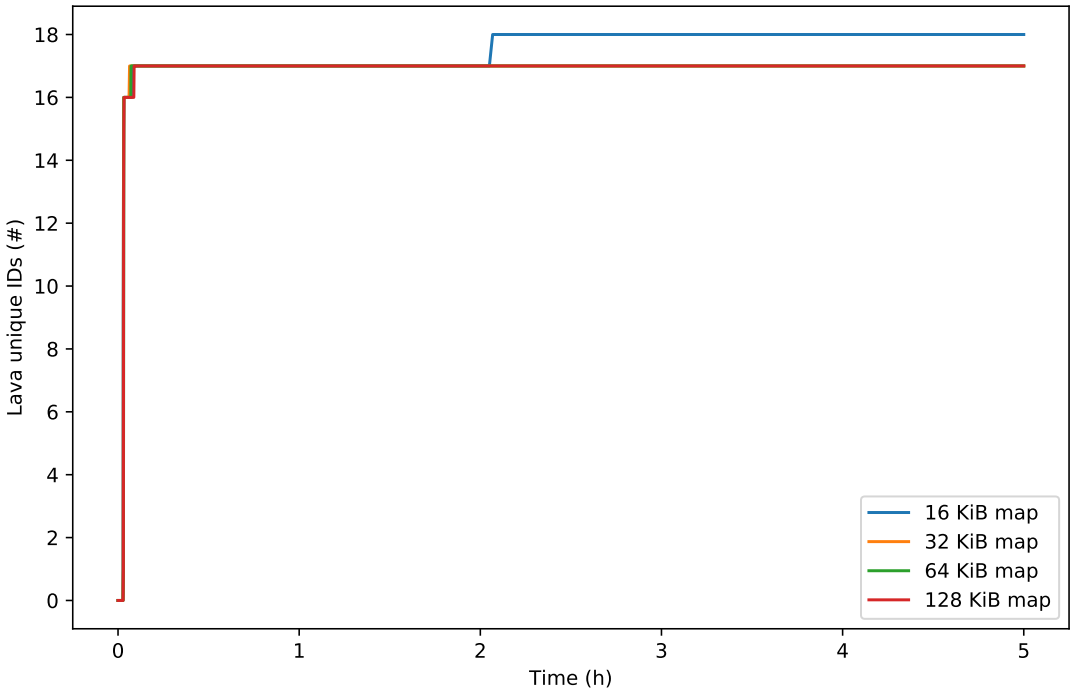

Figure 5.5: Cumulative crash count over time on `base64` with different bloom filter sizes

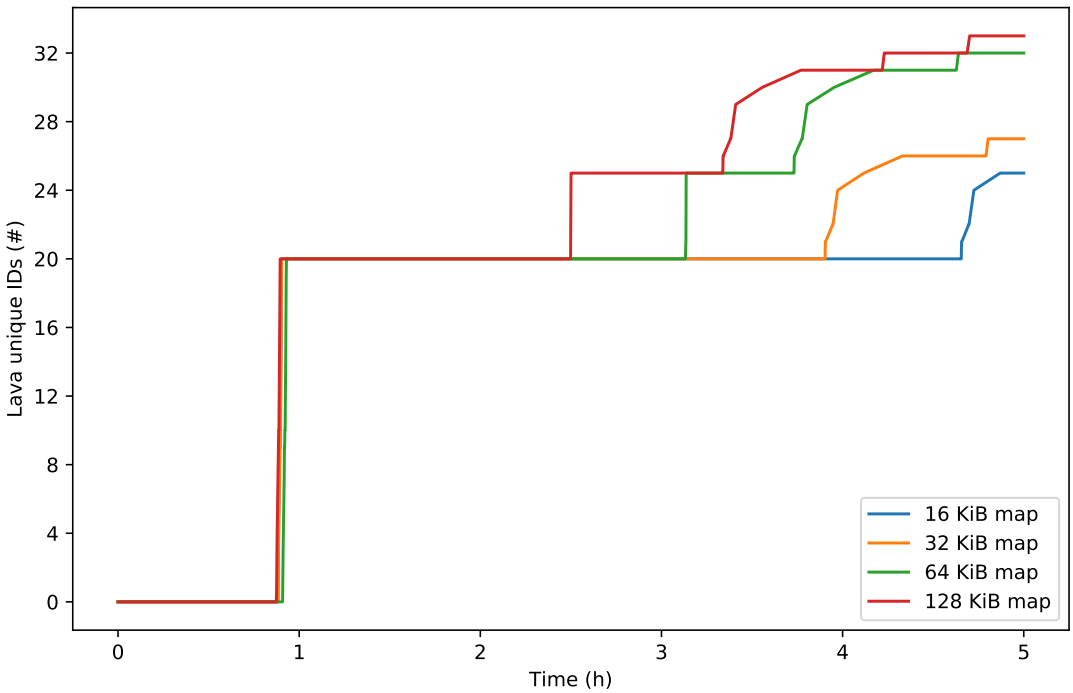

Figure 5.6: Cumulative crash count over time on `md5sum` with different bloom filter sizes

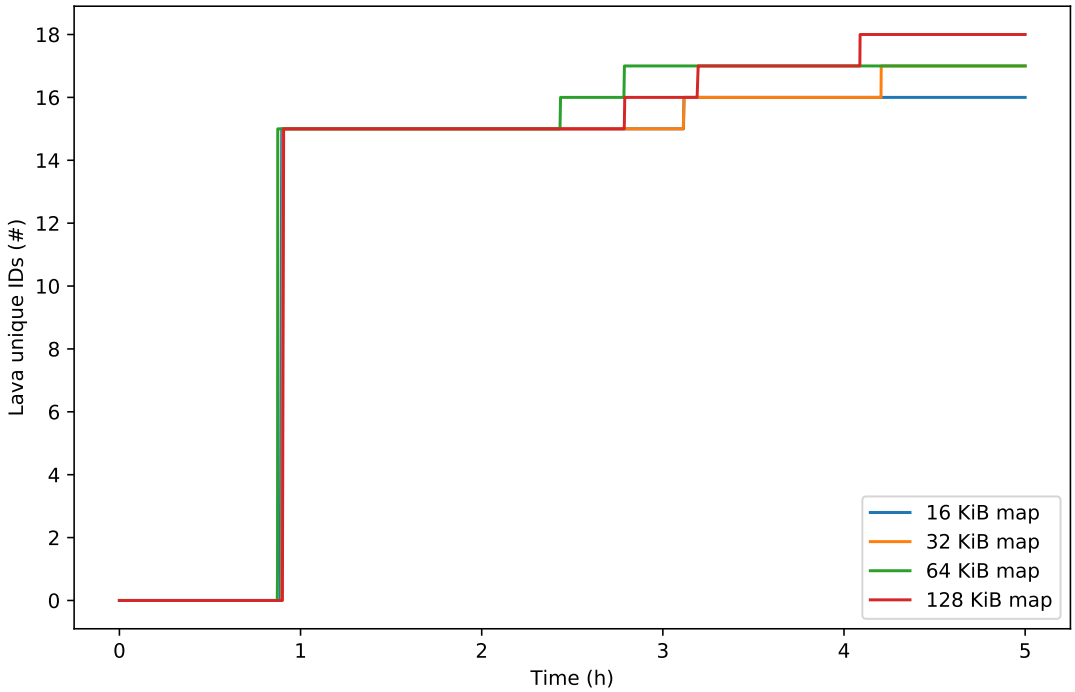

Figure 5.7: Cumulative crash count over time on `uniq` with different bloom filter sizes

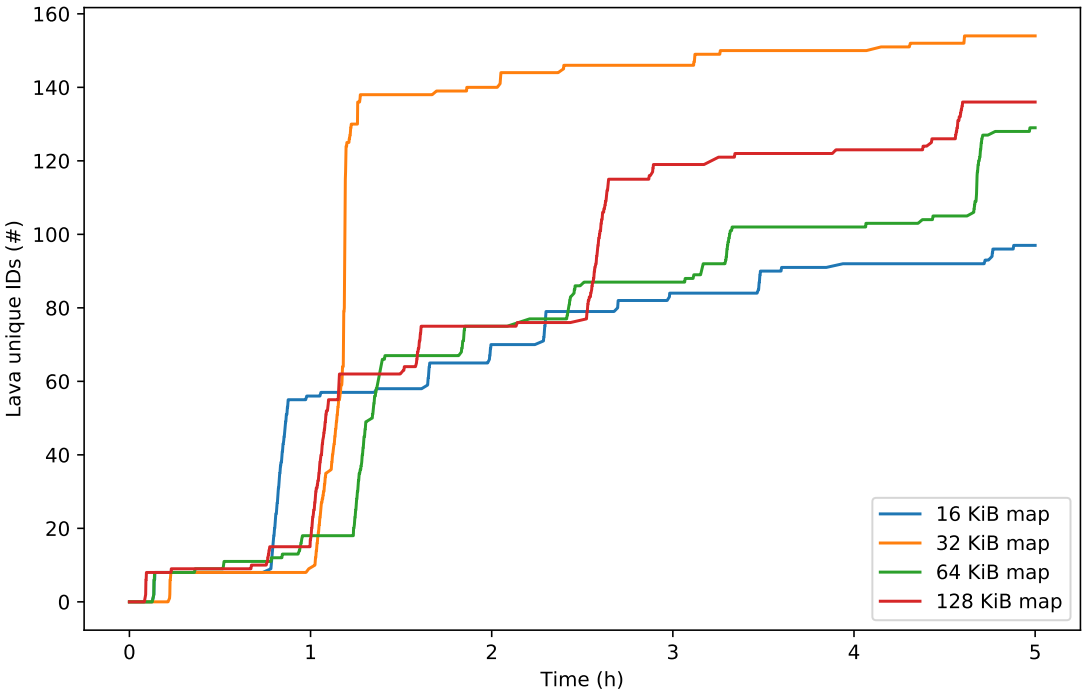

Figure 5.8: Cumulative crash count over time on `who` with different bloom filter sizes

Filter size	`base64` usage	`md5sum` usage	`uniq` usage	`who` usage
16 KiB	481 B	568 B	364 B	9549 B
32 KiB	479 B	570 B	365 B	10603 B
64 KiB	476 B	570 B	361 B	10507 B
128 KiB	469 B	590 B	380 B	10517 B

Table 5.2: Usage of the bloom filter recording the execution trace for LAVA-M binaries

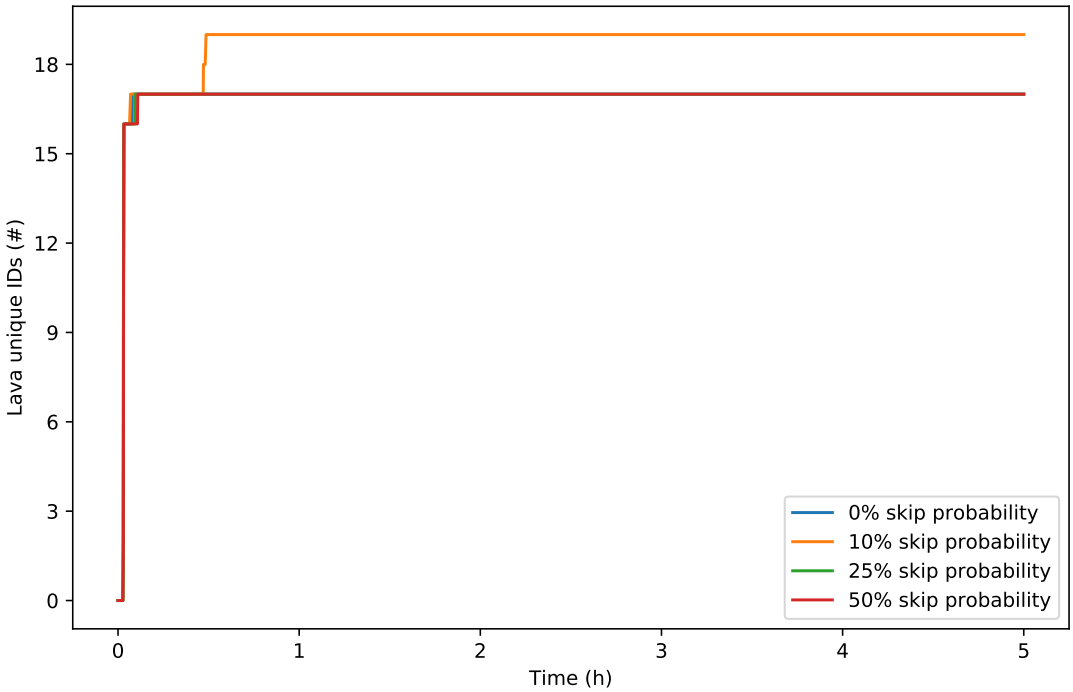

Figure 5.9: Cumulative crash count on `base64` using different skip probabilities

5.3.3 Skip probability comparison

This section is meant to assess the influence a different probability of skipping the processing of comparison progresses has on the overall performance of the fuzzer.

It can be expected that a low probability will help in cracking conditions faster, but it may lead to a higher overhead due to a slower processing of the main queue. This, in turn, can lead to a delay in the progress of the exploration of the binary.

A higher probability can instead help in diluting the overhead of the exploration, but may slow down the cracking of branch conditions, which can limit the exploration in itself.

Also in this case, only one run has been recorded for each skip probability, so it is necessary to compare these plots with those provided during the comparison with Steelix. These have been recorded with a skip probability of 1%, which was chosen as a way to be as close to 0% as possible, but still avoid starvation in edge cases.

base64 As it happened for the comparison between various bloom filter sizes presented in the previous section, Figure 5.9 does not provide any significant insight related to the changes in the skip probability. The behavior falls perfectly into the distribution shown in Figure 5.1, apart from a spurious crash discovered earlier than expected in the run with 10% probability.

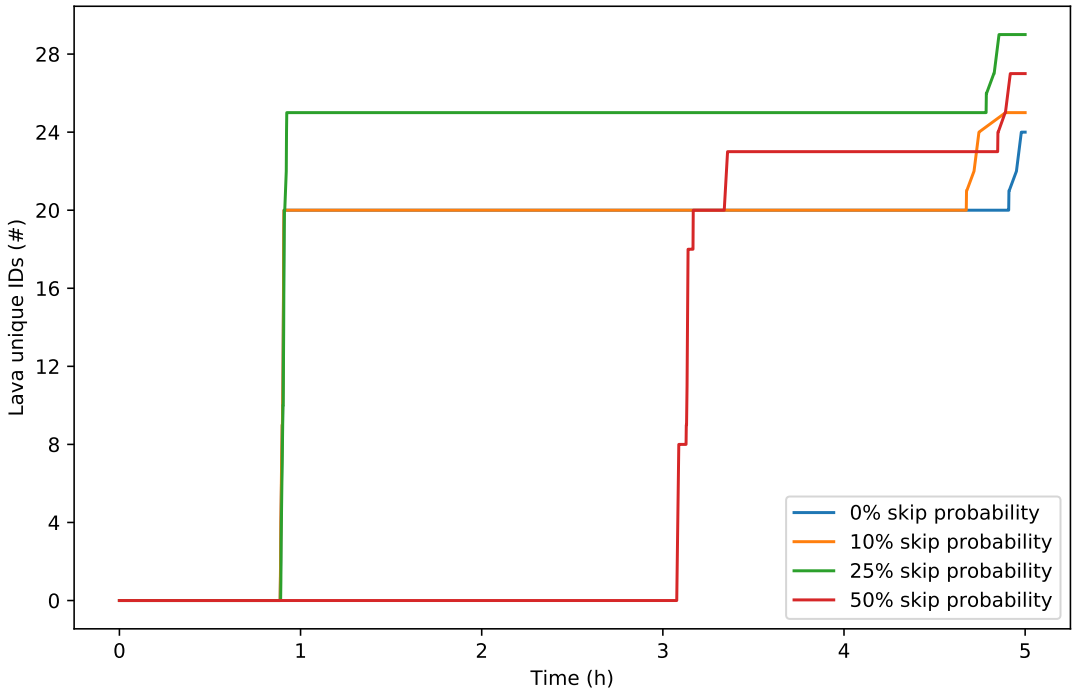

Figure 5.10: Cumulative crash count on `md5sum` using different skip probabilities

md5sum Observing Figure 5.10 instead, it is quite evident that the run with 50% skip probability shows the behavior that was expected. Indeed, there is a considerable delay with respect to the progression in the number of crashes up until 20, which are normally discovered all at once. It is even possible to see small plateaus which are caused by the choice of processing the main queue instead of the one containing comparison progress test cases. This is due to the fact that, normally, the guards on the bugs up to 20 are collected and cracked all at the same time.

uniq As it happened for `md5sum`, Figure 5.11 shows the same delay in finding the crashes for the runs with 50% and 25% skip probability, when compared with Figure 5.3. The same reflections can be applied to this case since it seems that the fuzzer is usually able to collect and crack all the guard statements up to bug 15 at the same time. It can thus be deduced that a significant delay in the processing of comparison progresses inevitably leads to a delay in the discovery of new crashes.

who In contrast to what happened for `md5sum` and `uniq`, it appears that the progression of the discovery of the crashes for `who` is not significantly influenced by changes in the skip probability. All the runs shown in Figure 5.12 fit perfectly in the distribution shown in Figure 5.4. This is probably due to the fact that a behavior as evident as the sudden discovery of more then 10 crashes at once is not present in `who` and thus the delay in crash discovery is less noticeable given the amount of runs available.

5.4 ImageIO evaluation

5.4.1 Heuristic evaluation

This section illustrates the results of the evaluation of the heuristic presented in the Steelix paper on ImageIO. The goal of this evaluation is to discuss the advantages in terms of exploration of binaries provided by the heuristic; for this reason, the plots do not show the amount of crashes found but instead the amount of edges of the control flow graph that have been explored.

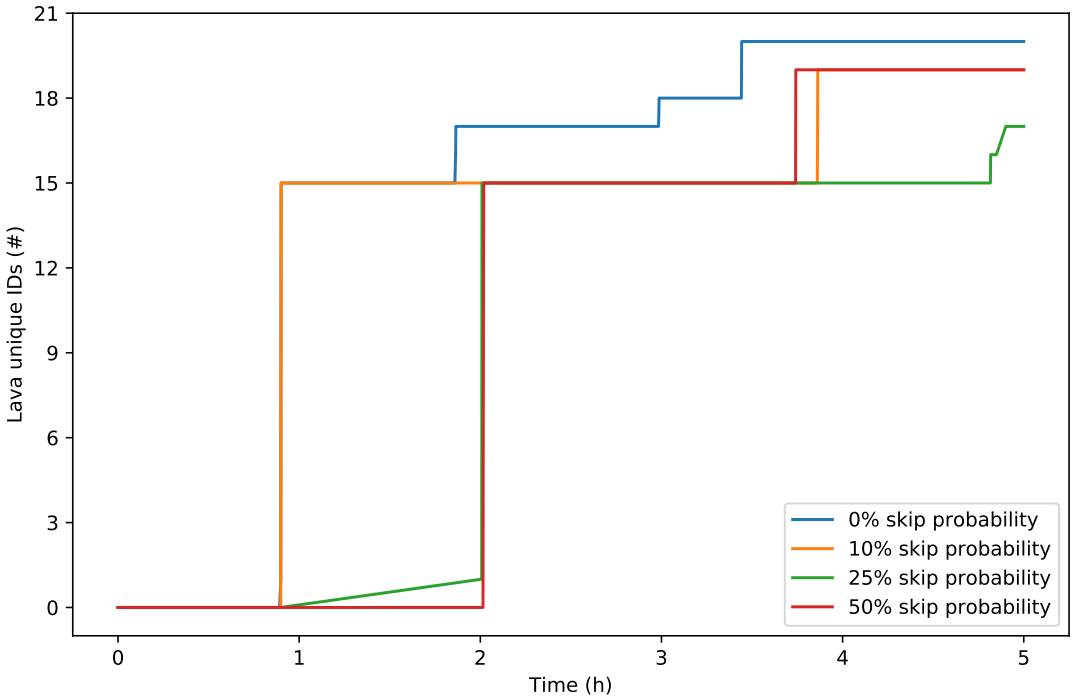

Figure 5.11: Cumulative crash count on `uniq` using different skip probabilities

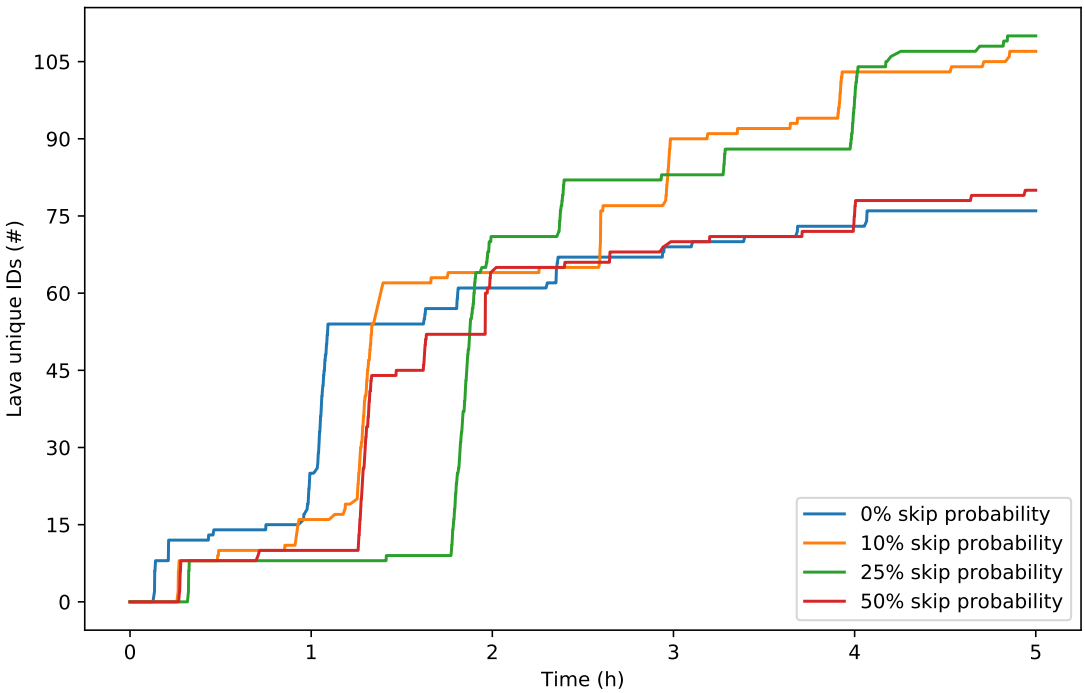

Figure 5.12: Cumulative crash count on `who` using different skip probabilities

The aggregation of the 10 runs recorded is performed, as before, following the guidelines presented by Klees et al. even if, in this case, further discussion is needed in order to properly illustrate the results. Since no restrictions where imposed on the time limit, it was decided to follow the advice proposed in the same work and set a time limit of 24 hours.

Observing Figure 5.13, the first thing that can be noted is that the run with the heuristic disabled has a behavior that is more consistent than the one shown with the heuristic enabled. In detail, the median value proves to be worse when the heuristic is enabled, but the confidence intervals show that there is a significant probability that the performance can be better.

In order to clarify the content of the plot, each run with the heuristic enabled was examined. The result was that the runs are not equally distributed in the confidence interval, they can take one of two paths with equal probability: one that approximately follows the median shown and another that is really close to the upper bound of the confidence interval. These two paths separate around 2 hours after the experiment has started, with a difference of almost 1000 edges explored; this is clearly noticeable by the square shape that the confidence interval draws. Investigating the reason of this difference in behavior, it became evident that it is generated by one single sample which contains a specific tag of the TIFF format that probably triggers an XML parser. This sample is not present in the runs following the lower path. Comparing the shapes of the median progressions with and without the heuristic, it is also evident that, when the heuristic is disabled, that particular sample is always generated.

The most likely reason for this behavior is that, when the heuristic is disabled, that sample is produced by a deterministic mutation operator and thus is always present. In the case in which the heuristic is enabled, instead, sterile comparison progress test cases are stored in the main queue; probably one of them, which is generated with a 50% probability, enters the AFL preferred set in place of the sample that, through the deterministic mutation cited before, generates the target TIFF test case.

If this is the case, the corpus chosen for the evaluation is favouring the fuzzer with the heuristic disabled, at least partially. It is likely that, using a different corpus, the final result would be more stable and in favour of AFL/QBDI with the heuristic enabled. This situation, however, is a consequence of the queue flooding effect illustrated in Section 3.1, which is only partially mitigated by Steelix, and thus should be taken into account.

Another feature of the plot that is worth noting is that, when taking the best progression for both the distributions, there is a delay when the heuristic is enabled that is caused by the additional overhead introduced. For this reason, using the heuristic does not provide an advantage with short runs, lasting 4 hours, but provides better results after that.

5.4.2 Bloom filter size comparison

This section provides an analysis of the behavior of the fuzzer when using shared memory areas with different sizes. As for the LAVA-M binaries, only one run for each size is available; this means that, again, the progressions presented in Figure 5.14 do not have statistical significance by themselves, they need to be compared to Figure 5.13.

Observing the progressions obtained when using different bloom filter sizes, the presence of the behavior characterized by two different paths which was described earlier is quite evident. Apart from this, it appears that there is a slight penalization of smaller bloom filters with respect to larger ones, probably due to collisions in the one that records the execution trace. This hypothesis is supported by the bloom filter usage reported in Table 5.3 which shows a decrease in the tainted cells for smaller filters. Within the plot, however, this difference is too small to be considered statistically significant.

In conclusion, it was decided to avoid using 16 KiB bloom filters since, as for `who`, they may provide lower performance due to collisions on larger programs. In order to statistically justify this choice, however, a larger sample should be used.

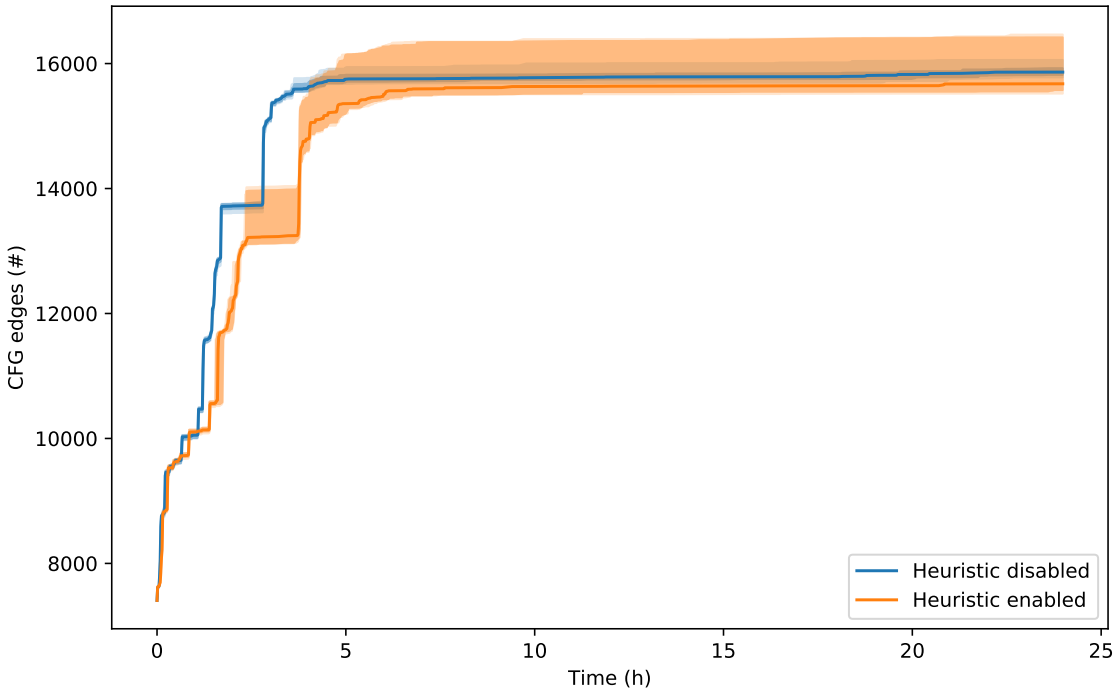

Figure 5.13: Cumulative edge count over time on ImageIO

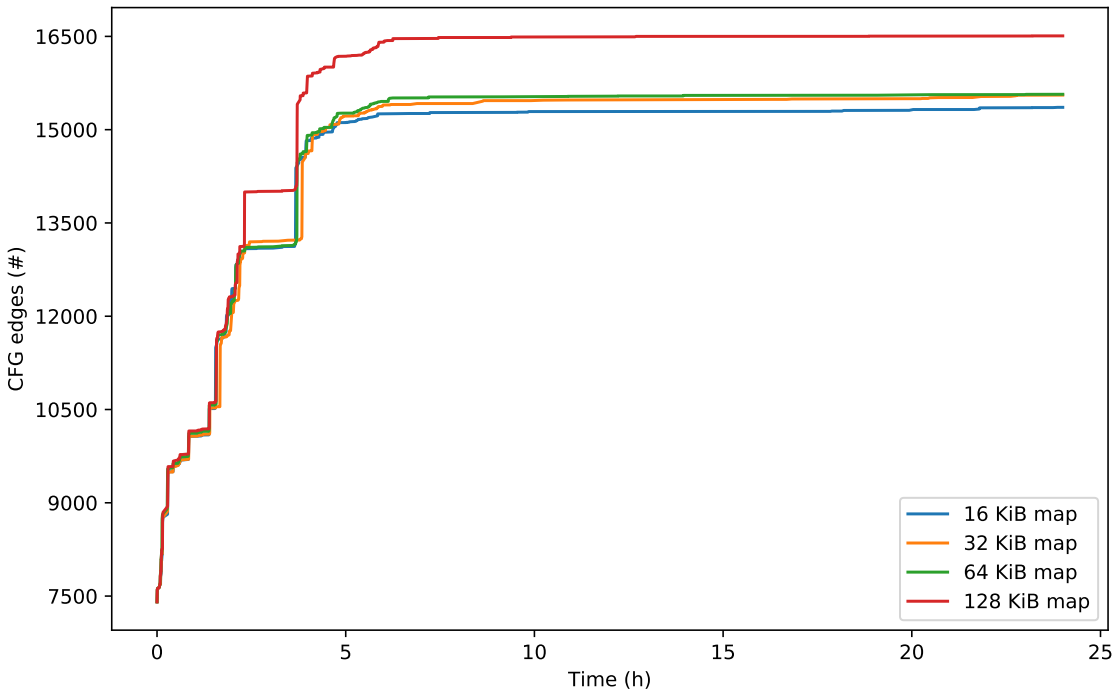

Figure 5.14: Cumulative edge count on ImageIO with different bloom filter sizes

Map size	ImageIO usage
16 KiB	10438 B
32 KiB	13029 B
64 KiB	14625 B
128 KiB	16649 B

Table 5.3: Usage of the bloom filter recording the execution trace for ImageIO

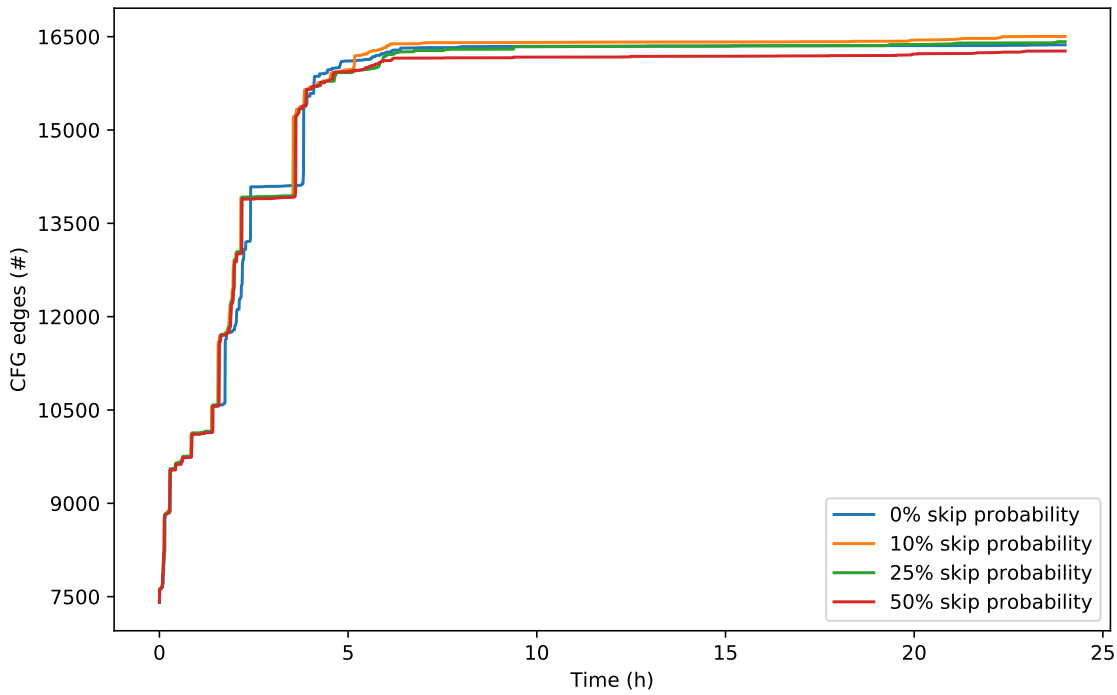

Figure 5.15: Cumulative edge count on ImageIO using different skip probabilities

5.4.3 Skip probability comparison

This section is meant to assess the influence that the probability of skipping the comparison progress queue has on the exploration of the target software. Given that, for ImageIO, the data collected include the coverage in terms of CFG edges, it should be possible to observe the effects of that change more precisely. As before, only one run is available for each skip probability presented in Figure 5.15; as a consequence, significant conclusions can be drawn only comparing that plot with Figure 5.13.

All the progressions exhibit a behavior that is consistent with the distribution represented before. As a consequence, there is no evident influence of the change in skip probability on the behavior of the fuzzer.

The only significant observation is that, in the first 4 hours, when the fuzzer is still performing deterministic mutations, there is an evident translation of the 0% probability plot with respect to the others. This translation is caused, on the x axis, by the time overhead introduced by the processing of the comparison progress queue and, on the y axis, by the advantage the early processing provides in terms of new edges. The translation generates a disadvantage in the first few hours, but then provides better results on the long run.

As a consequence of this observation, which needs additional runs in order to be statistically significant, it was decided to keep the default skip probability close to 0%.

Chapter 6

Conclusion

The first important result confirmed by this document is that the Steelix heuristic is useful for the purpose it was designed for. Indeed, on the LAVA-M binaries it gives the fuzzer the possibility of cracking the statements guarding the injected bugs; when running without it, the fuzzer is not able to overcome them.

The heuristic, however, comes with some limitation attached, mostly related to its applicability: the impossibility of using it when modifying more than one byte at a time limits the ability of the fuzzer to explore diverse inputs quickly. The appropriate solution to this problem would be to use true dynamic taint analysis, which bears the risk of slowing the execution; however, it can still provide good performance when used sparsely, as shown by Angora [6].

As far as the implementation presented in this document is concerned, the choice of using dynamic binary instrumentation proved to be right. Despite the fact that the technique is slower on paper, the optimizations performed by QBDI and the introduction of the double queue structure were able to provide results at least comparable to the original implementation, despite running at lower speeds. In addition, the instrumentation is less invasive and generates fewer risks of introducing inconsistencies in the original software.

In terms of pure coverage, instead, the advantage provided by the heuristic is limited even on parsing code, as shown by ImageIO. This advantage, however, is consistent on longer runs and compensates the overhead introduced for the execution of the instrumentation code. As a consequence, the heuristic can be deemed useful also on real software, not only on synthetic suites as LAVA-M.

With regard to the parameters that the implementation presented in this document offers, namely the size of the shared memory area and the probability to be assigned to the comparison progress queue, no complete conclusion can be drawn. The data collected show that bloom filters smaller than 32 KiB and probabilities higher than 10% tend to reduce the performance of the fuzzer, but a more thorough analysis is needed in order to point out the best values for those two parameters.

Bibliography

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. Announcing OSS-Fuzz: Continuous fuzzing for open source software. `https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html`, 2016.

[2] Frederic Besler. Circumventing fuzzing roadblocks with compiler transformations. `https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/`, 2016. [Online; accessed 27-July-2018].

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[4] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[5] Bryan Buck and Jeffrey K Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[6] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307*, 2018.

[7] Wikipedia contributors. Fuzzing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Fuzzing`, 2018. [Online; accessed 23-July-2018].

[8] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 110–121. IEEE, 2016.

[9] Marcus Geelnard. Benchmarking os primitives. `http://www.bitsnbites.eu/benchmarking-os-primitives/`, 2017. [Online; accessed 08-August-2018].

[10] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[11] Hex-Rays. IDAPython. `https://www.hex-rays.com/products/ida/support/idapython_docs/`. [Online; accessed 06-August-2018].

[12] Charles Hubain and Cédric Tessier. Implementing an LLVM based dynamic binary instrumentation framework. `https://media.ccc.de/v/34c3-9006-implementing_an_llvm_based_dynamic_binary_instrumentation_framework`, 2017. [Online; accessed 17-August-2018].

[13] Apple Inc. Image I/O programming guide. `https://developer.apple.com/library/archive/documentation/GraphicsImaging/Conceptual/ImageIOGuide/imageio_intro/ikpg_intro.html`, 2016. [Online; accessed 23-August-2018].

[14] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

[15] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[16] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637. ACM, 2017.

[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[18] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[19] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[20] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation, 2018.

[21] Quarkslab. QuarkslaB Dynamic binary Instrumentation. `https://qbdi.quarkslab.com/`, 2017. [Online; accessed 22-August-2018].

[22] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[23] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Address-sanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[24] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[25] Michał Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2014. [Online; accessed 23-July-2018].

[26] Michał Zalewski. Technical "whitepaper" for afl-fuzz. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2014. [Online; accessed 17-August-2018].

[27] Michał Zalewski. Symbolic execution in vuln research. `https://lcamtuf.blogspot.com/2015/02/symbolic-execution-in-vuln-research.html`, 2015. [Online; accessed 23-July-2018].