Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing

Emre Güler
Ruhr-Universität Bochum
emre.gueler@rub.de

Philipp Görz
Ruhr-Universität Bochum
philipp.goerz@rub.de

Elia Geretto
Vrije Universiteit Amsterdam
e.geretto@vu.nl

Andrea Jemmett
Vrije Universiteit Amsterdam
a.jemmett@vu.nl

Sebastian Österlund
Vrije Universiteit Amsterdam
s.osterlund@vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Thorsten Holz
Ruhr-Universität Bochum
thorsten.holz@rub.de

ABSTRACT

Combining the strengths of individual fuzzing methods is an appealing idea to find software faults more efficiently, especially when the computing budget is limited. In prior work, EnFuzz introduced the idea of *ensemble fuzzing* and devised three heuristics to classify properties of fuzzers in terms of diversity. Based on these heuristics, the authors *manually* picked a combination of different fuzzers that collaborate.

In this paper, we generalize this idea by collecting and applying empirical data from single, isolated fuzzer runs to *automatically* identify a set of fuzzers that complement each other when executed collaboratively. To this end, we present Cupid, a collaborative fuzzing framework allowing automated, data-driven selection of multiple complementary fuzzers for parallelized and distributed fuzzing. We evaluate the automatically selected *target-independent* combination of fuzzers by Cupid on Google's fuzzer-test-suite, a collection of real-world binaries, as well as on the synthetic Lava-M dataset. We find that Cupid outperforms two expert-guided, *target-specific* and *hand-picked* combinations on Google's fuzzer-test-suite in terms of branch coverage, and improves bug finding on Lava-M by 10%. Most importantly, we improve the latency for obtaining 95% and 99% of the coverage by 90% and 64%, respectively. Furthermore, Cupid reduces the amount of CPU hours needed to find a high-performing combination of fuzzers by multiple orders of magnitude compared to an exhaustive evaluation.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

ACM Reference Format:
Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. In . ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, fuzzing has become an essential tool for finding bugs and vulnerabilities in software. Fuzzers such as AFL [29] and HonggFuzz [12] have successfully been applied in practice to generate inputs to find bugs in a large number of applications [24]. Recent work in fuzzing [1, 4, 15, 23, 25, 28] has focused on improving test case generation by implementing new input mutation and branch constraint solving techniques.

Since it is common to use automated bug-finding tools to find newly introduced bugs in software development scenarios (e.g., on every new commit/release), in pentesting scenarios (e.g., to find evidence of vulnerabilities), or in server consolidation scenarios (e.g., where spare CPU cycles can be dedicated to fuzzing), producing results in bounded time is crucial. Consequently, we target practical use cases where the time budget available for fuzzing is limited and it may be difficult to saturate coverage within that budget. It is, thus, important to look at how existing tools can be utilized more efficiently. Large-scale fuzzing campaigns, such as OSS-Fuzz [24], have shown that fuzzing scales well with additional computing resources towards finding security-relevant bugs in software. Moreover, researchers further improved the speed of fuzzing by parallelizing and distributing the fuzzing workload [17, 18, 29]. Typically, in these setups, multiple instances of the same fuzzer run in parallel, where the findings are periodically synchronized between these fuzzers [24]. In contrast, EnFuzz [5] demonstrated that running *different* fuzzers in combination leads to a noticeable variation in performance, paving the way for further improvement. Intuitively, this stems from the fact that fuzzers that have different properties and advantages in some areas (e.g., certain types of binaries or conditions) often come with disadvantages in others. Hence, a *collaborative fuzzing* run using a combination of fuzzers with different abilities can outperform multiple instances of the same fuzzer. EnFuzz explored this idea and the authors introduced heuristics that can be used to select different fuzzers that cooperate to find bugs more efficiently.

We generalize this idea and show that, given a set of existing fuzzers and a number of cores, selecting a good mix of fuzzers is a non-trivial but crucial step in maximizing the overall performance of a collaborative fuzzing process. To achieve this, we develop a framework called Cupid to optimize a collaborative fuzzing run by automatically predicting which fuzzer combinations will perform well together. We show that data from a single, isolated fuzzing campaign on a representative set of branches can be used to estimate which fuzzers complement each other and maximize code coverage in a collaborative fuzzing run on unknown binaries.

In an extensive evaluation on different data sets and more than 40,000 CPU hours spent, we not only show that our prediction on how well fuzzer combinations will perform together closely resembles real-world results, but we also show that our proposed one-off *data-driven and automatic* prediction results in a fuzzer combination that clearly outperforms two different *expert-guided and hand-picked* combinations selected by EnFuzz in regards to branch coverage, bug finding, and latency to find coverage. Furthermore, finding a high-performing combination of fuzzers takes *linear time* and, compared to an exponentially growing exhaustive search, reduces the computation time by multiple orders of magnitude.

In this paper, we make the following contributions:

- We demonstrate that a high-performing combination of fuzzers can be predicted by measuring the single, isolated performance of fuzzers on real-world binaries. Based on this insight, we develop Cupid, an extensible and scalable tool to predict high-performing collaborative fuzzer combinations.
- We present and evaluate a novel *complementarity* metric for calculating how well multiple fuzzers collaborate.
- We demonstrate how our data-driven approach allows for a linearly scaling, automated fuzzer selection process, avoiding the need for expert guidance as well as avoiding exponentially growing exhaustive search (and in comparison, reducing the number of necessary CPU hours by multiple orders of magnitude).

2 BACKGROUND

Fuzzing is the process of automatically finding bugs by generating randomly mutated inputs and observing the behavior of the application under test [21]. Current fuzzers are mainly *coverage-guided* [29], meaning that they try to generate inputs to maximize code coverage. The ever-growing code size of projects like web browsers require developers to scale performance by running fuzzers in parallel [18, 24, 29]. When automatically testing heavyweight applications like Chrome, with over 25 million lines of code [6], it is clear that fuzzing tools need to utilize multi-core and distributed systems to maximize code coverage and to increase the likelihood of finding bugs. This strategy is already in use, e.g., by the ClusterFuzz project [24].

For this purpose, fuzzers like AFL ship with a parallel-mode [18, 29], where multiple AFL instances share a corpus and thus synchronize their efforts. Although this approach does indeed increase code coverage, it does not solve some of the limitations inherent to a specific fuzzer in question. For example, when plain AFL has difficulties solving *magic bytes* comparisons, multiple instances of AFL will still have a low probability of solving these conditions.

To counter the limitations imposed by using one single type of fuzzer, EnFuzz [5] introduces *ensemble fuzzing*. The authors demonstrate that combining a *diverse* set of fuzzers leads to greater code coverage compared to running multiple instances of the *same* fuzzer. The boost in performance seems to stem from the symbiosis of the different fuzzing techniques, where the combination of fuzzers are more likely to cancel out individual disadvantages, while retaining their strengths. Thus, it is important to find a high-performing combination of fuzzers to maximize the expected return.

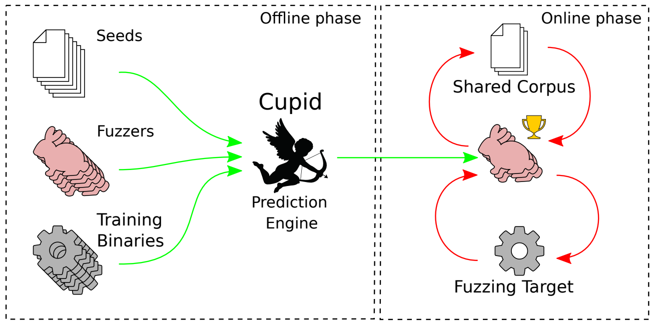

Figure 1: Overview of Cupid and its different components.

3 OVERVIEW

Prior to our work, there were two main approaches for finding a high-performing combination of fuzzers: (1) evaluating a hand-picked selection of fuzzers, or (2) exhaustively evaluating every possible combination of fuzzers on a number of applications. Both approaches come with multiple downsides:

(1) Hand-picking combinations requires expert knowledge of fuzzers and thus demands significant manual effort. Additionally, this process is likely to introduce human biases.
(2) An exhaustive evaluation requires a number of fuzzing runs that grows exponentially with the size of the combination and the number of fuzzers to test. This amount of compute quickly reaches infeasible levels.

With Cupid, we propose a scalable solution that requires *no human expertise in judging the performance of fuzzers* and only *linear computation time* in the number of fuzzers. This is achieved by evaluating fuzzers *individually* to collect information on their performance, while only simulating the collaborative aspect. We show that, given this data, it is possible to predict an approximate performance ranking of fuzzer combinations in a *collaborative* setting, which Cupid uses to select a *diverse* and high-performing combination of fuzzers. As this data has to only be collected once per fuzzer in isolation, adding new fuzzers does not lead to an exponential growth of necessary evaluation runs. In addition, it is not necessary to run additional evaluations when, for example, the available number of CPU cores change. After the individual fuzzer evaluation, Cupid is able to predict a likely candidate for the highest-performing combination in a matter of seconds for all practically relevant combination sizes.

An overview of our approach is shown in Figure 1. Cupid is split into a one-time *offline phase* to determine a likely candidate for a generally applicable high-performing combination of fuzzers, which is then used in future fuzzing scenarios (the *online phase*). The core idea behind the one-off training phase is to collect data on how well the single, isolated fuzzers perform on a representative set of real-world binaries, which would allow us to predict a complementary combination of fuzzers that would work *independently of the future fuzzing targets.* While we cannot prove that there exists such a universally representative set of binaries, our experiments show

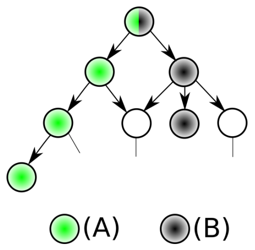

Figure 2: Different paths solved by fuzzer (A) and fuzzer (B) even after multiple runs.

how we were able to select a training set that enabled Cupid to select a fuzzer combination that performed well on a broad range of previously unseen test binaries.

4 DESIGN

When automatically searching for a high-performing combination of fuzzers for a collaborative fuzzing campaign, we not only need to define what constitutes a high-performing combination, but also how it could be calculated and predicted using a data-driven approach without guidance by a human.

Intuitively, fuzzers that reach different parts of the program under test should benefit from sharing their progress, since they will provide each other with new seed files that the other would possibly never find (i.e., they are *complementary*). However, several instances of the *same* fuzzer can also benefit from each other, as their cooperation increases the chance to solve branches in a given time-frame, i.e., their cooperation increases the average speed in which they solve branches. In real-world scenarios, however, fuzzer combinations neither show completely identical behavior nor are they completely orthogonal—it is thus our goal to find candidates with a high degree of *complementarity*.

4.1 Complementary Fuzzers

EnFuzz bases their prediction on the idea that the *diversity* of fuzzers is paramount, i.e., using as many different strong fuzzers as possible. While we agree with this approach in principle (Cupid applies diversity as a selection criteria), we improve upon this by ranking fuzzers based on a criteria that we call *complementarity* of fuzzers. To conceptualize what constitutes a complementary combination of fuzzers, we need to discuss several key questions:

(a) Why not run multiple instances of the best fuzzer?
(b) Why not just select for diversity?
(c) When are fuzzers complementary in practice?

To answer question (a), we refer to Figure 2. This figure illustrates a simplified example of two fuzzers (A and B) visiting two different paths of the program space. Suppose that, in this scenario, our performance data suggests that these fuzzers will always take these paths, and assuming that we have two CPU cores for fuzzing (*combination of size two*) to run collaboratively, there are only three available combinations to run: (1) A and A, (2) B and B, (3) A and B.

In this example, simply going by the number of basic blocks covered by each single fuzzer individually, one would erroneously predict option (1) as the best choice. However, as fuzzer A will always take the same path in this example, two instances of the same fuzzer will not increase code coverage. In fact, they would still find the same four basic blocks. Thus, the addition of more resources (CPU cores) would not result in better performance. This issue also affects option (2). In option (3), however, both fuzzers are diverse as well as complementary and contribute seeds the other is unable to generate. A collaborative run between the two would lead to a total code coverage of six basic blocks.

Although this clear-cut split between two fuzzers rarely happens in real-world scenarios, it illustrates an important point: running the best-performing fuzzer is the best choice in single-instance mode, but when two (or more) fuzzers run collaboratively, we want them to maximize code coverage. As such, we want the *union* of their coverage to add up to the maximum possible value—which is the case when they *complement* each other well.

To answer research question (b), i.e., why we should not just select for diversity, we refer back to Figure 2. In this case, the combination of two different fuzzers led to the best possible outcome, because they are guaranteed to cover the same basic blocks every time, an assumption highly unlikely in the real world. Suppose, as a very simplified example, fuzzer A can only solve the illustrated four branches in 5% of its runs, and fuzzer B solves its three branches in 60% of its runs. Although fuzzer A could potentially solve more branches than B (four vs. three), it is unlikely for this to happen. The expected number of branches that fuzzer A solves is $0.05 \cdot 4 = 0.2$. On the other hand, it is likely for fuzzer B to solve three branches, leading to an expected average number of branches of $0.6 \cdot 3 = 1.8$.

In this modified scenario, judging the performance of a combination of fuzzers is more complicated. Although only the combination of fuzzers A and B could reach the highest possible coverage (the union of the branches they can only reach in collaboration), this scenario is unlikely given the probabilities above. That is, in the average run, fuzzer A will not contribute any new branches due to their low probabilities. In contrast, fuzzer B has a more consistent performance. Regarding the combinations of two instances of fuzzer A, the *expected* average number of branches would still be low, whereas the two instances of fuzzer B would actually maximize the expected average number of branches. In this case, diversity on its own did not lead to the choice of the best fuzzer combination.

Thus, to answer research question (c), the complementarity metric measures the degree in which multiple fuzzers are complementary, i.e., the *union* of the expected *mean code coverage.* As such, we consider a combination to be high-performing (*highly complementary*), if the combination, on average, is expected to maximize code coverage in the shortest amount of time.

In conclusion, our approach called Cupid combines the advantages of both the data-driven complementarity metric as outlined in this work, as well as the diversity heuristic as outlined by EnFuzz. Cupid uses complementarity to automatically rank combinations by their predicted performance, and applies the diversity heuristic to select a single high-performing combination of fuzzers. Specifically, given the noise in the training data and the fact that Cupid's predictions are estimates, we consider two combinations to be reasonably *similar* if the difference in their predicted performances is less than 5%. As such, we collectively classify all combinations that are similar to the top-ranking combination as *high-ranking*. Cupid selects the highest ranked, most diverse combination out of

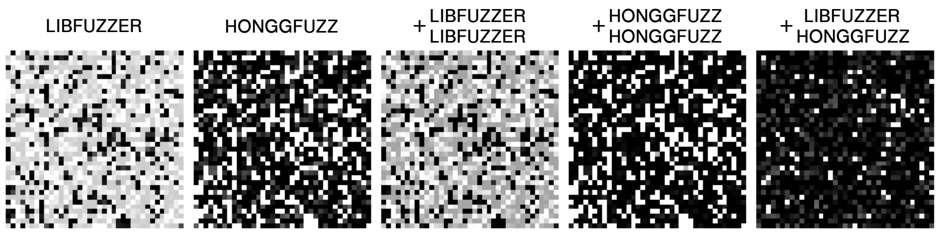

Figure 3: Actual probabilities (first two images) and predicted synchronized probabilities for the branches in FREETYPE2 when combining Honggfuzz and libFuzzer. Every cell represents a single branch. The alpha value of a cell represents the probability that this branch will be solved by this fuzzer (or fuzzer combination). For illustration purposes, we only selected an excerpt of all branches to make the differences visually distinguishable.

those high-ranking combinations. Intuitively, given two combinations that are predicted to perform similarly well, we should err on the side of diversity and choose the combination with the greatest number of different fuzzers, as this combination is less likely to be negatively affected in case of an underperforming fuzzer on some binaries (i.e., while maintaining a similar performance, a combination consisting of more different fuzzers makes it more unlikely that all of them will fail on a given branch).

4.2 Predicting high-performing fuzzer combinations

In theory, collecting data on how well fuzzers perform on a diverse set of branches is valuable information for assessing how complementary a combination of fuzzers is. If we assume that the set of branches in this training phase was representative of real-world binaries, we should be able to approximate which combination of fuzzers, on average, would have the best chance of maximizing code coverage in future runs on unknown binaries. Note that we only need to make this prediction once when a new fuzzer is introduced to the framework—the resulting choice of a specific fuzzer combination is, on average, likely to perform reasonably well in future collaborative fuzzing runs, independent of the binary. Given this data, we calculate which fuzzers would be complementary. Furthermore, we predict a ranking of all possible fuzzer combinations and then select a high-performing combination of fuzzers. The accuracy of the prediction, however, is dependent on the quality of the training data, i.e., if the training data reflects the mixture of branches seen outside of the training data reasonably well, our prediction is more likely to hold true on unknown binaries.

To make these predictions, Cupid assumes a collaborative fuzzing model, in which every fuzzer starts with the same seed and has some limited time frame to fuzz on their own, after which they share their seed files (*synchronization*). Afterward, each fuzzer continues on their own until the next synchronization happens. This is similar to real-world scenarios, as multiple fuzzers in parallel will start with the same seeds and try to solve the same branches reachable from these seeds. With a growing number of collaborating fuzzers, more parallel attempts are made at solving these branches.

The likelihood of one branch being solved by multiple collaborating fuzzers in the given time period can be calculated by the probability that *at least one* of the fuzzers will solve this branch. For example, assume the collaborating fuzzers A and B both have a probability of 50% for solving a branch in the given time period, then the *total* predicted probability for this branch would be 75%. That is, the branch would only *not* be solved if *both* fuzzers failed to do so, which would only happen with a predicted probability of 25%, see Equation 1.

$$BranchProb_F(b) = 1 - \prod_{f \in F} (1 - b_f) \tag{1}$$

where F are the collaborating fuzzers, b the branch we want to calculate the combined probability for, and b_f the probability of fuzzer f for solving branch b. This calculates the probability on a branch-basis, i.e., how likely it is that these collaborating fuzzers would solve this branch in the given time period.

To calculate how well fuzzers complement each other in *total*, we calculate the sum of all of their branch predictions (see Equation 2):

$$AverageBranchCov_F = \sum_{b \in B} BranchProb_F(b) \tag{2}$$

where F are the collaborating fuzzers, and B are all branches. This represents the expected average number of branches that would be solved by the collaborating fuzzers.

With this approach, we do not just calculate how many different branches two fuzzers would find—as we run multiple fuzzers in parallel, inevitably the probability of solving a branch will increase if the branch probability was greater than zero and less than one. Hence, two instances of the *same* fuzzer will also complement each other to a certain degree, which is in line with real-world scenarios, where it is expected that multiple synchronized instances of the same fuzzer will outperform a single instance.

Although this design choice tries to maximize the expected average branch coverage, it also inherently selects for branches that only one fuzzer can solve (*rare branches*). As an example, assume fuzzer A can solve a branch with probability 0.9. Two instances of this fuzzer would, given the above formula, achieve a probability of 0.99 for solving this branch, an increase of only 0.09. However, given a fuzzer B that can solve a different rare branch with a probability of only 0.2, the combination of A and B would increase the expected average branch coverage by 0.2 as opposed to 0.09. As seen in this case, Cupid leans toward fuzzers that solve rare branches if these fuzzers would increase the expected average branch coverage more than fuzzers with higher overall probabilities.

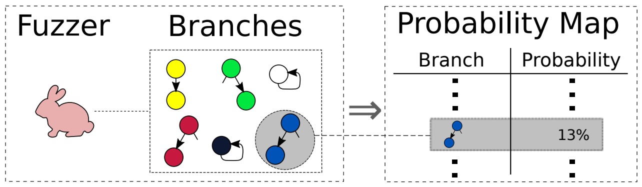

Figure 4: Cupid lets every fuzzer work on a diverse and representative set of branches to extract a probability map of how often a fuzzer was able to solve a branch in a limited time frame.

Note that one of our goals is to increase the return on investment, which can be simply translated into maximizing coverage per time. We achieve this only due to the way we collect the training data. By limiting the time fuzzers have, we condition our selection process to prefer faster fuzzers.

With these design choices, Cupid balances two important aspects of selecting fuzzers: (1) due to the complementarity metric, Cupid selects fuzzers that solve different branches, i.e., maximizes the *total coverage*; and (2) due to the way we perform data collection, Cupid selects fuzzers that are faster at finding coverage.

As an illustration of the results of the complementarity metric, refer to Figure 3. This figure depicts an excerpt of the resulting probabilities by all three possible combinations of two collaborating fuzzers (Honggfuzz, libFuzzer) while fuzzing freetype2. Each cell represents a specific branch. For each cell, a darker color represents a higher probability. In this case, the combined predicted bitmap of two different fuzzers is the best combination. This also illustrates how two instances of the same fuzzer will increase the probabilities on the same branches, whereas different fuzzers complement each other additionally on branches that only one of them could find.

4.3 Representative Branches

Since the prediction of Cupid is based on how well fuzzers perform on individual branches, an important factor is the selection of which branches to base that prediction on. The chosen branches should be *diverse*, to reduce the possibility of overfitting, and be *representative* of real-world binaries, to allow practical application. These branches are collected by fuzzing a diverse set of binaries with each individual fuzzer in isolation, while simulating a collaborative setting. This process only needs to happen *once* for any fuzzer. Additionally, adding a new fuzzer or changing configuration parameters of a fuzzer require only re-evaluation of the new or affected fuzzer.

As we are not interested in the binaries themselves, but only in their branches, we want to sample as many different branches as possible from the program space. As such, every fuzzer is run multiple times on all binaries, each time with different seed files that were chosen to allow different areas of the binary to be reached.

Additionally, since we want to maximize the return on investment on a limited budget, we focus on maximizing code coverage over time. Therefore, the empirical data on each fuzzer should reflect how well they operate on a given set of branches in a *limited* time window. Here, the central idea is to let the fuzzers run only for a limited time period and extract information on how many branches they are able to solve.

Finally, when the empirical training data is collected, we have a mapping from every fuzzer to a list of representative branches and their corresponding probabilities (see Figure 4). Given this mapping, as outlined earlier, Cupid calculates and predicts how complementary the fuzzers are.

With this in mind, a major novelty of Cupid is showing how data from a single, isolated fuzzing campaign on a representative set of branches can be extrapolated to predict the candidates that are likely to complement each other and maximize code coverage in a collaborative fuzzing run on unknown binaries.

5 IMPLEMENTATION

In the data-driven approach used by Cupid, the quality of the data directly influences the resulting prediction. Particularly, we need to avoid basing the prediction on a set of branches that is not actually representative. We can control data collection mainly with the chosen binaries and parameters of the individual fuzzing runs.

5.1 Collecting empirical data

In particular, we choose ten different binaries as the training set (freetype2, re2, boringssl, llvm-libcxxabi, libjpeg-turbo, pcre2, wpantund, lcms, vorbis, harfbuzz) which contain a large variety of branches. These libraries are vastly diverse in their nature and purpose, they cover categories such as font processing, regular expressions, encryption, image parsing, network interface management, color management, and audio processing. Hence, we think that these binaries are well-suited for our evaluation purposes. Note that adding binaries from additional application categories could further improve the representativeness of the training data. To ease the process of automation, for the evaluation, these training binaries are provided by Google's fuzzer-test-suite, as this allows better extendability to a multitude of fuzzers through a unified compilation process for all binaries. Note that, although all binaries in the training and test set are from Google's fuzzer-test-suite, the projects themselves are separate and developed by different teams.

To create the seed files, which act as the starting points for the individual fuzzing runs, we fuzz these training binaries for 12 hours. This is a one-off effort that does not have to be repeated. Out of these seed files, we select five seeds that were found 2-3 hours apart, this temporal distance reduces the overlap between the branches the fuzzers can reach. We observe the performance of each fuzzer by letting them run once for each seed file for all training binaries. This resembles the collaborative model outlined above, where every new seed simulates a completed synchronization.

Due to inherent randomness in any fuzzing process, we let the fuzzers run 30 times and calculate probabilities for all branches that reflect how often a branch was found by this fuzzer in that time period (i.e., a branch will have a probability of 50% for fuzzer A, if fuzzer A was only able to solve that branch in 15 out of its 30 runs in time t). Again, this is a one-off effort for each fuzzer, which does not add any overhead to the actual fuzzing process in the future.

As we need to balance the required computation time to exhaustiveness, we limit each run to $t = 30min$ for our evaluation. While users may choose to tune it, in our analysis, we found this time limit to work consistently well in practice. One disadvantage with using such a time limit is that some fuzzers might need more time to reach their peak performance. Although we were unable to empirically observe this effect with our fuzzer selection, this might be an issue for future fuzzers. Additionally, some fuzzers focus on more difficult branches but take longer to solve them—these fuzzers could be negatively affected by this time limit.

However, we believe that the numerous advantages vastly overshadow these possible disadvantages: First, limiting the run time reduces the risk of fuzzers getting stuck on an initial branch, while only one fuzzer might be able to solve that branch and get a *disproportional* advantage by solving all the following branches. Although it is important to reflect in the data the advantage this fuzzer brings, if the timeout were longer, it would increasingly appear that many branches were only solved by this one fuzzer, thus skewing how many rare branches this fuzzer is actually able to solve (i.e., due to the unsolvable initial branch, the other fuzzers were not given the chance to directly test themselves on the following branches). Second, even with this time limit, the data reflects the internal short-term scheduling mechanism. Each fuzzer faces a multitude of branches reachable from any given seed file—how it chooses what branches to solve and what corpus files to mutate, impacts the overall performance of the fuzzer. Third, due to the time limit, the execution speed of the fuzzer is reflected in the data. Fuzzing speed is an important factor in maximizing code coverage. [10, 13] Fourth, our evaluation suggests that our prediction framework is accurate in predicting a high-performing fuzzer combination. Although improvements might be possible, this time limit does not seem to impact the prediction accuracy significantly. In conclusion, our approach approximates real-world fuzzing scenarios and therefore we take many performance-relevant properties of fuzzers into account. The empirical data reflects, at least, the following attributes of fuzzers: (i) ability to solve a variety of branches, (ii) short-term scheduling policies, and (iii) execution speed.

As mentioned earlier, note that Cupid has to collect this data only *once* per fuzzer to update the prediction. As the goal of the prediction is to be generalized via a representative set of branches, the prediction is made *independent of the future fuzzing target.* Additionally, it is not necessary to recollect data when the combination size changes. Furthermore, if one were to add a new fuzzer to the framework, one would *only* need to collect empirical data on this new fuzzer. This is the main improvement that allows for predictions that scale linearly with the number of fuzzers in the pool (whereas an exhaustive search would grow exponentially with the number of fuzzers as well as the combination size). However, as mentioned earlier, the collected data is reliant on the same fuzzer configuration for future runs (e.g., fuzzing mode or instrumentation method), as such, significant changes to the fuzzer configuration will require new runs and a new prediction for accurate results.

5.2 Components and Implementation

Cupid consists mainly of the prediction engine and some small packaging for the different fuzzers.

5.2.1 Prediction engine (Python, ~4k LOC). This component collects the resulting corpus files and generates code coverage information. To collect this information, we developed a Python library that internally uses AFL (built as a C library) to quickly collect code coverage given a corpus directory. The resulting data is exported as a bitmap converted to a NumPy [22] array to allow for easy probability calculations. Finally, these probabilities are then converted to a ranking of fuzzer combinations.

5.2.2 Fuzzer drivers (Python, ~2k LOC). Each fuzzer runs in its own Docker [20] container which is controlled by a driver that coordinates the Docker images, assigns them to their respective CPU cores, and manages their start and stop times. The framework is extensible, allowing developers of new fuzzers to easily add support by simply creating a new container image and adding around 100 lines of Python code to our driver. We implemented support for eight fuzzers, AFL-based fuzzers are supported out-of-the-box.

5.2.3 Patches for compatibility. We patched Honggfuzz [12] and libFuzzer [19] (both less than 150 LOC) to allow for external test case syncing at runtime.

6 EVALUATION

To evaluate the effectiveness of the prediction of Cupid, we first show that the predicted ranking of combinations corresponds to the ranking based on real-world performance. Based on these results, we predict a combination of fuzzers of length $n = 4$ to replicate the evaluation of EnFuzz and show that our automatically selected combination outperforms the expert-guided, hand-picked selection of EnFuzz in terms of code coverage and bug finding capabilities.

For providing a valid evaluation, the set of fuzzer-test-suite binaries is split in two for most experiments: the *training set* (freetype2, re2, boringssl, llvm-libcxxabi, libjpeg-turbo, pcre2, wpantund, lcms, vorbis, harfbuzz) and the *test set* (woff2, sqlite, proj4, openthread, openssl-1.1.0c, openssl-1.0.2d, libxml2, libssh, libpng, libarchive, json, guetzli, c-ares). One binary out of 24 was excluded because it did not run for all of the fuzzers (openssl-1.0.1f). We used the seeds supplied with fuzzer-test-suite—if none were available, we instead used a seed containing only the null byte. As mentioned earlier, fuzzer-test-suite allows for a unified compilation process, but the included projects are separate, diverse and developed by different teams, which is required to be able to train on a set of branches that is representative of unknown future binaries.

To avoid discrepancies due to implementation details, we use our own framework and Docker images to evaluate both EnFuzz and Cupid. This is to keep the playing field as leveled as possible, so that the recorded differences are only due to the selection of fuzzers and not due to any other factors.

Note that the numbers presented by EnFuzz [5] report the improvement over the sum of the coverage of the whole dataset. For comparison, we also report this number (as *Improvement (sum)*), but we believe that this metric heavily skews the results as large programs are weighted higher and thus, this metric is not very informative. When not stated otherwise, we use the geometric mean to calculate the overall improvement across programs, which is standard practice in the field, to circumvent this issue. We limit the experiments to 10 hours to reduce the total time needed for the experiments (in total, the experiments presented in this section have required 40,000 CPU hours or ~4.5 CPU years). Furthermore,

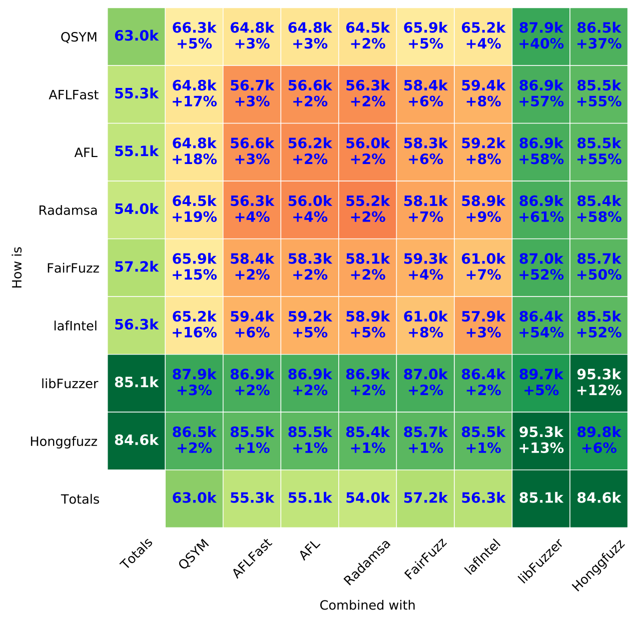

Figure 5: The *totals* column and row show the solved branches for an average fuzzing run with the respective fuzzer. The inner blocks show the *predicted* performance of each two fuzzer combination. Note that two instances of the same fuzzer is also a valid combination. The upper value represents the complementarity, the lower value is the percentage increase compared to the "How is" fuzzer.

even with the runtime limit of 10 hours, we saturate the fuzzing coverage for all binaries except two.

6.1 Evaluated fuzzers

Cupid ships with eight default fuzzers in total:

- AFL [29] is a superseded fuzzer that requires little set-up. However, as seen by several fuzzers we use for the evaluation, AFL is often used as a starting point for new research.
- QSYM [28] is the concolic execution engine of a hybrid fuzzer together with AFL. QSYM shows that loosening the strict soundness requirements of concolic executors leads to better performance and showcases how integration with a fuzzer can help validate and improve the speed of the concolic engine. For our evaluation, QSYM acts as a fuzzer that trades targeted solving of branches against higher execution count.
- AFLFast [3] is based on AFL, and improves the scheduling of testcases. The basic idea is that fuzzers with simple scheduling algorithms spend a lot of time on testcases that exercise a small number of paths. The authors show that time would be better spent on less frequently visited branches.
- radamsa [11] is a black-box mutator with more sophisticated mutators than AFL. We use AFL++ in radamsa mode, which adds coverage feedback.
- FairFuzz [16] improves upon AFL by putting more focus on rare branches, as well as a mutation mask that specifies which parts of a testcase to modify. Both of these techniques act as a form of attention to focus more time on interesting parts of the binary.
- lafIntel [15] is a collection of compiler passes that help fuzzers to progress through difficult branches by splitting them up into several branches. These individual branches are easier to solve. Due to difficulties compiling libraries from the fuzzer-test-suite, we instead use the CompCov mode of AFL++ [8] in QEMU mode.
- libFuzzer [19] is a library that adds a fuzzer stub into the program at compile time. libFuzzer can be run in-process, which vastly reduces the overhead compared to the common fork server approach (AFL). However, the programs needs to be of high quality, since memory leaks and crashes also affect the fuzzing run. libFuzzer comes with a multi-process fork mode that makes it easier to ignore crashes, which is what we use in our case. libFuzzer is not based on AFL and thus is another diverse addition. We patched libFuzzer to allow for external test case syncing at runtime.
- Honggfuzz [12] is another fuzzer that is not based on AFL and thus differs significantly in its code base, e.g., it introduces a different seed scheduling algorithm, as well as mutators and different coverage metrics. We patched Honggfuzz to allow for external test case syncing at runtime.

For an overview of the configuration parameters for each fuzzer, refer to Table 5 in the appendix.

Despite the fact that we support eight fuzzers out-of-the-box, to ensure fairness in the evaluation, we limit the pool of fuzzers to those that EnFuzz had access to (AFL, AFLFast, FairFuzz, libFuzzer, QSYM, radamsa). Although this guarantees a fair comparison to EnFuzz, these fuzzers have a lower overall diversity, which is a crucial factor in maximizing possible performance improvements in a collaborative run [5], so the restriction on fuzzers also limits the possible magnitude of our results.

For a visual representation of the diversity in the pool of fuzzers, refer to Figure 5. The top value in every cell represents the average predicted branch coverage for this combination, and the bottom value describes the improvement provided by the second fuzzer compared to a single instance of the first fuzzer. For instance, AFL and QSYM complement each other well: when run collaboratively, the predicted average branch coverage will increase by 18% (from ~55k to ~64k) as compared to a single AFL instance. Although an improvement by 18% is valuable, the total predicted average branch coverage is still low compared to the best predicted combination of Honggfuzz and libFuzzer (~95k). Furthermore, AFL-based combinations show only low improvements when combined with most other AFL-based fuzzers, but their average predicted branch coverage is also significantly lower than non-AFL based combinations.

In conclusion, we recommend to extend the pool of fuzzers by non-AFL based fuzzers to increase the diversity and thus improve the overall performance of collaborative fuzzing. However, as mentioned before, to keep the evaluation fair, we use the same fuzzers that EnFuzz has access to as well.

Table 1: The result of the prediction framework for n=2. "Complementarity" represents the predicted total probability for this fuzzer combination, i.e. the predicted average number of branches visited on all of the training data by this fuzzer combination. Combinations in bold are the selections we used in Experiment 1.

Ranking	Combination	Complementarity
1.	**libFuzzer, libFuzzer**	**93456.32**
2.	FairFuzz, libFuzzer	93099.12
3.	AFL, libFuzzer	92510.05
	...	
10.	FairFuzz, radamsa	76347.06
11.	**FairFuzz, QSYM**	**75623.01**
12.	AFL, AFL	73953.05
	...	
19.	AFLFast, QSYM	70916.66
20.	QSYM, radamsa	69329.05
21.	**QSYM, QSYM**	**62795.15**

6.2 Prediction of fuzzer combinations

In this section, we evaluate the accuracy of our prediction framework by comparing the predicted rankings to real-world results. Additionally, we evaluate against fuzzer-test-suite and Lava-M to compare the performance of our best-predicted combination to EnFuzz.

6.2.1 Comparing predicted and actual rankings. Cupid uses empirical data to predict a ranking for fuzzer combinations. In this instance, *ranking* describes the performance of one combination relative to other combinations. Because our predicted ranking is based on empirical data extracted through time-limited fuzzing runs in isolation, it is necessary to evaluate if our ranking accurately resembles real-world results in collaborative runs.

In order to evaluate the quality of certain combinations of fuzzers, the most indicative metric is the median code coverage at the end of the selected time frame. However, for some of the binaries, most fuzzer combinations reach the maximal coverage too quickly. For this reason, we decided to calculate the area-under-curve (AUC) as well because it takes into account the time taken to reach every given value of coverage. As a result, if two combinations find the same total code coverage, we prefer the combination that achieves this in the shortest amount of time, which would be reflected by a larger AUC.

To conduct this experiment, we face two issues:

(1) On some binaries, fuzzers find nearly all coverage in a fraction of the total scheduled runtime (i.e., they flat line after a short time). For these binaries, the performance measurements are difficult to compare. If most fuzzers reach the same code coverage after a few minutes, not only will the mean or median code coverage be near identical after 10h, but also the area-under-curve.
(2) Some combinations of fuzzers reach very similar performance, thus predicting an accurate ranking is not only difficult but could also be misleading, since the ranking obfuscates how similar combinations are.

To address the first issue, we set combination size of $n = 2$, a smaller number of parallel fuzzers will take longer to achieve maximum coverage and thus, allow for more accurate rankings. Additionally, due to the magnitude of necessary CPU hours for 10 runs for all 21 combinations on all 13 test binaries, we let all combinations fuzz every binary 10 times for only 1h. With these settings, it will take longer to hit the upper bound of code coverage on a binary in the given time window.

To address the second issue, we first evaluate those fuzzer combinations that significantly differ in their predicted performance and, subsequently, we perform an exhaustive ranking correlation with all $n = 2$ combinations. Thus, for our first experiment, we reduce our list of combinations to the best predicted combination, the worst predicted combination, and an additional combination in-between the two (as highlighted in Table 1) and compare how well their predicted rankings match real-world results. The evaluation was done on six machines with 40-core/80-thread Intel Xeon Gold 6230 CPU @ 2.10GHz processors and 192GB RAM, where each fuzzer got assigned a dedicated core.

Experiment 1. We fuzz the 13 test binaries 10 times for 1h each, select the median coverage over time and calculate the area-under-curve. Then we rank the three selected fuzzer combinations using this value. We use the Pearson correlation coefficient to measure the linear correlation between our predicted ranking and the real-world ranking. Additionally, we use the predicted performance value and the actual median AUC to calculate a more detailed correlation coefficient on a per-binary basis.

Results. The Pearson coefficient is calculated as $r = 0.81$ with $p < 0.01$. This represents, according to Evans [7], a *very strong* positive correlation ($0.8 \leq r \leq 1.0$). On a per-binary basis, 11 out of 13 binaries have a very strong positive correlation with $r \geq 0.9$ when their predicted performance is compared to the actual median AUC, while only two binaries have a negative correlation. The rankings of 10 binaries are *identical* to our prediction. In conclusion, our framework is indeed able to accurately predict the rankings when the combinations are dissimilar in their predicted performance.

Experiment 2. To showcase that similarly predicted combinations will still match reasonably well, we rerun the evaluation for not just the three distinct combinations, but all possible 21 fuzzer combinations for $n = 2$. More specifically, we modify the previous experiment to rank *all* possible 21 fuzzer combinations for each of the 13 test binaries.

Results. For this case, the correlation coefficient drops to $r = 0.6$ with $p < 0.01$. This is, according to Evans [7] slightly above a *moderate* positive correlation ($0.4 \leq r \leq 0.59$). With experimental data, this is expected: our prediction relies on data collected over a range of binaries and input seeds and thus Cupid is only able to predict the average performance over multiple fuzzing runs and binaries. Due to inherent randomness in any fuzzing process, a greater variance in the performance of some combinations is expected, and thus, combinations that are predicted to be similar in their performance will not match their predicted *ranking* perfectly in all cases. However, even in this scenario, 7 out of 13 binaries display a *very strong* positive correlation coefficient, with $r \geq 0.8$.

Note that although this evaluation is important in determining the accuracy of the predicted ranking, in real-world scenarios a security analyst is only interested in the top results. While there

Table 2: Median branch coverage on the test binaries from fuzzer-test-suite (10 runs with a run time of 10h). Bold values highlight the best result. The four following columns represent the speed-up in latency to reach the given percentage of the observed coverage. Bold values highlight positive speed-ups. The last column represents the p-value according to the Mann-Whitney U test between EnFuzz and Cupid. Bold values highlight statistical significance ($p < 0.05$). No test was possible for c-ares because both results were identical.

	Fuzzer Combination		Cupid Coverage Speed-up				
Binary	**EnFuzz**	**Cupid**	**90% Coverage**	**95% Coverage**	**97% Coverage**	**99% Coverage**	**p-value**
c-ares	**58**	**58**	0.00%	**+10.00%**	**+10.00%**	**+10.00%**	-
guetzli	**2617**	2603	-6.28%	-11.73%	-2.58%	-3.62%	0.26
json	707	**711**	**+135.91%**	**+59.89%**	**+45.03%**	**+1827.58%**	**0.01**
libarchive	3161	**3577**	**+36.54%**	**+10.50%**	**+3.06%**	**+2.39%**	**< 0.01**
libpng	668	**697**	**+64.57%**	**+543.47%**	**+1135.38%**	**+54.97%**	**< 0.01**
libssh	809	**811**	**+51.18%**	**+16.96%**	-20.29%	-44.23%	0.48
libxml2	2014	**2123**	**+160.85%**	**+268.99%**	**+169.16%**	**+74.59%**	**< 0.01**
openssl-1.0.2d	**786**	784	-5.26%	**+6.38%**	-3.39%	**+8.22%**	0.08
openssl-1.1.0c	777	**779**	**+29.17%**	**+29.17%**	**+29.17%**	**+84.92%**	0.07
openthread	**864**	863	**+69.23%**	**+27.68%**	-33.14%	-14.83%	0.48
proj4	2715	**2819**	-2.80%	**+57.57%**	**+45.21%**	**+28.68%**	0.11
sqlite	**913**	**913**	**+489.64%**	**+489.64%**	**+489.64%**	**+489.64%**	0.08
woff2	1058	**1102**	**+75.32%**	**+432.28%**	**+351.31%**	**+48.58%**	**< 0.01**
Total	17147	**17835**					
Improvement (sum)	-	**+4.01%**					
Improvement (geomean)	-	**+2.31%**	+59%	+90%	+74%	+64%	

Table 3: Median branch coverage on the trainings binaries from fuzzer-test-suite (10 runs with a run time of 10h). Bold values highlight the best result. The last column represents the p-value according to the Mann-Whitney U test between EnFuzz and Cupid. Bold values highlight statistical significance ($p < 0.05$).

Binary	**EnFuzz**	**Cupid**	**p-value**
boringssl	**1145**	**1145**	0.47
freetype2	5235	**6055**	**< 0.01**
harfbuzz	4124	**4272**	**< 0.01**
lcms	970	**1385**	**< 0.01**
libjpeg-turbo	1227	**1386**	**< 0.01**
llvm-libcxxabi	3305	**3432**	**< 0.01**
pcre2	**4377**	4189	**< 0.01**
re2	2190	**2205**	**0.03**
vorbis	932	**946**	0.12
wpantund	4186	**4302**	**0.02**
Total	27691	**29317**	
Improvement (sum)	-	**+5.87%**	
Improvement (geomean)	-	**+7.25%**	

might not exist one ultimate combination of fuzzers that outperforms all other combinations on *all* future binaries – as there are fuzzers that work exceptionally well on some specific binaries but not on others – we are able to find a combination that outperforms all other combinations *on average*. In this regard, our top predicted combination was indeed most often and more consistently ranked in the first place than all other combinations (i.e., if one were to use the real-world rankings to extract what should have been the best-predicted combination, one would end up with our prediction).

In conclusion, our predicted ranking is effective in finding well-performing combinations of fuzzers that realistically represents the real-world performance seen by the combinations in question.

6.2.2 *Evaluating on fuzzer-test-suite.*

To replicate the evaluation against EnFuzz for the fuzzer-test-suite binaries, we first predict the best $n = 4$ combination of fuzzers. As the empirical performance data was already collected for the previous experiment and no new fuzzers were introduced, only the prediction had to be updated. Predicting the full $n = 4$ combination of fuzzers takes less than a second on a off-the-shelf notebook (single process on an Intel(R) Core(TM) i7-8550U @ 1.80GHz with 32 GB RAM)—even for a rather unrealistic extreme case of $n = 10,000$ it takes less than five minutes to calculate the top 1,000 predictions. An excerpt of the resulting prediction ranking of $n = 4$ is displayed in Table 7 in the appendix. Given the training data, Cupid uses the complementarity metric and diversity heuristic, as outlined above, to automatically select the *highest ranking, most diverse* set of fuzzers: FairFuzz, libFuzzer, QSYM, AFL. The fuzzers included in the EnFuzz combination are AFL, libFuzzer, AFLFast, radamsa.

It is important to note that, according to our predicted ranking, due to the low diversity in the pool of fuzzers, our selected combination and EnFuzz will be very similar performance-wise—the difference in their predicted performances is less than 1.4%—so in total, only a small improvement is expected. Nevertheless, when an *expert-guided, hand-picked selection* (EnFuzz) is outperformed by a *data-driven, linearly-scaling automatic process* (Cupid), that is already in itself a significant improvement in terms of man-power, human bias, and processing time.

Experiment As the complementarity metric of Cupid is designed to prefer fuzzers that achieve the same code coverage in a shorter amount of time, we also calculate the speed-up of Cupid, in comparison to EnFuzz, to reach 90%, 95%, 97% and 99% of the maximum code coverage (i.e., the improvements in the median time to

Table 4: Median number of unique bugs found in the Lava-M set (10 runs for 10h each). The improvement is measured in comparison to EnFuzz-Q (which outperformed EnFuzz). (*) md5sum was excluded from the improvement calculation as EnFuzz-Q did not run properly for this binary.

Binary	EnFuzz	EnFuzz-Q	Cupid
base64	42	**48**	**48**
md5sum*	24	-	**25**
uniq	7	22	**29**
who	95	340	**360**
Total bugs	144	410	**437**
Improvement (sum)	-	-	**+6.59%**
Improvement (geomean)	-	-	**+11.75%**

reach that coverage). Calculating the speed up to compare fuzzers is also suggested in a recent work, that discusses the scalability of fuzzing [2].

Results. Cupid outperforms EnFuzz in terms of median code coverage in 16 out of 23 runs. The geomean improvement in branch coverage is +2.31% for the test binaries (see Table 2) and +7.25% for the training binaries (see Table 3). Furthermore, we achieve a +90% speed-up to reach 95% of the maximum coverage, and +64% to reach 99% of the maximum coverage. Our results show that the differences in branch coverage are often less than 100, i. e., the binaries flatline too fast and do not make much difference in the total branch coverage after a 10h run. In fact, the median branch difference (in the cases where EnFuzz has a higher score) is only *eight* branches.

When we repeated the experiment with more diversity in the pool of fuzzers (i. e., by adding Honggfuzz and lafIntel), it resulted in even better and more statistically significant results with +6.4% for the median branch coverage on the test binaries (see Table 6 in the appendix).

In conclusion, our automatic, data-driven process was able to select a better combination than a process consisting of expert guidance and extensive evaluation. Furthermore, no additional human action or runs are necessary if a different combination size (e. g. $n = 5$) is required—the new prediction would take *less than a second*, as opposed to multiple CPU years in an exhaustive evaluation. Additionally, the results suggest our combination did not overfit on the training data and generalizes well to the test binaries.

6.2.3 *Evaluating on Lava-M.*

Although we do not encourage evaluating against Lava-M as recent research suggests that it is rather unlike real-world vulnerabilities, mostly due do to its simplicity[9] (i.e., bugs are triggered by finding a 4-byte magic value), we include this experiment to replicate the Lava-M experiment of EnFuzz. To this end, we compare the median number of bugs found by EnFuzz to Cupid. As the authors of EnFuzz introduced a new combination called EnFuzz-Q (AFL, AFLFast, FairFuzz, QSYM) specifically picked for this evaluation, we additionally compare against this combination of fuzzers.

Also, as libFuzzer is included in the evaluation, we have to add libFuzzer support to Lava-M, which requires a LLVMFuzzerTestOneInput function that uses the given data to call the function to be fuzzed in persistent mode. As no other fuzzer needs this functionality, we only allowed libFuzzer to run in persistent mode.

We had to exclude the binary md5sum from the improvement calculation as one of the fuzzer combinations (EnFuzz-Q) did not run properly on this binary. The combination of md5sum and EnFuzz-Q are the only ones affected by this.

Experiment We run each combination 10 times for 10h and collect all corpus files to later extract all triggered bug IDs. For the evaluation, we compare the median number of unique bugs each fuzzer combination found.

Results. As shown in Table 4, the median run of Cupid finds 6.39% more bugs than EnFuzz in the same time (with the geometric mean of the improvements being +11.75%). Note that the authors of EnFuzz manually chose this selection of fuzzers (EnFuzz-Q) to specifically target the Lava-M binaries, while we did not.

6.2.4 *Summary.*

Not only does our evaluation show a strong positive correlation of Cupid's approximate ranking of fuzzers with real-world performances, but the *same* automatically selected *target-independent* combination of fuzzers by Cupid outperforms *two* expert-guided, hand-picked *target-specific* selections (for Google's fuzzer-test-suite and Lava-M).

7 RELATED WORK

Previous work on supporting fuzzing at a large scale has focused on implementing parallel fuzzer synchronisation [18, 29], where multiple parallel instances of the *same* fuzzer (e.g., AFL) share a single corpus and synchronize their efforts.

EnFuzz [5] introduces the idea of *ensemble fuzzing*, i.e., a set of fuzzers that synchronize, showing how selecting an ensemble of diverse fuzzers can increase code coverage. The authors of EnFuzz hand-pick a number of fuzzer configurations that perform best in different scenarios (e.g., EnFuzz-Q performs better on LAVA-M, while their EnFuzz selection performs better on Google's fuzzer-test-suite). In this paper, we generalize the intuition provided by EnFuzz. In contrast to EnFuzz, Cupid presents an automated, data-driven way of selecting a set of fuzzers to be used, without requiring a knowledgeable expert to manually select the set of fuzzers best suited for the task.

Previous work on hybrid fuzzing [25, 28, 30] can be considered close in nature to *ensemble fuzzing*, in the sense that these solution often contain a fast, lightweight base fuzzer (typically AFL) which delegates hard-to-solve test cases to the heavyweight symbolic execution engine. In fact, hybrid fuzzing can be expressed as an instance of ensemble fuzzing.

Xu et al. [27] have identified *fork* (mainly due to kernel-side locking mechanisms) and file system operations as bottlenecks in libFuzzer and AFL-based fuzzers in collaborative fuzzing runs. To avoid the negative impact of this on the scalability of fuzzers, they propose new operating primitives, i.e., they replace the *fork* system call, add a file system service that is specifically designed for small file operations, and introduce a new test-case syncing mechanism.

Recent work by Böhme and Falk [2] shows that finding a linear number of new bugs using parallel instances of the same fuzzer requires an exponential increase in the number of fuzzing instances. Whether this assumption holds when scaling up using a diverse set of fuzzers has not been investigated yet and would be an interesting area for future research.

8 DISCUSSION

In this paper, we have focused on having fuzzers collaborate to achieve a higher coverage. As such, in our evaluation, all the fuzzers we have showcased are mutation-oriented, coverage-guided fuzzers. We have not yet evaluated how, for instance, grammar-based fuzzers would fit into our design. Moreover, we have limited ourselves to COTS fuzzers with the explicit goal of not making significant modifications to the fuzzers. We are thus limited to the interface that the selected fuzzers provide.

In our current implementation, the set of selected fuzzers is static over the whole run. In some cases, dynamically changing the set of fuzzers over the fuzzing campaign, might yield a better result (although we have not yet found practical cases of interest). In this paper, we focus on the *selection* of fuzzers, and leave the question of *scheduling* different fuzzers as a future research topic.

Similarly, a future area for research could be *testcase scheduling and distribution*, i.e., deciding which testcase to assign to which fuzzers during runtime. A smarter approach to distributing all testcases to all fuzzers might be to determine which fuzzer would benefit the most from a particular testcase and avoid assigning sharing less beneficial ones.

As Cupid tries to make a target-independent prediction, it may be possible to make target-specific predictions as well. However, although it would be possible to collect data and train on the same fuzzing target for which the prediction is for, this target-specific approach may offer less benefit than expected. The training data gathered from the parts of the fuzzing process that were already seen may not necessarily reflect the remaining unseen parts of the same binary. Therefore, any prediction based on this data might actually be misleading. This is why Cupid is trained on a vast variety of different branches, sampled from the whole program space of different binaries and not only on the initial stages of one specific binary.

The data we collect to make our predictions are based on branch-based coverage metrics. However, recent research has shown that there are significant differences in how coverage is measured [26]. Different measurement could potentially improve our complementarity metric by taking other information into account, such as memory-access or context-sensitive information.

For our evaluation, to bound the (lengthy) required time to run the experiments, we set a time limit of 10 hours for each run. Two of the larger programs are not fully saturated within this timeout. If saturated, these programs might show a different end result. However, since these programs are a small part of the overall dataset, and we use geometric means for average improvement calculation, the overall impact is marginal. We refer the reader to the work by Klees et al. [14] for a comprehensive work on best practices when evaluating fuzzers.

9 CONCLUSION

In this paper we present Cupid, a collaborative fuzzing framework that uses single, isolated fuzzer runs to automatically predict and select, in *linear evaluation time*, a high-performing combination of fuzzers to use in a collaborative fuzzing scenario. We show how different combinations of fuzzers influence the overall result and how Cupid is capable of automatically selecting a good, complementary set of fuzzers, eliminating the need for manual selection or an exhaustive (exponentially costly) evaluation of all possible fuzzer combinations. In summary, we have shown how selecting complementary fuzzers can improve the both the final coverage, as well as the latency of finding coverage significantly. To foster further research in the area, we will open source Cupid at https://github.com/RUB-SysSec/cupid.

ACKNOWLEDGMENTS

We would like to thank our shepherd Hayawardh Vijayakumar and the anonynous reviewers for their constructive feedback. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CaSa – 390781972. In addition, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct). This paper reflects only the authors' view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.
[2] M. Böhme and B. Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *ESEC/FSE*, 2020.
[3] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *IEEE Transactions on Software Engineering*, 2017.
[4] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.
[5] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *USENIX Security Symposium*, 2019.
[6] The chromium (google chrome) open source project on open hub: Languages page. https://www.openhub.net/p/chrome/analyses/latest/languages_summary. Accessed: September 26, 2020.
[7] J. D. Evans. *Straightforward statistics for the behavioral sciences*. Thomson Brooks/Cole Publishing Co, 1996.
[8] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++ : Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.
[9] S. Geng, Y. Li, Y. Du, J. Xu, Y. Liu, and B. Mao. An empirical study on benchmarks of artificial software vulnerabilities. *CoRR*, abs/2003.09561, 2020.
[10] E. Güler, C. Aschermann, A. Abbasi, and T. Holz. Antifuzz: Impeding fuzzing audits of binary executables. In *USENIX Security Symposium*, 2019.
[11] A. Helin. A general-purpose fuzzer. https://gitlab.com/akihe/radamsa. Accessed: September 26, 2020.
[12] Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: September 26, 2020.
[13] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim. Fuzzification: anti-fuzzing techniques. In *USENIX Security Symposium*, 2019.
[14] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.
[15] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/. Accessed: September 26, 2020.
[16] C. Lemieux and K. Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. In *ACM International Conference on Automated Software Engineering (ASE)*, 2018.
[17] Y. Li, C. Feng, and C. Tang. A large-scale parallel fuzzing system. In *International Conference on Advances in Image Processing*, 2018.
[18] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2018.
[19] Libfuzzer. https://www.llvm.org/docs/LibFuzzer.html. Accessed: September 26, 2020.

[20] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014.

[21] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12), Dec. 1990.

[22] T. Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. Accessed: September 26, 2020.

[23] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[24] K. Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. In *USENIX Security Symposium*, 2017.

[25] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[26] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019.

[27] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *ACM Special Interest Group on Security, Audit and Control (SIGSAC)*, 2017.

[28] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, 2018.

[29] M. Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: September 26, 2020.

[30] L. Zhao, Y. Duan, H. Yin, and J. Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

Table 5: Configuration parameters of all fuzzers.

Fuzzer	Version	Environment Variables	Configuration
AFL	v2.52b	AFL_NO_AFFINITY=1	afl-fuzz -M/-S -i input_dir -o output_dir -m none -- binary_command
AFLFast	Commit e672d6e92	AFL_NO_AFFINITY=1	afl-fuzz -M/-S -i input_dir -o output_dir -m none -- binary_command
FairFuzz	Commit 9c1f1b366	AFL_NO_AFFINITY=1	afl-fuzz -M/-S -i input_dir -o output_dir -m none -- binary_command
radamsa	AFL++, commit 2ff174e58	AFL_NO_AFFINITY=1	afl-fuzz -M/-S -i input_dir -o output_dir -m none -R -- binary_command
libFuzzer	Commit dce08fd05 with custom patch		binary_path -fork=1 -ignore_crashes=1 -artifact_prefix=crash_dir/ cov_dir input_dir libfuzzer_simulated_sync
QSYM	AFL: v2.52b, QSYM: commit aabec86ea77	AFL_NO_AFFINITY=1	afl-fuzz -M/-S -i input_dir -o output_dir -m none -- binary_command && python /workdir/qsym/bin/run_qsym_afl.py -a docker_name -o output_dir -n qsym_name -- uninstrumented_binary_command
Honggfuzz	v2.0 with custom patch		/home/coll/honggfuzz/honggfuzz --input input_dir --workspace workspace_dir --crashdir crash_dir --covdir_all cov_dir -n 1 -y honggfuzz_simulated_sync -Y 60 -- binary_command
lafIntel	AFL++, commit 2ff174e58	AFL_NO_AFFINITY=1 AFL_PRELOAD=libcompcov.so AFL_COMPCOV_LEVEL=2	afl-fuzz -M/-S -i input_dir -o output_dir -m none -t 1000000 -Q -- binary_command

Table 6: Median branch coverage on the test binaries from fuzzer-test-suite (10 runs with a run time of 10h) when Honggfuzz and lafIntel are included in the training phase. Cupid selected the combination consisting of QSYM, libFuzzer, Honggfuzz, Honggfuzz (denoted as CupidExt). Bold values highlight the best result. The last column represents the *p*-value according to the Mann-Whitney U test between EnFuzz and Cupid. Bold values highlight statistical significance ($p < 0.05$).

Binary	EnFuzz	CupidExt	*p*-value
c-ares	**58**	**58**	-
guetzli	2612	**2657**	**< 0.01**
json	710	**711**	**0.02**
libarchive	2984	**3576**	**< 0.01**
libpng	651	**659**	**< 0.01**
libssh	800	**846**	**< 0.01**
libxml2	2110	**3265**	**< 0.01**
openssl-1.0.2d	**787**	786	0.10
openssl-1.1.0c	**778**	768	**< 0.01**
openthread	859	**866**	**< 0.01**
proj4	2800	**2896**	**< 0.01**
sqlite	**913**	907	**< 0.01**
woff2	1065	**1157**	**< 0.01**
Improvement (sum)	-	**+11.82%**	
Improvement (geomean)	-	**+6.4%**	

Table 7: The result of the framework prediction for n=4. "Complementarity" represents the predicted total probability for this fuzzer combination, i.e. the predicted average number of branches visited on all of the training data by this fuzzer combination. Combination in bold is the selection by Cupid due to complementarity and diversity.

Position	Combination	Complementarity
1.	FairFuzz,libFuzzer,libFuzzer,libFuzzer	99691.98
2.	AFL,libFuzzer,libFuzzer,libFuzzer	99404.65
	...	
26.	FairFuzz, FairFuzz, libFuzzer, radamsa	96236.07
27.	**FairFuzz, libFuzzer, QSYM, AFL**	96176.07
28.	FairFuzz, libFuzzer, AFL, AFL	96064.53
	...	
124.	AFLFast, QSYM, QSYM, QSYM	72796.42
125.	QSYM, QSYM, QSYM, radamsa	71395.88
126.	QSYM, QSYM, QSYM, QSYM	66091.62