Reducing Time-To-Fix For Fuzzer Bugs

Rui Abreu*, Franjo Ivančić†, Filip Nikšić†, Hadi Ravanbakhsh†, Ramesh Viswanathan†
*University of Porto, Porto, Portugal
†Google, Inc., New York City, NY, USA

***Abstract*—At Google, fuzzing C and C++ libraries has discovered tens of thousands of security and robustness bugs. However, these bugs are often reported much after they were first introduced. In many cases, developers are provided only with fault-inducing test inputs and replication instructions that highlight a crash, but additional debugging information may be needed to localize the cause of the bug. Hence, developers need to spend substantial time debugging the code and identifying commits that introduced the bug. In this paper, we discuss our experience with automating a fuzzing-enabled bisection that pinpoints the commit in which the crash first manifests itself. This ultimately reduces the time critical bugs stay open in our code base. We report on our experience over the past 12 months, which shows that developers fix bugs on average 2.23 times faster when aided by this automated analysis.**

I. Introduction

Fuzzing has emerged as one of the most effective testing techniques for discovering security vulnerabilities and reliability issues in software. The idea behind fuzzing is simple: the fuzzer executes programs with randomly generated inputs, and monitors them for invalid behavior, such as crashes, memory corruption, and internal assertion violations. Recent advancements in fuzzing technologies such as coverage-guided fuzzing [1], [2], combined with compiler instrumentation such as LLVM [3] sanitizers [4], have enabled fuzzing to reach deep program paths and uncover significantly more bugs.

The success of fuzzing has led to its widespread adoption in the industry, most notably through the emergence of services that provide continuous fuzzing for open-source and commercial software. For example, Google has developed continuous fuzzing infrastructures both for its internal software and for external open-source projects. As of April 2021, Google's ClusterFuzz project [5], through its OSS-Fuzz instance [6], has alone filed nearly 30,000 bugs to developers by fuzzing over 340 open-source projects [7]. Recently, in addition to fuzzing C/C++, OSS-Fuzz has also expanded to continuously test programs written in Go, Python, Rust, and Java.

The continuous fuzzing infrastructure at Google is integrated in the software engineering development workflow. Software engineers utilize semi-automated frameworks such as FUDGE [8] to own fuzz targets that are checked into Google's monolithic repository [9]. Fuzzing infrastructures have converged to a minimal interface between fuzzers and fuzz drivers, which was introduced by libFuzzer [1].

When a new bug is found, the fuzzing infrastructure creates a bug report for the relevant team that owns the fuzz target. The bug report contains details such as the test input that causes the bug, which compilation mode was used to find the bug, and one-click reproduction instructions for developers to investigate the issue.

Despite the information provided in the report, investigating the issue and localizing the cause of the bug may still be difficult and time consuming. The bug is often reported days, months, and in extreme cases even years after being introduced into the code base. During this time, Google's monolithic repository goes through a large number of changes. Even though not all changes are relevant to the bug, the cause may be hidden in a transitive dependency with which the fuzz target's owner has little experience.

To aid the debugging, we have developed an automated fuzzing-enabled bisection service that pinpoints code changes likely to be relevant to the bug. In this paper we report on our experience with the service over the last 12 months. In particular, we observe that providing the code-change bisection information speeds up fixing fuzzer-reported bugs in Google's proprietary code on average by a factor of 2.23.

II. Continuous Fuzzing at Google

Fuzzing is often used in the context of discovering security issues, such as memory-unsafety bugs. Recently, there has also been interest in using fuzzing to test for other properties as well, such as reliability, performance or functional requirements. As an example, there are fuzz targets that aim to test multiple implementations or versions against each other to find discrepancies using differential testing [10].

Coverage-guided Fuzzing. Fuzzing infrastructures, such as OSS-Fuzz [6], generally utilize multiple so-called *coverage-guided fuzzing engines*, such as AFL [2], libFuzzer [1], Honggfuzz [11], and AFL++ [12]. These fuzzing engines collect *coverage feedback* from an instrumented version of the source code under test. For each input, the collected coverage is compared to the combined coverage profile obtained during fuzzing so far. New test inputs that induce an increase in coverage are then added to the current *corpus* of coverage-increasing tests. Such test inputs are then favored as seeds during the next test input mutation stage.

Fuzz Targets. Fuzzing infrastructures have converged to use the so-called `LLVMFuzzerTestOneInput` interface. This interface and the notion of library-based *fuzz targets* was initially introduced by libFuzzer [1]. Fuzz targets receive an input buffer generated by a fuzzing engine through a sized buffer argument and use it to invoke some relevant functionality of the targeted code. For example, in Listing 1, the input buffer `data` consisting of `size` number of bytes is fed it into the OpenCV datatype (`cv::Mat`).

Listing 1 A fuzz target for OpenCV [13].

```
int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size){
  std::vector<uint8_t> arr = {data, data + size};
  cv::Mat row = cv::Mat(1, arr.size(), CV_8UC1, arr.data());
  try {
    cv::Mat m = cv::imdecode(row, CV_LOAD_IMAGE_UNCHANGED);
  } catch (cv::Exception e) { }
  return 0;
}
```

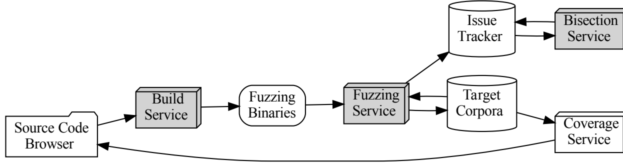

Fig. 1. Continuous fuzzing developer workflow

Fuzzing Workflow. Figure 1 highlights the life cycle of a fuzz target. Fuzzing, due to its nondeterministic nature, does not guarantee that it can find regressions quickly. Thus, a dedicated infrastructure is advantageous: It rebuilds every fuzz target in multiple configurations and for multiple fuzzing engines once per day. The generated drivers are then used until the next rebuild on dedicated sandboxed distributed infrastructure to look for coverage-increasing tests and new bugs.

Corpus Handling. The fuzzing infrastructure chooses randomly among the generated fuzzing executables. At the end of each invocation of a fuzzing executable, it collects the generated *corpus* of discovered test inputs and potential *artifacts*. Artifacts are those test inputs that caused the fuzzing executable to report any suspicious behavior, such as inducing a crash, assertion violation, triggering a memory leak, etc.

The corpus that was collected from a single invocation is then added via a *minimization* step to the global corpus for that fuzz target. That is, for each target a globally coverage-increasing corpus is maintained by the fuzzing infrastructure. Newly discovered test inputs that increase this global coverage are added to the global corpus.

Crash Deduplication. The goal of fuzzing is not just to find bugs, but rather to have the reported bugs fixed. In order to do so, it is of utmost importance to make sure that developers are notified when necessary but not unduly inconvenienced by these notifications. Thus, to make the fuzzing output useful and actionable, we try to group as many found crashes as possible together as related bugs and only report the most security-sensitive version of a group to the user as a new bug. To form groups of crashes, we compute a representative by choosing relevant portions of the stack trace of a failing execution.

Automated Bug Monitoring. Since bugs reported by the fuzzing infrastructure are reproducible, the infrastructure can monitor the status of previously filed bugs. Thus, as daily rebuilds of fuzz targets are completed, a crash reproduction analysis is performed. Test inputs that fail to induce a crash by the fuzz target are marked as resolved (regardless of whether a developer already marked the bug as fixed). Similarly, if a developer closed a bug that the fuzzing infrastructure continues to reproduce as a crash, the bug will be reopened for the developers to investigate again.

Fuzzing Coverage. As previously mentioned, the fuzzers are running in a distributed fashion, and collaboratively build up a corpus for a given fuzz target. Developers are often interested to monitor the combined *fuzzing coverage*. Thus, we provide multiple ways of inspecting the obtained coverage, e.g. overlayed within our codesearch browsing tool [14].

Shift-left on Fuzzing. Developers are generally likelier to fix code issues if they are highlighted during code review than after submission [15]. However, fuzzing is expensive and may take a long time to find certain bugs. Thus, we developed a *presubmit fuzzing* or *CI fuzzing* [16] framework that executes shallow and fast fuzzing runs for targets potentially impacted by code changes under review. If a target crashes with the proposed code changes, but it does not crash without them, the finding is reported to developers during code review.

III. Approach to Automated Bisection

The fuzzing infrastructure automatically deduplicates crashes, files bugs for newly discovered issues in an issue tracker and monitors these issues until the crash is shown to be fixed. This automation helps developers to focus on the important, remaining open issues that need to be fixed. However, so far, developers were asked to own all aspects of the debugging process. In this section, we describe a simple yet effective automation capability that helps developers fix issues faster by pointing out relevant code changes that have an impact on the found crashing inputs.

Since fuzzer reported bugs are easily reproducible, we added the capability to automatically perform a bisection on code changes to discover the likely fault-inducing change. This capability was earlier advertised to users in the created issues. The main feature is that the fuzzing infrastructure performs this analysis automatically for newly opened bugs and reports its findings on the opened issues. Even so, we observed that this feature significantly decreased the time it takes to fix bugs—simply due to the automation.

Automated Bisection. The bisection algorithm is relatively straightforward. Due to the size of Google's monorepository [9], the bisection first tries to find a code change within the project of interest. That is, we limit the bisection search based on the fuzz target that is being investigated. This might seem like an optimization only, but allows the search to focus on relevant code changes and not potential unrelated failures outside the horizon of interest for the team that will try to fix the issue in question. If that analysis fails to identify a possible fault-inducing code change, the analysis falls back to do a complete bisection over a predefined time period.

Issue Tracker Notification. Once a bisected code change has been identified, the tool appends a message to the relevant issue in the issue tracker, including the following information:

- The tool reports that the bisected code change was found relevant to the issue.

- The tool displays the failing (i.e. crashing) reproduction run at the bisected code change. It also shows the successful (i.e. non-crashing) reproduction run with the same input at the previous code change.
- The tool adds the author of the bisected code change as CC to the issue. Crucially, it does not assign the issue to the author. This is partly due to the fact that Google's mono-repository allows widespread code sharing. Thus, the author of the bisected code change may not have the requisite knowledge about the fuzz target of another team relying on some library that they provide.
- If the bisected code change relates to changes in our so-called `//third_party` open-source repository [17], the tool highlights this information in the update message as well. Changes in `//third_party` are often large-scale updates from public repositories.
- If the bisected code change includes changes to the fuzzer itself, this is highlighted in the update message. Changes to the fuzzing harness are often intended to cover previously uncovered code behaviors and may as such intentionally cause new bugs to be found.
- The update to the issue also contains instructions how to provide feedback in case developers disagree with the classification. This has helped us resolve some issues as well as tailor the issue update messages. For example, highlighting that a change was made in `//third_party` as well as being cognizant of changes to the fuzz harness itself were due to earlier developer feedback.

Avoiding Fuzzing Efficacy-related Properties. Fuzzing engines and fuzzing infrastructures often also report fuzzing efficacy issues to developers. For example, if an input causes a target to run too slowly (using a fixed threshold such as 20 seconds), a *time-out* issue is filed. Similarly, if an input causes a target to allocate too much memory, an *out-of-memory* issue is filed. These issues may showcase performance issues in the code under test. For example, we have seen instances where fixes involved improving the runtime complexity of nested loops. These issues also impede the fuzzers from making efficient progress, and as such it is useful for the fuzzing infrastructure to report this feedback to developers.

However, trying to perform a bisection analysis at the level of code changes is hard for such scenarios for a number of reasons: First, execution runtimes and allocation behavior can vary between different invocations due to a variety of unrelated factors. Further, while it may be possible to find a code change that triggers the increased runtime to cross our predefined threshold, it is not evident that such a code change is useful to consider as the *cause* of a performance concern. Imagine a scenario where a prior code change increased the runtime to within 95% of the timeout threshold, while a final code change added the remaining 5%. Identifying the second code change as the bisection output due to the fact that it caused the threshold to be reached seems appropriate from the bisection analysis point of view, but developers may not see the minor performance increase as a valid concern. Thus, we stopped reporting bisection results for issues related to *time-outs* and *out-of-memory* bugs due to developer feedback.

IV. Usage Experience

In this section, we present evaluation results of the automated bisection capability for the past 12 months. The bisection tool runs as a best-effort analysis that only analyzes new bugs as they are reported by the fuzzing infrastructure. As of September 2020 [18], the fuzzing infrastructure fuzzes several thousand fuzz targets on 30,000 VMs. Fuzzing at Google has reported tens of thousands of bugs across various ecosystems.

Furthermore, we restrict reported findings to go back to code changes that are at most six months old from the bug-filing date. We do not report bisected code changes that are purely due to changes to the fuzz target such as adding a new fuzz target, for example. While it would be technically correct to report such a code change as the bisected code change, it does not actually point developers to the potential underlying root cause of a reported bug.

In the following, we present data for the following questions of interest based on 12 months of running the automated bisection for fuzzer-reported bugs in production:

- Does providing the bisected code change along with the reported test input improve the speed of fixing the bugs?
- Do bugs with bisected code changes get fixed more often than other bugs for various time intervals of interest?
- Does the fix rate improvement for fuzzer-reported bugs depend on whether the fuzz targets are in Google's proprietary code compared to targets covering `//third_party`?

Time-to-fix Improvements. We have monitored the time it takes to fix fuzzer-reported bugs since introducing the automated bisection over the past 12 months. Bugs filed against fuzz targets in our proprietary code have been fixed on average 2.23 times faster over the past year. One unexpected side-benefit of the automated bisection is that we observe an even more pronounced improvement in declaring duplicate bugs: Bisected bugs filed in the past five months have been marked as duplicates 4 times faster than non-bisected bugs.

Fix Rate Improvements. The previous analysis investigated the speed-up of fixing known fuzzer-reported bugs. While fixing bugs faster is important to address newly discovered issues, it is not evident whether this also improves how many bugs get fixed when looking at longer time-periods. That is, does the analysis only improve the speed of fixing these bugs or does it also increase the fixed bug count?

To answer this question, we investigate bugs reported by the fuzzing service over the past 12 months. Figures 2 and 3 show the fix rate improvement for filed bugs with bisection results compared to bugs without such additional reports. Bugs filed in the past 12 months are bucketed by the month in which they were reported in the issue tracker. We then observe the bug fix rate with and without bisection reports for each month. Obviously, for older bugs, developers had more time to prioritize fixing such reported bugs.

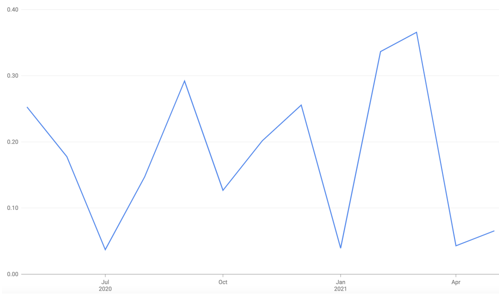

Fig. 2. Fix rate improvement for Google's proprietary targets.

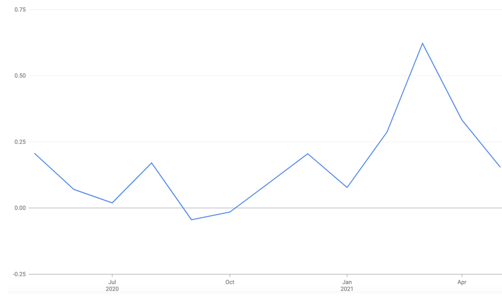

Fig. 3. Fix rate improvement across all fuzz targets

In Figure 2, we first consider only those bugs filed on fuzz targets in non-`//third_party` code—that is in Google's proprietary code. As can be seen, the improvement in the fix rate ranges from 4% to 37% for every month in the past 12 year. For bugs reported in the past four months, we can see a significantly larger improvement in the fix rate. Over time, as bugs age, the effect of the previously discussed shorter time to fix bugs decreases somewhat. However, evidently, the bisection reports have an impact on overall bug fixing rates even beyond that initial time-period. This can be seen from the fact that for every single month, bugs with bisection reports continue to have a positive fix rate improvement—even as this improvement reduces somewhat for older bugs.

Proprietary and `//third_party` Code. Figure 3 shows the bug fix rate improvement for all fuzzer-reported bugs, including ones in `//third_party` projects. The figure highlights the difference between the improvement for more recent bugs compared to older bugs. Overall, the monthly bug fix improvement rate varies from -4% to 62%.

An important distinction between `//third_party` targets and proprietary targets is the familiarity of developers with the source code. Fixes in `//third_party` are often accomplished through re-imports of an open-source package. The fact that we continue to see a significant improvement in recently reported bugs is due to time-to-fix improvements. At the same time, we also highlight a few months with very small improvements, and even one month with a deterioration. This is likely due to predetermined scheduled updates of some large open-source projects. Since our analysis avoids reporting the same code change too frequently, as described in Section III, many bugs not considered as bisected may get fixed by a future combined import.

V. Lessons Learned

This section describes a number of lessons that we learned during the development and usage of our automated bisection notification system for fuzzer-reported bugs.

Lesson 1: Importance of allowing developers to communicate back to infrastructure providers. Continuous fuzzing runs on thousands of fuzz targets across hundreds of projects in a variety of ecosystems, such as OSS-Fuzz [6]. For infrastructure providers, it is not possible to be intimately familiar with all the various use cases that are being supported. Thus, it is important to educate the developers to guarantee that the infrastructure is utilized well. At the same time, it is important to provide avenues for developers to ask questions, raise issues, and provide feedback. We have found multiple bugs in the tooling due to developer feedback.

As discussed in Section III, we also used this feedback channel to improve our communication around bisected code changes. An example of this was how the tooling handled reports with respect to `//third_party` code changes. Another feature improvement driven by user feedback was to understand the difference in bisection results to a property such as a buffer overflow when compared to a runtime performance property such as a target timing out. The former properties lend themselves much easier to a bisection tooling, whereas fuzzing-efficacy related properties require a much deeper investigation to be generally useful.

Lesson 2: Being annoying is not helpful. Bugs in widely used libraries such as protocol buffers [19] can induce failures in many fuzz targets. The fuzzing infrastructure does not deduplicate crashes across targets, however. This is due to the fact that a bug in a low-level library indeed induces a new bug in the client code. Not reporting this as a bug to these teams would be misleading, since they may otherwise decide to deploy their code to production with a bug known to the infrastructure.

In general, we report bisection results by CC-ing code authors to fuzzer-reported issues, even if the fuzz target is not owned by the code author's team. At the same time, it would likely not be useful to notify the code author about *every* instance across the whole mono-repository. There could be hundreds of such bugs across fuzz targets. Thus, we limit the number of times we report the same code change. This allows us to highlight the issue to the code author without unnecessarily notifying them too frequently.

Lesson 3: Not all bisected code changes are considered useful by developers. We have used the developers' feedback to improve the bisection results over time, such as for code changes adding or changing fuzz targets. However, there are also some instances where the bisection is in the code under test, but the benefit of highlighting this code change to users is unclear.

As an example of such a scenario, consider a renaming of a key in a key-value pair in the input to an API such as shown

Listing 2 An API where a renaming might result in bisected code changes that some developers may find irrelevant.

```
#include <map>
#include <string>

int GetSpecialValue(const std::map<std::string, int>& map) {
  const char* special_key = "old_key_name";
  const std::map<std::string, int>::const_iterator it =
      map.find(special_key);
  if (it == map.end()) {
    return -1;
  }
  const int return_value = it->second;
  if (return_value < 0) {
    __builtin_trap();  // An error that we want to catch.
  }
  return return_value;
}
```

in Listing 2. Consider a fuzz target that takes the input buffer and creates an appropriate map to fuzz the `GetSpecialValue` API. Imagine that a developer changes the code to update the variable `special_key` to be renamed from `old_key_name` to `new_key_name`. This is likely going to cause those previously reported bugs that used `old_key_name` in the generated test input to be considered as fixed. At the same time, new bugs would eventually be found with a new input, which would now contain the key name `new_key_name`.

Once the new bug is found and reported to users, the bisection run will consider the renaming of `special_key` as the relevant code change. While this finding is correct from the bisection point of view, it still does not provide much value to developers in terms of debugging.

This type of inadequate finding is rare enough that we have not yet decided to address the concern. However, it does highlight the potential for additional future improvements. For example, one might want to find a prior code change that modulo the refactoring would have been an interesting code change to point out. A similar situation occurs when the bisected code change ends up being a change to or the addition of a fuzz targets. For example, it may be possible to back-propagate the introduction of a fuzz target to find a code change that would have been caught with the given input had the fuzz target existed at that point. These are areas of future improvements for our bisection tooling.

Lesson 4: Automating even simple steps and notifying relevant developers can improve outcomes. As described in Section IV, developers fix more bugs and fix them faster if the automation helps in localizing the issue. As we had mentioned previously, developers were always able to manually run a similar bisection analysis themselves for a given bug. However, the fact that the automated solution provided this feedback to developers proactively caused them to look into the bugs. This is in contrast to the previous state of the continuous fuzzing infrastructure, which filed bugs against project *teams* (not individuals).

Lesson 5: Developers that are owners of projects are instrumental in fixing security bugs. Section IV highlighted the difference in bug fix rate improvement between Google's proprietary and `//third_party` code. Developers working with `//third_party` code do not always have enough background to fix complex bugs. That is, open-source developers and code-owners are in the best position to fix any bugs in their open-source projects. Thus, we are very excited to continue to support fuzzing of open-source projects in OSS-Fuzz [6].

VI. Conclusions

This paper described our experience with adding the capability to automatically bisect fuzzer-reported bugs as part of the fuzzing infrastructure. We report that due to this automation, bugs were fixed on average 2.23 times faster. Finally, we would like to note that the recent OSV database [20] started providing similar capabilities for fuzzer-reported bugs in OSS-Fuzz [6].

References

[1] K. Serebryany, "libfuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html#fuzz-target, 2015.

[2] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl, 2014.

[3] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*. IEEE Computer Society, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[4] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX ATC 2012*, 2012.

[5] A. Arya, O. Chang, M. Moroz, M. Barbella, J. Metzman, and the ClusterFuzz Team, "Open sourcing ClusterFuzz," Google Open Source Blog, February 2019. [Online]. Available: https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html

[6] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker, "Announcing OSS-Fuzz: Continuous fuzzing for open source software," Google Testing Blog, December 2016. [Online]. Available: https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[7] M. Ruhstaller and O. Chang, "A new chapter for OSS-Fuzz," Google Security Blog, November 2018. [Online]. Available: https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html

[8] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "FUDGE: Fuzz driver generation at scale," in *ESEC/FSE 2019*. New York, NY, USA: Association for Computing Machinery, 2019, p. 975–985. [Online]. Available: https://doi.org/10.1145/3338906.3340456

[9] R. Potvin and J. Levenberg, "Why Google stores billions of lines of code in a single repository," *Commun. ACM*, vol. 59, no. 7, pp. 78–87, June 2016. [Online]. Available: http://doi.acm.org/10.1145/2854146

[10] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu, "DifFuzz: Differential fuzzing for side-channel analysis," in *ICSE*, 2019.

[11] R. Swiecki, "HonggFuzz," http://honggfuzz.com, 2015.

[12] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[13] Intel Corporation, Willow Garage, and Itseez, "Open source computer vision library," 2019. [Online]. Available: https://opencv.org

[14] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: A case study," in *ESEC/FSE*. ACM, 2015, pp. 191–201.

[15] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Commun. ACM*, vol. 61, no. 4, March 2018.

[16] Google. Continuous integration. [Online]. Available: https://google.github.io/oss-fuzz/getting-started/continuous-integration/

[17] ——, "Third-party," Google's open source documentation, 2019. [Online]. Available: https://opensource.google.com/docs/thirdparty

[18] ——. ClusterFuzz. [Online]. Available: https://google.github.io/clusterfuzz/

[19] K. Varda, "Protocol buffers: Google's data interchange format," Google Open Source Blog, July 2008.

[20] O. Chang and K. Lewandowski, "Launching OSV - better vulnerability triage for open source," Google Security Blog, February 2021. [Online]. Available: https://security.googleblog.com/2021/02/launching-osv-better-vulnerability.html