A QBDI-based Fuzzer Taming Magic Bytes

Elia Geretto[1], Cédric Tessier[2], and Fabio Massacci[1]

[1] University of Trento, Trento, Italy
elia.geretto@alumni.unitn.it
fabio.massacci@unitn.it
[2] Quarkslab, Paris, France
ctessier@quarkslab.com

Abstract. One of the major limitations of mutation-based grey-box fuzzers is that they struggle in accessing code protected by magic bytes comparisons, which are routinely employed by parsers. The best solution to this problem, the Steelix heuristic, proposes an implementation based on static binary instrumentation. This work demonstrates that, by using instead dynamic binary instrumentation, it is possible to obtain comparable performance and gain advantages in terms of precision and flexibility of the instrumentation. We have demonstrated the feasibility of this approach both on a standard academic benchmark, LAVA-M, and on real-life large-scale software, using the macOS framework ImageIO.

Keywords: Binary fuzzing · Binary instrumentation · Coverage-based fuzzing

1 Introduction

Fuzzing is among the automated software testing techniques that enjoyed the most widespread adoption in the last decades: its first application dates back to 1990 [11] and its modern variations, grey-box and mutation-based, have been integrated in the software development process of various large companies [5] and commonly used open source projects [1].

However, fuzzing has significant limitations that might hinder its effectiveness. The most important one is that fuzzers might not be able to reach portions of code that are protected by conditions which have a low probability of being guessed providing random inputs. The cause is that these tools generate the test cases that are fed to the target program using mutation operators that do not take into account the structure of the code.

The problem of reaching code protected by complex conditions translates into a variety of concrete problems that are commonly found, for example, when testing parsers. Indeed, they employ conditions which require that a sequence of adjacent input bytes matches a specific sequence of values, called magic bytes, in order to access a given code branch. The inability to solve such conditions can hinder the penetration power of a fuzzer [9]; this makes efficiently dealing with magic bytes a major open problem.

Several works address this issue, but most of them target the general problem of difficult branch conditions, as in the case of Driller [15] and T-Fuzz [12], and thus are unnecessarily computationally expensive when considering only magic bytes. The best solution for this particular problem is the heuristic implemented in Steelix [9], which is based on the unrolling of comparison instructions and is able to provide good results while generating a limited overhead.

The implementation proposed for Steelix, however, has several limitations that this work tries to tackle while preserving the underlying idea. The main one is that it relies on static binary instrumentation: despite being faster than the alternatives, it requires the use of static analysis, which is often imprecise.

This work proposes instead a new implementation based on dynamic binary instrumentation, through the use of the QuarkslaB Dynamic binary Instrumentation framework (QBDI) [6]; this choice increases the precision of the instrumentation, since the use of static analysis is completely absent, but imposes lower execution speeds. To compensate for this, the original algorithm was modified to help the fuzzer in solving the conditions faster through the introduction of a double queue architecture.

The results obtained when comparing the original Steelix implementation with the tool presented in this document show that it provides comparable performance proving that the use of dynamic binary instrumentation in the context of fuzzing can be effective. This result can be leveraged to explore the use of other dynamic techniques, such as dynamic taint tracking, in order to improve the quality of the mutation operators used in fuzzers.

In addition, testing was also conducted to assess the scalability of the new implementation when considering more complex software. The results show that the overhead introduced by the heuristic is low and is quickly compensated by the advantages obtained in terms of coverage.

2 Problem Statement

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

Compartment model. Software can be seen as a set of communicating "compartments": branch conditions that check for a very restricted set of values in a specific input split the application into two different compartments [15]. Fuzzing 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. However, it struggles in making the execution flow from one compartment to another. This happens because conditions that have a large search space and a really small set of satisfying solutions are nearly impossible to guess randomly. As a consequence, 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 unlikely explored afterwards.

Difficult branch conditions. Considering the branch conditions that usually block grey-box fuzzers, the following subdivision can be made: those that can be easily solved with a satisfiability modulo theories (SMT) solver, as equality conditions, and those for which these tools are ineffective, as non-linear relations or checks on the results of cryptographic hash functions.

The first category is way more common and thus several solutions that employ, or imitate, SMT solvers have been presented in the literature; this is the case for Driller [15] and Angora [3]. If conditions cannot be dealt with using solvers, the only proposed solution is the one implemented in T-Fuzz [12]: patch[3] difficult conditions one by one in order to force the fuzzer to explore the portions of the program which were not accessible in the original binary.

However, both categories require the use of computationally expensive techniques, as symbolic execution or dynamic taint tracking. These might hinder the ability of the fuzzer to explore new portions of code due to the overhead in the generation of new test cases. As the most common application of fuzzers is testing parsing code, it is likely that solutions focusing on a smaller problem and employing more lightweight techniques will provide increased performance.

Magic bytes comparisons. When considering parsing code, fuzzers struggle to explore areas that are protected by branch conditions which verify magic byte sequences. These are commonly used to check the format or the version of a data stream, but are also employed to separate different sections within the same stream.[4] Examples of these entities are the `PNG` magic sequence and the identifiers for "chunk types", both present in PNG images.

In detail, magic bytes conditions check that a series of adjacent input bytes corresponds to a precise sequence of values and, if this is not the case, make the program take another path. Mutation-based grey-box fuzzers struggle with such constructs since they do not have any notion of what is composing a single branch condition: they receive feedback only regarding whether it is entirely satisfied or not, without the possibility of observing progress in solving it.

This limitation has been tackled by three different tools: AFL-lafintel [2], Steelix [9] and VUzzer [14]. Due to their relevance for this work, Steelix and AFL-lafintel are analyzed separately in Section 3. VUzzer, instead, proposes a solution based on dynamic taint tracking; unfortunately, this technique, even if computationally cheaper than symbolic execution, is still slow when compared with the instrumentation code proposed by Steelix.

3 Steelix Approach and Limitations

The main goal targeted by the authors of Steelix [9] was 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.

[3] In this case, "patching" refers to the process of modifying a binary file in order to alter its intended behavior.

[4] In the second case, they can also be defined as "tokens". This definition is broad on purpose.

Its fundamental idea is to transform each condition involving magic byte sequences into a series of conditions involving single bytes, effectively unrolling the comparison. This technique 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 smaller. This technique was first presented by the authors of AFL-lafintel [2].

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 of that byte, exploring all the possible 256 values.

The individuation of the next byte to mutate is achieved simply supposing that the next byte to be matched is either the one following, or the one preceding, the byte that was last mutated 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 complete example that illustrates how the heuristic is applied can be found in the original paper [9].

Limitation in the instrumentation code. The original implementation of Steelix injects the tracing callbacks in the software under test using static binary instrumentation. This technique requires the knowledge of all the locations to be instrumented before the execution of the program. As a consequence, they can only be listed through the use of static analysis.

However, static analysis may notoriously produce incomplete results when trying to extract specific features of the code, such as a complete control flow graph (CFG). For it to fail, it is sufficient for the analysis to encounter an indirect jump whose target is computed at runtime, as it happens for C++ virtual methods. While it never happens targeting single instructions, it is a common problem when trying to isolate all the basic blocks in a program, as it is required by AFL-style instrumentation.

The only way around this issue is to use dynamic binary instrumentation (DBI), so that the locations to be instrumented can be determined at runtime. This allows the DBI framework to extract the control flow features at runtime too, allowing to correctly report every basic block. The main drawback is that the execution speed is reduced; understanding the magnitude of this degradation, and thus the convenience of this change, was the main goal of this study.

Limitation in the taint tracking heuristic. No taint tracking mechanism is present since the authors argue that canonical solutions are too heavyweight to be employed in a fuzzer. As a consequence, they simulate the result of taint tracking making the following two assumptions: when a comparison progress is reported, the last mutated byte generated the progress and its neighbors are candidates to be part of the same magic bytes sequence.

The problem with these assumptions is that they force the fuzzer to mutate only one byte per execution. Indeed, if more than one byte is mutated, it is impossible to understand which is the byte that generated a new 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.

This issue is disruptive since it forces the implementer to either disable the heuristic when mutating more than one byte per test case or to limit the ability of the fuzzer to produce diverse input to keep the heuristic constantly enabled.

4 Implementation

The alternative implementation of the Steelix heuristic presented in this work was obtained starting from a pre-existing tool, called AFL/QBDI, which was then modified in order to integrate the Steelix heuristic in it.

Pre-existing tool. The starting point for this work, AFL/QBDI, is a mutation-based grey-box fuzzer that injects tracing instrumentation code whose output is compatible with American Fuzzy Lop (AFL) [16]. This design choice was made so that the fuzzer program belonging to that project could be used to handle the fuzzing process. As a consequence, the internal logic of the fuzzer is the same as AFL; the method with which the instrumentation is inserted is the only part that was changed.

The instrumentation is performed through QBDI [13], a dynamic binary instrumentation framework similar to PIN [10]. The goal of this design choice was to allow the fuzzer to test closed-source binaries while still preserving good performance: QBDI is obviously not as fast as the source instrumentation employed by AFL, but provides better performance than that obtained with AFL-QEMU.

Comparison progress reporting. The first modification introduced was to add comparison unrolling callbacks for CMP and TEST instructions. The recording of the bytes that form a single comparison is performed exactly as the recording of the execution trace: when a byte in a CMP or TEST instruction is matched, the address of the instruction is used to generate an index which is then used to increment a counter byte in a bloom filter.

As opposed to Steelix, which seems to mix execution trace and comparison progresses in the same filter, AFL/QBDI uses two separate bloom filters with the same size, thus employing two times as much memory. This change does not appear to influence performance, at least when considering the execution speeds imposed by dynamic binary instrumentation. The two memory areas are obviously parsed separately by the fuzzer once a single execution is finished.

An important technical limitation that needs to be highlighted is that, when considering a single instruction, QBDI currently allows to access the runtime values of register operands only, the results of memory accesses are not available. Instructions with such accesses are less common in optimized code, but they are still present; when encountered, they can only be ignored.

Apart from CMP and TEST instructions, the original implementation also hooks `strcmp`, `strncmp` and `memcmp` in order to apply the same unrolling technique to their arguments. However, that component was not integrated in this implementation and is left as future work.

Queue handling. The second modification that was implemented is related to the processing of test cases. This modification slightly deviates from what the Steelix heuristic prescribes, but it allows to obtain better performance.

In detail, test cases that produced new coverage and test cases that generated a new comparison progress are stored in two different queues, instead of being appended to the same one as in Steelix. When a new test case to be mutated needs to be selected, this change allows to give a higher priority to the comparison progress queue as opposed to the new coverage one.

On the other hand, if a single queue is used, a considerable amount of time elapses between the moment in which the first byte of a comparison is matched and the one in which the last one is. Indeed, 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 match the whole condition. As a consequence, mutating comparison progress test cases first allows to crack a whole condition without interacting with the new coverage queue. This modification is important since, as it is explained later, comparison progress test cases use only one mutation operator and thus are processed faster then normal ones.

The priority given to comparison progress test cases is not absolute: there is a certain probability that the fuzzer will jump directly to the new coverage queue. This probability was introduced in order to avoid starvation in edge scenarios, but it was kept low (1%) since higher values generated performance drops.

In addition, the heuristic states that comparison progress test cases 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 new coverage queue. Queueing sterile comparison progress test cases to the new coverage queue can, in the worst case scenario, flood the new coverage queue. However, this also holds for what Steelix does and it is, in practice, quite unlikely.

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, preserving the rest of the input. Its use is equivalent to the one made by Steelix.

In detail, when a comparison progress is produced, the last mutated byte is recorded, if this is supported by the current mutation operator. 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 will be mutated is determined using 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 one of the two neighbors will need to be fully mutated, allowing to process the test case even faster.

5 Target Binaries

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

LAVA-M. This suite was selected since it was used in the evaluation of Steelix and it was the one for which the authors published the largest amount of information among the tested projects. As a consequence, it is the best basis for the comparison between Steelix and AFL/QBDI.

The first thing that is important to note is that this suite was automatically generated and thus is synthetic, even if it closely resembles real software. It was produced running a fault injection tool called LAVA [4] 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` and were compiled for x86_64, which is the only architecture fully supported by QBDI at the moment. In addition, a default seed for the fuzzing process is provided for each of them.

One of the factors that make LAVA-M distant from real software is that the bugs introduced by LAVA tend to be similar: they all contain out-of-bounds accesses which are generated by adding a random offset to an existing pointer. In addition, they are all protected by similar guard conditions, which compare four adjacent bytes from the input with an integer. These conditions match precisely the problem the Steelix heuristic aims at solving.

It is important to note that the test suite provided with LAVA-M was not fully functioning when executed using QBDI. The reason is that the framework aims at replicating the defined behavior of software, but makes no guarantees regarding the undefined one. In particular, there were differences in the amount of crashes that the test suite was able to reproduce, as shown in Table 1.

Table 1. Results of LAVA-M test suite in QBDI

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

ImageIO framework. The second target selected for the evaluation is the ImageIO framework [7] provided by Apple as part of macOS. It was selected since it allows to evaluate the magnitude of the overhead the new implementation of the Steelix heuristic introduces on large and complex software. 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 contains a collection of parsers for various image formats; its goal is to hide from the user the complexity of detecting the format of the image and then forwarding it to the correct library. The target was the function `CGImageSourceCreateImageAtIndex`, which was fuzzed through a library wrapper that simply read the raw data from a file and fed it to the function.

With regard to the seed inputs used, the fuzzer was provided 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 among those supported by ImageIO.

Since there were not any bugs known before the beginning of the evaluation, the best metric to perform it was cumulative edge coverage. Indeed, its improvement helps in increasing the likelihood of encountering bugs [17].

6 Evaluation

The following paragraphs provide a description of the results of the evaluation conducted on LAVA-M and ImageIO. In the first case, AFL/QBDI is evaluated against the data published by the authors of Steelix on their website; in the second case, AFL/QBDI with the Steelix heuristic enabled is compared against a version in which the heuristic was disabled.

LAVA-M. While the experiment has been run 10 times using 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. As a consequence, every plot in Figure 1 shows the best run obtained by Steelix as a single line and aggregates instead the results obtained by AFL/QBDI following the guidelines presented in [8]: the plots show a 95% confidence interval for a median, calculated following the same guidelines. In addition to that, the plots contain two dashed lines representing the maximum and minimum results obtained.

An evaluation between AFL/QBDI with and without the Steelix heuristic was also performed; 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. This fact proves that the presence of a heuristic targeting magic bytes is necessary to obtain good results in this benchmark.

The results prove that the two implementations provide comparable results despite having considerably different execution speeds: AFL/QBDI performed better in `md5sum` and `uniq`, while in `base64` and `who` Steelix prevailed. In detail, the worse results obtained in `base64` and `who` can be attributed to two

main causes: first of all, if a test case that is fundamental for the process can be produced only with a non deterministic mutation, the faster the execution speed is, the sooner it will be produced. Secondly, due to technical constraints, AFL/QBDI is missing the possibility to unroll comparison instructions containing memory accesses and comparison functions; areas of the program protected by such constructs are unlikely to be explored as a result.

The test was conducted 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. Following the guidelines provided by the authors of LAVA, each single fuzzing experiment performed on the four binaries was run for 5 hours.

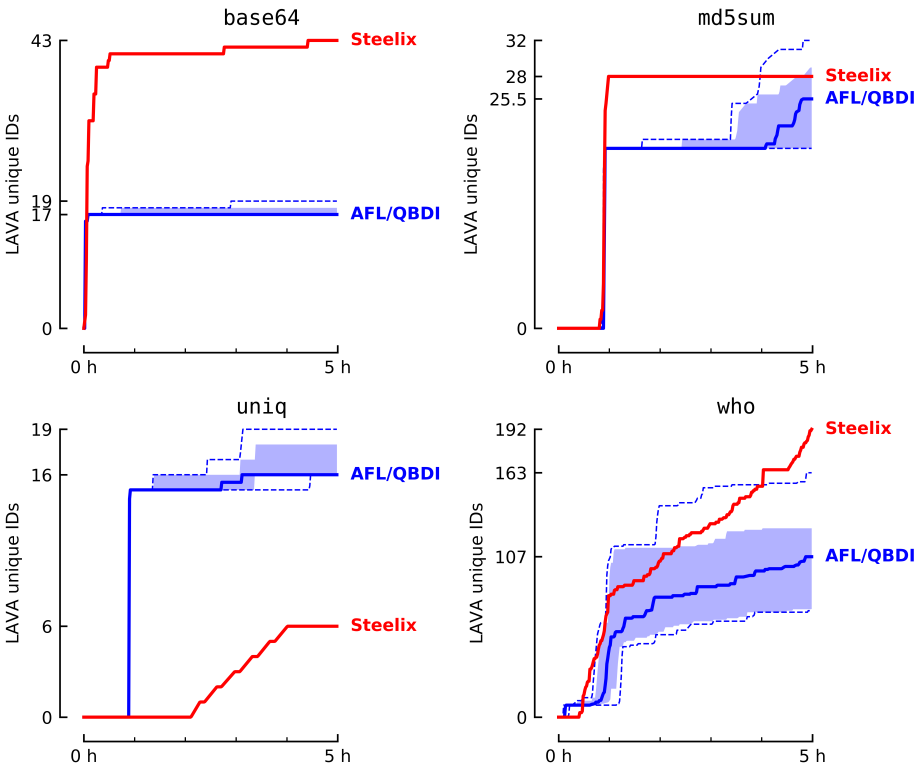

Fig. 1. LAVA-M evaluation comparing AFL/QBDI and Steelix

ImageIO framework. The results of this evaluation are displayed in Figure 2; the aggregation of the 10 runs recorded is performed, as before, following the guidelines presented in [8] even if, in this case, further discussion is needed in order to properly illustrate the results obtained.

The first thing that can be noted is that the experiments with the heuristic disabled have a behavior that is more consistent than the ones with the heuristic enabled. In detail, the median value proves to be better when the heuristic is disabled, but the confidence intervals indicate that there is still a significant probability of obtaining better performance when the heuristic is enabled.

Examining the single runs recorded with the heuristic enabled, it is evident that they 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. Investigating the reason of this difference in behavior, it became evident that it is generated by one single sample which, if not present, does not allow the fuzzer to explore a specific portion of the program. Comparing the shapes of the median progressions with and without the heuristic, it is also evident that that particular sample is always generated when the heuristic is disabled.

The probable explanation 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 new coverage queue; probably, one of them may enter the AFL preferred set in place of the sample that generates the incriminated test case through the deterministic mutation referenced before. 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 corpus formed by different initial seeds, the final result would be more stable and in favour of AFL/QBDI with the heuristic enabled.

Considering now the overhead introduced by the heuristic, the delay in the exploration is quite evident for the first 5 hours but, after that, it is compensated by the increase in coverage provided by the heuristic. Assuming that the explanation given before for the large confidence interval is correct, this conclusion supposes that sterile test cases do not interfere with the preferred set.

Given that 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. Given the large dimension of the software, each experiment was run for 24 hours in order to improve the stability of the coverage.

7 Conclusion

This paper presented a new implementation of the heuristic proposed by the authors of Steelix, which was named AFL/QBDI. The goal was to preserve the ability to tame magic bytes conditions while preserving performance and providing higher flexibility and reliability. This was made possible through the use of dynamic binary instrumentation, which completely eliminated the need for static analysis in the fuzzing process. The tool presented in this work was evaluated using LAVA-M and ImageIO as target binaries.

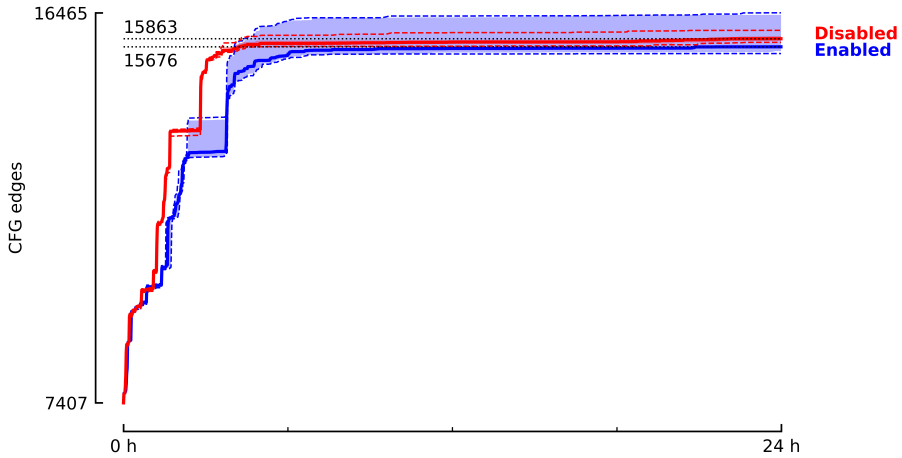

Fig. 2. Cumulative edge count over time on ImageIO

The first important result is that 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 which are at least comparable to the original implementation, despite running at lower speeds. In addition, the instrumentation is more precise since it does not rely on static analysis and thus avoids the risk of missing some basic blocks.

The second conclusion which can be drawn is that, as shown in ImageIO, the new implementation is able to scale also on complex software. Indeed, it introduces only a small overhead which is then compensated in a few hours by the advantages it provides in terms of edge coverage.

The heuristic, however, comes with some limitations 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 down the execution; however, the successful use of dynamic binary instrumentation suggests that the application of this other dynamic technique may be possible.

Acknowledgments

This work has been partially supported by the European Institute of Innovation and Technology - EIT Digital through task T16448 S&P Education, supervising a MSc dissertation in cooperation with Quarkslab, where the first author has been an intern on this subject.

References

1. Aizatsky, M., Serebryany, K., Chang, O., Arya, A., Whittaker, M.: Announcing OSS-Fuzz: Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html (2016)
2. Besler, F.: 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. Chen, P., Chen, H.: Angora: Efficient fuzzing by principled search. arXiv preprint arXiv:1803.01307 (2018)
4. Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R.: Lava: Large-scale automated vulnerability addition. In: Security and Privacy (SP), 2016 IEEE Symposium on. pp. 110–121. IEEE (2016)
5. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Queue **10**(1), 20 (2012)
6. Hubain, C., Tessier, C.: 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]
7. Inc., A.: 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]
8. Klees, G.T., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (Oct 2018)
9. Li, Y., Chen, B., Chandramohan, M., Lin, S.W., Liu, Y., Tiu, A.: Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 627–637. ACM (2017)
10. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Acm sigplan notices. vol. 40, pp. 190–200. ACM (2005)
11. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the ACM **33**(12), 32–44 (1990)
12. Peng, H., Shoshitaishvili, Y., Payer, M.: T-fuzz: fuzzing by program transformation (2018)
13. Quarkslab: QuarkslaB Dynamic binary Instrumentation. https://qbdi.quarkslab.com/ (2017), [Online; accessed 22-August-2018]
14. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: Application-aware evolutionary fuzzing. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2017)
15. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. vol. 16, pp. 1–16 (2016)
16. Zalewski, M.: American fuzzy lop. http://lcamtuf.coredump.cx/afl/ (2014), [Online; accessed 23-July-2018]
17. Zalewski, M.: Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt (2014), [Online; accessed 17-August-2018]