# BigMap: Future-proofing Fuzzers with Efficient Large Maps

Alif Ahmed, Jason D. Hiser, Anh Nguyen-Tuong, Jack W. Davidson, Kevin Skadron
*Department of Computer Science, University of Virginia*
{*alifahmed, hiser, an7s, jwd, skadron*}*@virginia.edu*

***Abstract*—Coverage-guided fuzzing is a powerful technique for finding security vulnerabilities and latent bugs in software. Such fuzzers usually store the coverage information in a small bitmap. Hash collision within this bitmap is a well-known issue and can reduce fuzzers' ability to discover potential bugs. Prior works noted that collision mitigation with naïvely enlarging the hash space leads to an unacceptable runtime overhead. This paper describes BigMap, a two-level hashing scheme that enables using an arbitrarily large coverage bitmap with low overhead. The key observation is that the overhead stems from frequent operations performed on the full bitmap, although only a fraction of the map is actively used. BigMap condenses these scattered active regions on a second bitmap and limits the operations only on that condensed area. We implemented our approach on top of the popular fuzzer AFL and conducted experiments on 19 benchmarks from FuzzBench and OSS-Fuzz. The results indicate that BigMap does not suffer from increased runtime overhead even with large map sizes. Compared to AFL, BigMap achieved an average of 4.5x higher test case generation throughput for a 2MB map and 33.1x for an 8MB map. The throughput gain for the 2MB map increased further to 9.2x with parallel fuzzing sessions, indicating superior scalability of BigMap. More importantly, BigMap's compatibility with most coverage metrics, along with its efficiency on bigger maps, enabled exploring aggressive compositions of expensive coverage metrics and fuzzing algorithms, uncovering 33% more unique crashes. BigMap makes using large bitmaps practical and enables researchers to explore a wider design space of coverage metrics.**

## I. Introduction

Real-world applications usually have a large codebase, making it difficult to detect security vulnerabilities while providing a vast attack surface to the adversaries. Fuzzing techniques are geared towards automatically generating test vectors to expose these vulnerabilities or to improve the code coverage in general. Among different types of fuzzer, black-box fuzzers blindly generate random test vectors without resorting to any form of program analysis. Consequently, these fuzzers scale very well with program size and are easily parallelizable but are unlikely to find rare bugs. On the other end of the spectrum, white-box fuzzers can do a directed and exhaustive search of the coverage space with symbolic execution, but are prohibitively slow to be useful for large, real-world applications [1]–[4]. Coverage-guided grey-box fuzzers fill the middle ground and so far have been most successful in finding software bugs. At the time of this writing, Google's OSS-Fuzz platform uncovered over 20,000 vulnerabilities on 300 projects [5] with the help of three coverage-guided grey-box fuzzers - libFuzzer [6], Honggfuzz [7], and American Fuzzy Lop (AFL) [8].

As the name suggests, coverage-guided fuzzers use some form of a coverage metric to track and guide their test generation process. For example, libFuzzer and Honggfuzz use basic block coverage. AFL, on the other hand, tracks edge hit counts with the help of a coverage bitmap. Each edge encountered while executing a test case is dynamically assigned to a location on this bitmap to store and update the hit count. This coverage bitmap is accessed very frequently and should occupy faster cache levels to maximize the test case generation throughput. For this reason, the size of the bitmap has historically been kept small (the default size is 64kB for AFL). Due to the bitmap's size limitation and the randomness of the location assignment, it is possible to have hash collisions, where two or more edges point to the same location on the bitmap. Hash collisions introduce ambiguity in coverage feedback and can severely limit the fuzzer's ability to find bugs [9]. Our work seeks a better understanding and efficient mitigation of this issue.

The straightforward way for reducing hash collisions is to expand the hash space (i.e., increase coverage bitmap size). Prior works noted that naïvely enlarging the bitmap can severely diminish the test case generation throughput, potentially resulting in lower code coverage within the same time budget [9]. We investigated the reason behind the throughput drop with larger bitmaps. We observed that for large bitmaps, most of the time is spent doing a few specific operations (e.g., reset, classify, compare, and hash) on the bitmap. These operations are performed on the full bitmap, although only a small fraction of the bitmap is actively used for storing coverage statistics. This type of access pattern is inefficient and heavily pollutes the processor's data cache, ultimately lowering the throughput. In this paper, we introduce BigMap, a *two-level* bitmap scheme that optimizes these map operations. BigMap adds an extra level of indirection to bitmap accesses to condense randomly scattered coverage metrics in a sequential bitmap, vastly improving cache locality behavior. Furthermore, the map operations now only need to be performed on the used portion instead of the full bitmap. Overall, our proposed approach enables using large maps without sacrificing throughput.

We integrated BigMap into AFL and conducted experiments with benchmarks from FuzzBench [10] and OSS-Fuzz [5]. With AFL's carefully tuned default map size of

64kB, BigMap demonstrated identical throughput, despite adding an extra level of indirection. The throughput gain over AFL increased with map size, with up to 13.6x (average of 4.5x) for a 2MB map and up to 114x (average 33.1x) for an 8MB map. BigMap also demonstrated better scalability with concurrent fuzzing instances, achieving an average of 9.2x higher throughput than AFL for a 2MB map and up to 12 parallel instances. The higher throughput resulted in uncovering 37% more unique crashes on average.

Interestingly, BigMap is compatible with any coverage metric (not just edge hit count) as long as it uses some form of a coverage bitmap. This property, along with the efficiency of BigMap with large maps, enables exploring aggressive compositions of coverage metrics and algorithms previously thought infeasible. To demonstrate this capability of BigMap, we selected a few large applications from OSS-fuzz [5] as seed benchmarks. Their discoverable edges are further amplified by enabling laf-intel transformations[1] [11] and then combining it with a more expressive coverage metric, N-gram [12]. This combination resulted in fuzzing harnesses with over 600k discoverable edges (over 5.5 million static edges). To put it into context, typical real-world applications have around 1k - 50k discoverable edges [13]–[15]. After mitigating hash collisions on these benchmarks with the help of BigMap, we saw a 33% increase in the number of unique crashes. In summary, this paper makes the following contributions:

- We investigate the shortcomings of enlarging bitmap to mitigate hash collision and identify the following key reasons: frequent operations on the full coverage map and excessive cache pollution.
- We leverage our findings in designing BigMap. BigMap introduces a two-level mapping scheme to limit the operations on the used region of the map. With this adaptive technique, the bitmap can be made arbitrarily large without sacrificing speed.
- We extend base AFL with our proposed method. Compared to AFL, we see an average of 4.5x higher test case generation throughput for a 2MB map and 33.1x for an 8MB map.
- We evaluate the scalability of BigMap with concurrent fuzzing instances. Compared to AFL, we see an average throughput gain of 9.2x for a 2MB map. Furthermore, BigMap was able to uncover 37% more unique crashes.
- BigMap's enables the aggressive composition of coverage metrics. We evaluate the composition of two well-known coverage metrics, laf-intel and N-gram, and find that unique crash coverage improved by 33%.

The BigMap fuzzing framework is open-source and available at: https://github.com/alifahmed/BigMap.

[1]Laf-intel transforms multi-byte comparisons into a cascade of single-byte comparisons. Laf-intel also deconstructs switch statements and strcmp/memcmp functions into if-else statements.

## II. Background

### A. American Fuzzy Lop (AFL)

AFL is one of the most popular fuzzers currently available. Many prior works [9], [16], [17] built upon AFL, including our work in this paper. AFL uses an evolutionary algorithm for fuzzing. Figure 1 illustrates this flow. In general, this workflow is applicable to other coverage-guided fuzzers as well. At the beginning of the process, AFL instruments the target application and populates the seed pool with user-provided seed inputs. Afterwards, AFL enters a fuzzing cycle: i) Selects a seed from the seed pool for mutation. ii) Mutates the seed to generate many new test cases. A seed is usually mutated tens of thousands of times before moving to the next seed. iii) Executes the generated test cases and checks the coverage feedback. If any test case crashes or hangs, it is reported to the user. If a test case covers an interesting path dictated by the fitness function (e.g., improves coverage), it is added to the seed pool as a potential candidate for future mutations. Otherwise, the test case is discarded. iv) After finishing with the current seed, the flow goes to (*i*) and selects a new seed for mutation. Fuzzing cycle continues until the user interrupts, or some other criteria is met (e.g., coverage goal or time budget).

*1) Seed Scheduling and Mutation:* This section is kept short because our approach is orthogonal to the seed scheduling and mutation strategy. Seed scheduling policy determines which seed from the seed pool will be fuzzed next. AFL prioritizes the seeds based on their execution speed and input file length. Short input files are preferred because a mutation is more likely to touch important control structures and not just redundant data blocks on a smaller file [13]. As for mutating the seed, AFL applies a few deterministic (i.e., not random) mutation steps followed by random mutations. The mutation steps involve bit-flips, block substitution, splicing, etc. The deterministic mutation steps usually take a long time to finish. It is a common

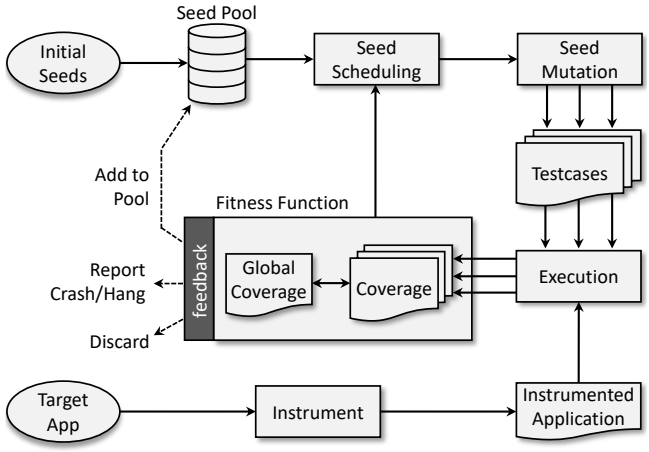


Figure 1: The generic workflow of a coverage-guided fuzzer.

practice to skip this deterministic stage and directly apply random mutations for shorter runs (e.g., 24 hours).

*2) Execution and Coverage Feedback:* AFL collects the coverage of a test case with the help of the instrumented target. The exact execution path is not tracked. Instead, a coarse-grained edge hit count is used as the coverage metric [13]. The edges are identified as a hash of the (source_block, destination_block) tuple. Listing 1 shows the necessary steps.

```
1  B_X , B_Y = random % MAP_SIZE <COMPILE TIME>
2  E_XY = (B_X >> 1) ⊕ B_Y
3  coverage_bitmap[E_XY]++
```

Listing 1: Instrumentation capturing the hit counts of $E_{XY}$.

Here MAP_SIZE is the size of the coverage bitmap. $B_X$ and $B_Y$ are the source and the destination basic block IDs, respectively. $E_{XY}$ is the ID corresponding to the $X \rightarrow Y$ edge. Basic block IDs are assigned at compile time following a discrete uniform distribution over the [0..MAP_SIZE) range. On the other hand, edge IDs are calculated at runtime and also falls within [0..MAP_SIZE). The shift operation in the edge ID calculation makes it possible to preserve the directionality of the edges (e.g., $E_{XY} \neq E_{YX}$). It also helps in properly identifying distinct tight loops (e.g., $E_{XX} \neq E_{YY} \neq 0$). AFL sports an alternative technique for getting edge IDs that leverages the *trace-pc-guard* coverage sanitizer of the Clang compiler [18]. In this method, the Clang compiler itself instruments static edges without any need to instrument at the basic block level. Unfortunately, this method cannot detect indirect edges as the target basic block information is unavailable at compile time.

Irrespective of how the edge IDs are generated, they act as an index to the coverage bitmap. The corresponding byte at that index stores the desired statistics (e.g., hit count for vanilla AFL) of that particular edge. The following steps are performed to collect the coverage of individual test cases:

- **Bitmap reset:** The coverage bitmap is a shared data structure and is used by all the test cases. Thus, before executing a test case, the coverage bitmap is cleared to remove any artifact of previous runs. A simple memset to zero does this job.
- **Bitmap update:** The instrumented target executes the test case and records the edge hit counts on the bitmap.
- **Bitmap classify:** The exact hit counts are converted to coarse hit counts by mapping them into buckets. The buckets used by AFL are: [1], [2], [3], [4-7], [8-15], [16-31], [32-127], [128,$\infty$]. Hit counts that fall into different buckets are considered as an interesting change in the control flow. Change within the same bucket is ignored. Bucketing also mitigates the impact of accidental hash collisions.
- **Bitmap compare:** After the classify step, the modified bitmap is compared with a global coverage bitmap that keeps track of all the edges covered so far. Newly discovered edges, if any, are added to the global coverage map at this point. If the test case crashes/hangs instead, it is compared to a global crash/hang coverage bitmap.
- **Bitmap hash:** If the test case is considered interesting, a hash of the bitmap is calculated and saved for rapid comparison in the future.

Since these bitmap operations are performed for every test case (expect bitmap hash, which is performed for every *interesting* test case), it is crucial to minimize the time spent on these operations. One way to facilitate faster bitmap operations is to keep the bitmap size small. This limitation on map size leads to a high number of hash collisions. As stated earlier, collisions introduce ambiguity in coverage feedback and may result in discarding interesting test cases. This paper's primary objective is to enable large coverage bitmaps (thus reducing hash collisions) without incurring associated runtime overhead.

### B. Collision Rate

In our work, the severity of the hash collision is quantified using the ***collision rate*** metric. Consider drawing $n$ keys from a hash space of size $H$. Among the $n$ draws, if $c$ number of key matches with one of the previously drawn keys, then the collision rate is defined as $c/n$ (where $c < n$). If the key draw follows a discrete uniform distribution, then the collision rate can be expressed using Equation 1.

$$CollisionRate(H, n) = 1 - \frac{H}{n}\left[1 - \left(\frac{H-1}{H}\right)^{n}\right] \quad (1)$$

Equation 1 is consistent with how AFL generates the block and edge IDs. Here, the hash space size $H$ is analogous to the coverage bitmap size, and the number of drawn keys $n$ is equivalent to the number of generated IDs.

Note that the collision rate does not indicate the actual number of keys with collision. Consider an example where the following keys are sequentially drawn: $\{4, 2, 5, 3, 2\}$. Here, the collision rate is $1/5$ and not $2/5$. Although the given collision rate definition does not account for all the colliding keys, we have used it to remain consistent with the existing literature [9], [13].

## III. Implication of Naïve Hash Collision Mitigation Strategy

Hash collisions can be completely avoided by assigning unique IDs to every discoverable edge. Otherwise, traversing two (or more) different edges will update the same location in the coverage bitmap. Unfortunately, assigning unique IDs may not always be possible. AFL's default bitmap size is 64kB, where each byte stores the statistics of an edge. Thus, even in the best scenario, at most 64k edges can be assigned with different IDs. Any more than that, and collision will be unavoidable. The birthday problem suggests that the

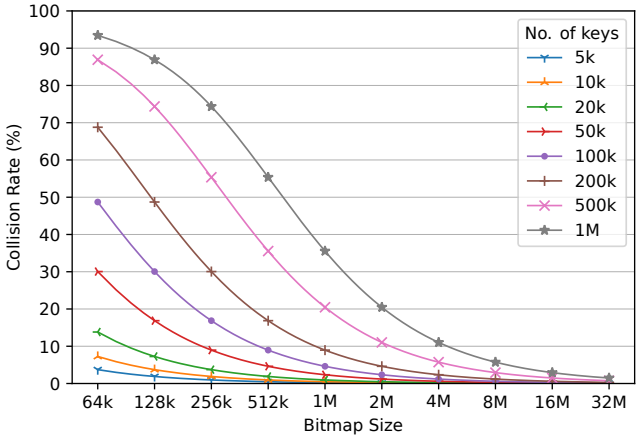


Figure 2: Hash collision rate drops as bitmap size is increased (derived from Equation 1).

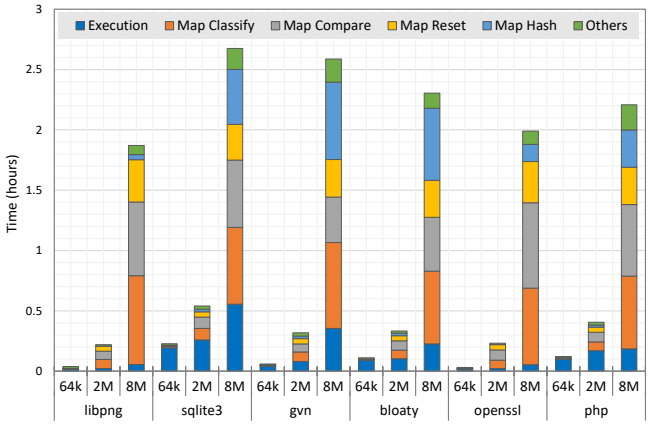


Figure 3: Runtime composition with varying bitmap sizes. Map operations dominate the runtime for bigger maps. The reported time is for one million test case generation.

collision is likely to occur with significantly less than 64k edges [19]. Assuming a uniform distribution of the edge IDs within the 64kB bitmap range, the probability of having at least one collision is ~50% after assigning only 300 IDs.

Similar to edge IDs, block IDs are also randomly generated within the [0..MAP_SIZE) range (Listing 1). Thus, it is quite possible to have more than one basic block with the same ID. Edges originating from or entering these colliding blocks will point to ambiguous locations in the coverage bitmap. Bucketing the hit counts provides some protection against such accidental hash collisions. Having too many collisions still severely limits the fuzzer's ability to guide its fuzzing process by providing incorrect coverage feedback.

The straightforward way of reducing hash collisions is to expand the hash space (i.e., use a larger bitmap). Figure 2 shows the collision rates with different bitmap sizes and the number of keys drawn (derived from Equation 1). The keys here are analogous to the discoverable edges and blocks. For real-world applications, the number of discoverable edges usually ranges from 1k to 50k. As a result, a 64kB map is subjected to ~30% collision rate. Using more thorough coverage metrics like full/partial path coverage [12], context-sensitive edge coverage [17], or branch condition transformations [11] can make the required number of IDs go well over 500k. These techniques can be stacked, further increasing the collision rate. We need a much larger map than 64kB if we want to explore these techniques without worrying about hash collisions.

### A. Cost of Expanding Hash-space

The bitmap should be much larger than the number of required IDs to keep the collision rate in check. Unfortunately, increasing bitmap size also increases the runtime overhead of the bitmap operations. Figure 3 shows the runtime composition for six benchmarks with 64kB, 2MB, and 8MB bitmap sizes. For the small 64kB map, the fuzz target's execution time dominates the overall runtime. The costs of bitmap operations are negligible at this point. On the other hand, the runtime is dominated by the bitmap operations for the larger 8MB map. The **classify**, **compare**, and **reset** operations require iterating through the full bitmap for *every* test case. As a result, they are impacted most by the increase in bitmap size. Bitmap **hash** operation also needs to go through the full bitmap but is only performed on the *interesting* test cases. Therefore, the overhead of hash operation varies considerably depending on the benchmark. There are a few other bitmap operations not shown in this figure, simply because they are too infrequent to have any perceivable impact on the runtime.

## IV. BigMap: Adaptive Two-Level Bitmap

In the AFL's data structure for coverage tracking, the keys are randomly distributed throughout the bitmap. Figure 4(a) shows an example where the edge ID $E_{XY}$ is used as the key to access the coverage map. In this example, only five of the twelve locations are modified. However, since there is no information on exactly where these modified locations are, the bitmap operations like reset, classify, compare, etc., have to traverse the complete map. We propose the use of a two-level bitmap scheme to consolidate these scattered accesses.

### A. Two-Level Bitmap Scheme

In our proposed scheme, the consolidation process is carried out during the bitmap update phase by maintaining three data structures: i) A $coverage\_bitmap$ that holds the coverage statistics. ii) An $used\_key$ that points to the next available space in the coverage_bitmap. iii) An $index\_bitmap$ that maps an edge ID to a location in the coverage_bitmap. Figure 4(b) demonstrates the update steps. First, we query index_bitmap[$E_{XY}$] to get the location of the stored hit count. If the edge is encountered for the first

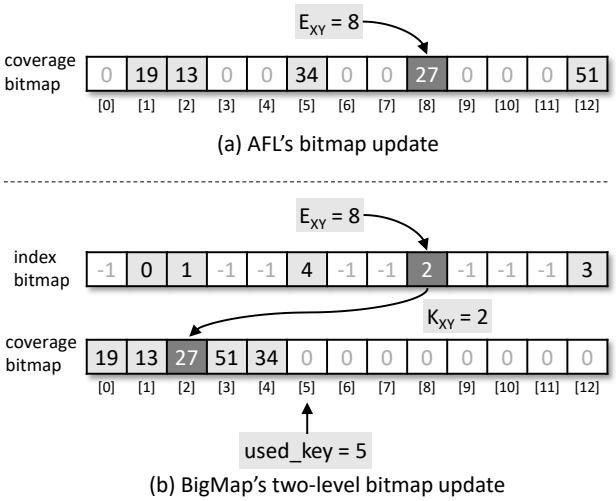


Figure 4: Steps of bitmap update operation for AFL's and BigMap's data structure. The hit counts in the coverage_bitmap are scattered in (a), while consolidated in (b).

time, we will get an invalid location (-1 in our implementation). In this case, the index_bitmap[$E_{XY}$] is assigned to the next available location in the coverage_bitmap (= used_key). Once we have the location, the hit count in the coverage_bitmap is incremented.

As depicted in Figure 4, BigMap's scheme makes the coverage statistics contained within the first used_key locations, unlike AFL. Therefore, all the bitmap operations (except bitmap update) need to iterate over the [0..used_key) range instead of the full bitmap. As a result, the runtime of the map operations will depend on how many edges are discovered instead of how big the coverage bitmap is. An interesting aspect of this solution is its adaptive nature, where the default bitmap size can be arbitrarily large irrespective of the target application's size. Applications with a large number of discoverable edges will benefit from hash collision mitigation, and applications with few discoverable edges will not incur any significant overhead despite having large map structures. This flexibility helps in situations when it's difficult to assess the optimal map size in advance.

### B. Illustrative Example

Figure 5 shows a step by step example of how the map operations are performed. We will focus on BigMap and will contrast it with AFL towards the end of this section.

At the beginning of the fuzzing session, BigMap initializes the index_bitmap to -1, indicating none of the edges are assigned any location yet. The hit counts in coverage_bitmap are also set to zero. This initialization is performed a single time during the whole fuzzing campaign, and it is the only time BigMap accesses the full bitmaps. At this point, the index_bitmap and the coverage_bitmap are ready to capture the test case's coverage information. The used portion (none for the first run) of the coverage_bitmap is reset before each test is executed. During execution, the index_bitmap is updated as new edges are being discovered. Corresponding locations in the coverage_bitmap is also updated. After the execution is finished, the hit counts are bucketed and compared with the global coverage maps. If the test case is deemed interesting by the fuzzer, additional bitmap operations such as hashing, rank update, etc., may be performed. These steps read/modify only the used portion of the coverage_bitmap as well.

A few things to note here. The index_bitmap is touched only during the update phase. It is not accessed at any other phase, including reset. Therefore, the same edge will point to the same coverage_bitmap location for all the test cases. Also, the update phase is the only stage where AFL's data structure is more efficient. AFL's structure does one data access per edge compared to two accesses of BigMap's structure. Fortunately, the extra access shows good cache locality, as will be discussed in the next section.

### C. Access Patterns of the Bitmap Operations

*1) AFL's Data Structure:* Table I(a) summarizes the access patterns for AFL's data structure. The bitmap update does sparse access over the coverage_bitmap. These accesses correspond to the IDs of the encountered edges. It has a high temporal locality because the same edges are likely to be traversed again within the same program execution (e.g., edges inside loops or common functions). The same edges are also likely to be traversed across different executions due to the overlap of execution paths.

The rest of the bitmap operations iterates the full map. Most of these locations do not contain any useful information, therefore causes heavy cache pollution. In turn, the cache pollution may trigger the eviction of useful data to slower cache levels or memory. For example, pollution may

Table I: Access Patterns of the Bitmap Operations

(a) AFL's Data Structure

| Map Operation | Bitmap | Access to | Temporal locality | Spatial locality | Cache pollution |
|---|---|---|---|---|---|
| Update | Coverage | Used map[2] | High | Low | Low |
| Others[1] | Coverage | Full map | Low | High | High |

(b) BigMap's Data Structure

| Map Operation | Bitmap | Access to | Temporal locality | Spatial locality | Cache pollution |
|---|---|---|---|---|---|
| Update | Index | Used map[2] | High | Low | Low |
| | Coverage | Used map[2] | High | High | None |
| Others[1] | Index | None | – | – | None |
| | Coverage | Used map[2] | High | High | None |

[1] Bitmap reset, compare, classify, hash etc.

[2] Corresponds to the highlighted cells in the example of Figure 5.

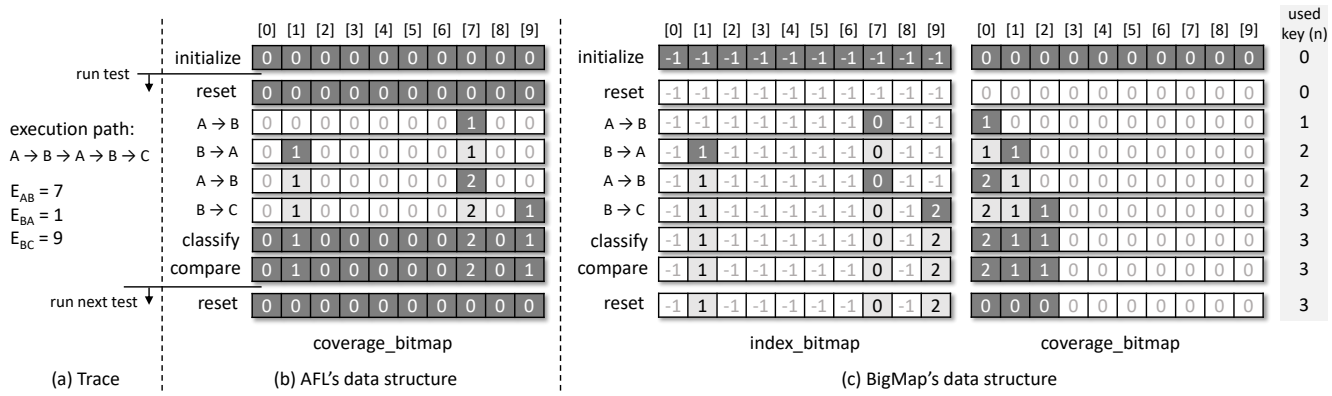


Figure 5: An illustrative example of bitmap operations on AFL's and BigMap's data structures. Value on top is the index of the bitmaps. Locations accessed at each step are highlighted in bold. (a) Execution trace and the assigned edge IDs (random). (b) AFL's data structure. Reset, classify, compare, etc., operations need to access the full bitmap. (c) BigMap's data structure. The full map is accessed only during initialization. Afterward, reset, classify, compare, etc., accesses only the used region of the coverage bitmap. Index bitmap is only accessed during the hit count update.

prevent keeping common edge locations in L1/L2 cache across consecutive executions.

*2) BigMap's Data Structure:* Table I(b) shows the access patterns for BigMap's data structure. During the update operation, BigMap's structure makes two accesses per edge, first to the index_bitmap and then to the coverage_bitmap. Access to the index_bitmap is scattered and is identical to the pattern of AFL's data structure. On the other hand, access to the coverage_bitmap has a high spatial and temporal locality. The spatial locality stems from the fact that the edge hit counts are now residing in close vicinity. The rest of the bitmap operations do sequential access to the coverage_bitmap, exhibiting high spatial and temporal locality. We infer high temporal locality for BigMap's structure and not for AFL's structure. This is because AFL's structure has a high reuse distance as it accesses the full map. Overall, BigMap's structure demonstrates vastly improved cache locality behaviors compared to AFL.

### D. Implementation Details

The BigMap approach requires minor modifications of AFL's instrumentation to support the two-level bitmap update. The new instrumentation is shown in Listing 2.

```
1  B_X , B_Y = random % MAP_SIZE <COMPILE TIME>
2  E_XY = (B_X >> 1) ⊕ B_Y
3  if (index_bitmap[E_XY] == -1)
4      index_bitmap[E_XY] = used_key++
5  K_XY = index_bitmap[E_XY]
6  coverage_bitmap[K_XY]++
```

Listing 2: BigMap instrumentation for map update.

Here, lines 1, 2, and 6 are identical to AFL's instrumentation scheme (Listing 1). Lines 3-5 are added to query and modify the index_bitmap. Since these instructions are executed for every edge, overhead can be a big concern. Given the rarity of new edge discovery, most of the time, the overhead will consist of one branch condition check (at line 3) and one extra access to the index bitmap (at line 5). The branch condition outcome is highly skewed towards not-taken and will be predicted correctly by the branch predictor almost always. Furthermore, the access to the index_bitmap is amenable to hit the L1 or L2 cache, making the access time negligible. The index_bitmap update (at line 4) will be invoked only when a new edge is discovered for the first time. Interestingly, while the index_bitmap is indexed by the edge ID, it does not necessarily have to be the case. In fact, any coverage metric can be used in edge ID's place, trivializing the integration process.

The modification in the instrumentation takes care of the bitmap update operation. Further adjustments are required to support the rest of the bitmap operations. It primarily involves changing the iteration count from the full map size to *used_key*. Bitmap hash operation is an exception to this rule. AFL uses CRC32 for calculating bitmap hash. If we always calculate hash in the [0..used_key) range, it might lead to wrong hash values. Consider the following example with three test case executions:

| Execution Path | used_key | coverage_bitmap | Bitmap Hash |
|---|---|---|---|
| P1: $A \rightarrow B \rightarrow C$ | 2 | {1,1,0,0,...} | crc32({1,1}) |
| P2: $A \rightarrow B \rightarrow C \rightarrow D$ | 3 | {1,1,1,0,...} | crc32({1,1,1}) |
| P3: $A \rightarrow B \rightarrow C$ | 3 | {1,1,0,0,...} | crc32({1,1,0}) |

The hash of first case is $crc32(P1) = crc32(\{1,1\})$. Here, $\{1,1\}$ are the hit counts up to the used_key. While executing the second case, the used_key will be incremented to 3. Therefore, the hash of third case will be calculated as $crc32(P3) = crc32(\{1,1,0\})$. The first and third paths are essentially the same. However, the calculated hash values

do not match because $crc32(\{1,1\}) \neq crc32(\{1,1,0\})$. To avoid such discrepancy, BigMap calculates the hash up to the last non zero value in the coverage_bitmap.

### E. Additional Optimizations

We carried out a few additional optimizations to make our implementation faster. These optimizations are orthogonal to the two-level bitmap scheme and can be adopted by any AFL based fuzzers. First, we merged the bitmap classify and compare steps. The bitmap compare operation almost always follows the classify operation. Because these operations are carried out in the same region of the bitmap, they can be easily merged. This merging allows more efficient use of cache and cuts the cost of (compare + classify) to half. The second optimization is to replace normal reset operation with a non-temporal version. The reset operation happens just before the execution and can pollute cache with regions of the bitmap that are never used. Using non-temporal stores prevents this pollution. This optimization is only beneficial to the vanilla AFL because BigMap already limits the map operations to the used region. Our final optimization is to allocate the index and coverage bitmap using the OS-provided facility for huge pages. There are limited numbers of slots on L1/L2 DTLB, and a large bitmap can consume many of them, resulting in frequent page-walks caused by DTLB misses. Allocating the bitmaps on a huge page reduces these overheads.

## V. Evaluation

We evaluated our proposed approach in three steps. First, we demonstrated that BigMap could support larger maps without sacrificing test generation throughput, unlike standard AFL. Second, we investigated BigMap's ability to support coverage metric compositions and whether that leads to practical benefits in terms of improved code coverage. This step also acts as a justification for using large coverage maps. Finally, we evaluated the scalability of both fuzzers with respect to the number of concurrent fuzzing instances.

### A. Experimental Setup

*1) System Configuration:* The experiments were conducted on a system with two Intel Xeon E5645 CPUs (totaling 12 physical cores) clocked at 2.40GHz. Each fuzzing instance was pinned to a separate physical core with a private 32kB L1 data cache, 256kB unified L2 cache, and a shared 12MB L3 cache. Fuzzers were run for 24 hours. Because the run time is relatively short, the deterministic fuzzing step is skipped, and the fuzzers were configured to run in persistent mode. Persistent mode enables feeding multiple inputs in a loop and does not have any fork() call or initialization overheads, thus significantly boosts the test execution rate. This setup is adopted from FuzzBench [10]. As for the instrumentation mode, we used the *afl-clang-fast* that leverages an LLVM compiler pass to inject the instrumentation code. This mode is faster than the gcc-based or coverage-sanitizer based alternatives [13]. Optimizations mentioned in Section IV-E applied to both AFL and BigMap.

Table II: Benchmark Characteristics

| Benchmark | Number of seeds | Discovered edges[1] | Collision rate[2] (%) | Static edges[3] | Version |
|---|---|---|---|---|---|
| zlib | 77 | 722 | 0.55 | 875 | v1.2.11 |
| libpng | 1 | 1,218 | 0.92 | 2,987 | v1.6.35 |
| systemd | 6 | 2,314 | 1.74 | 53,453 | v245 |
| libjpeg | 1 | 2,928 | 2.20 | 9,542 | v2.0.4 |
| mbedtls | 1 | 5,377 | 3.99 | 10,942 | v2.21.0 |
| proj4 | 43 | 6,379 | 4.71 | 7,830 | v6.3.1 |
| harfbuzz | 58 | 8,930 | 6.51 | 10,021 | v2.6.4 |
| libxml2 | 1 | 9,422 | 6.86 | 50,327 | v2.9.10 |
| openssl | 2,241 | 10,297 | 7.46 | 45,989 | v1.0.2u |
| bloaty | 94 | 10,536 | 7.62 | 89,658 | v1.0 |
| curl | 31 | 12,728 | 9.11 | 62,523 | v7.68.0 |
| php | 2,782 | 20,260 | 13.98 | 123,767 | v7.4.3 |
| sqlite3 | 1,256 | 40,948 | 25.64 | 45,136 | v3.31.1 |
| licm | 101 | 64,317 | 36.29 | 977,899 | v10.0.1 |
| gvn | 140 | 65,781 | 36.89 | | |
| strength-reduce | 122 | 76,065 | 40.83 | | |
| indvars | 174 | 82,105 | 42.98 | | |
| loop-vectorize | 345 | 108,231 | 51.06 | | |
| instcombine | 1,046 | 131,677 | 56.90 | | |

[1] Maximum edge coverage among all fuzzing configurations.
[2] With a 64kB map.
[3] Derived using SanitizerCoverage [18].

*2) Benchmarks:* We used 19 benchmarks in our experiments. The characteristics of these benchmarks are given in Table II. The first 13 benchmarks are taken from FuzzBench [10]. These benchmarks are relatively small and have low collision rates. The remaining six benchmarks are LLVM optimization passes collected from OSSFuzz [5]. These benchmarks share the same LLVM-opt binary [20], and different fuzzing harnesses are selected via command-line arguments. The LLVM-opt binary itself has a high number of static edges. Collectively, the benchmarks span a wide range of discoverable edges ($\sim$1k - 131k) and collision rates.

*3) Performance Metrics:* The following metrics are used to evaluate the performance of our approach:

- ***Test case generation throughput or execution rate:*** Denotes the number of test cases evaluated by the fuzzer per unit time. Everything else being equal (e.g., seed selection and mutation strategy), a fuzzer with a higher throughput is expected to give better coverage.
- ***Unique crashes:*** AFL has a built-in deduplication mechanism for finding unique crashes. AFL considers a crash unique if it covers an edge unseen by the previous crashes or does *not* cover an edge common in all the previous crashes. This mechanism requires maintaining a local and global *crash-coverage* bitmap,

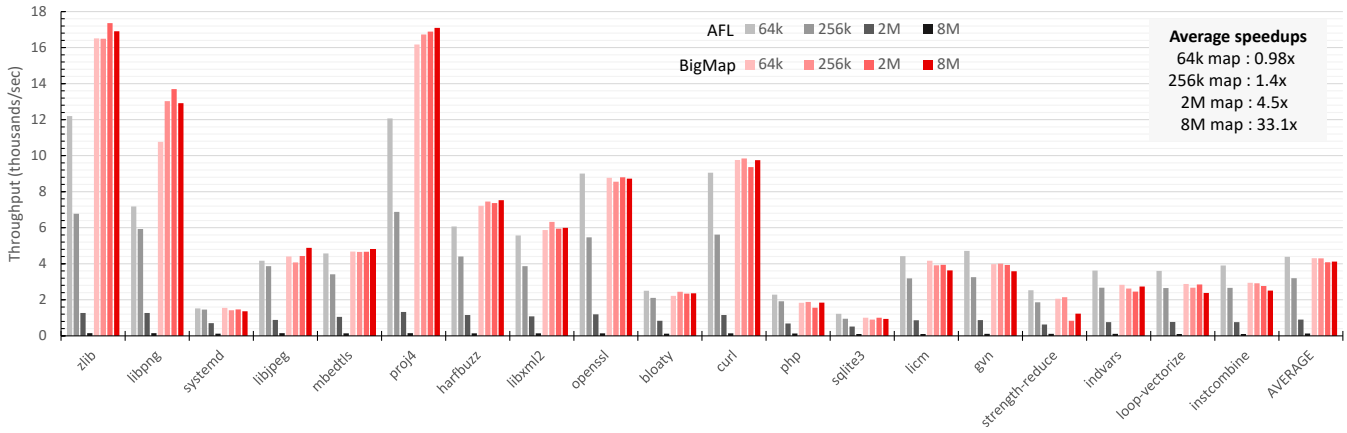


Figure 6: Test case generation throughput of AFL and BigMap with different map sizes. AFL's throughput drops significantly as the map size is increased. Map size variation has considerably less impact on BigMap.

making it inherently biased towards larger maps. To avoid this bias, we resorted to Crashwalk [21], which takes the hash of the call stack and the faulting address for deduplicating crashes.

- ***Edge Coverage:*** While crash coverage is the proper way of quantifying a fuzzer's performance, crashes in a program are typically sparse. Therefore, in addition to crash coverage, we also report edge coverage. Intuitively, a fuzzer that covers more edges are also likely to discover more bugs. To get the edge coverage, we collected the output corpus of the fuzzers and subjected them to a bias-free independent coverage build.

### *B. Evaluating the Impact of Map Size Variation*

We claimed that BigMap performs efficiently regardless of the size of the coverage bitmap. This section validates the claim by comparing BigMap's performance with AFL for four different map sizes: 64kB, 256kB, 2MB, and 8MB. In this experiment, we used an average of three runs to reduce the variations introduced by random mutation steps.

*1) Impact on Test Case Generation Throughput:* The test case generation throughput of AFL and BigMap is shown in Figure 6. As expected, AFL's throughput dropped dramatically as the map size is increased. On average, AFL's throughput went from 4,400/sec for a 64kB map to only 125/sec for an 8MB map. BigMap handled large maps gracefully without any significant drop, and the average throughput remained consistently above 4,100/sec irrespective of the map size.

***For the 64kB map***, BigMap usually outperformed AFL for smaller benchmarks (e.g., zlib, libpng, proj4), while AFL performed better on larger benchmarks (e.g., sqlite3, indvars, instcombine). This outcome is because only a tiny portion of the 64kB map is used for the small benchmarks. In such cases, BigMap gained the advantage by traversing the used region of the map. On the other hand, larger benchmarks almost completely filled the 64kB map. As a result, BigMap and AFL performed nearly the same during the classify, compare, and reset stages, but BigMap is ultimately slightly slower due to the extra indirection overhead during the update stage. One thing to note here is that the nearly full map also implies very high collision rates, suggesting the use of maps bigger than 64kB would be beneficial. Other factors impacting the throughput include the working-set size and access-pattern of the benchmark itself.

***For larger maps***, BigMap universally provided higher throughput than AFL. The 8MB map, in particular, incurred an extremely high performance hit for AFL. This performance hit is because, with an 8MB map, the combined size of the local and global coverage maps exceeded the last-level cache capacity of our experimental setup. Note that for a few benchmarks (e.g., libpng, proj4, libjpeg etc.), BigMap attained higher throughput at larger map sizes. We attribute this behavior to the various non-deterministic steps applied throughout the fuzzing process. As mentioned before, we have aggregated multiple runs to reduce the impact of randomness. Still, a fuzzing run can produce test cases that exercise longer (or shorter) execution paths more frequently relative to other runs.

On average, BigMap attains 0.98x, 1.4x, 4.5x, and 33.1x higher throughput for 64kB, 256kB, 2MB, and 8MB maps, respectively. We conclude that BigMap might not be an attractive choice for small map size of 64kB. However, if larger map is required (e.g., for reducing hash collisions or to support complex coverage metrics), then BigMap clearly provides superior test generation throughput.

*2) Impact on Edge Coverage and Unique Crashes:* Figure 7 shows the edge coverage with map size increase. During a fuzzing campaign, the rate of discovering new edges is initially high and then flattens out as time progresses. Our results indicate that BigMap reached the plateau for all of the benchmarks within the 24 hour time budget.

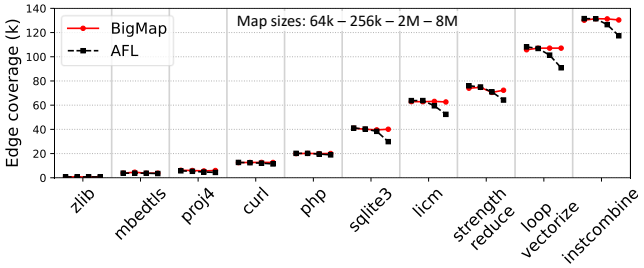


Figure 7: Edge coverage with varying map sizes. AFL's edge coverage suffers due to throughput loss with bigger maps. Not all benchmarks are shown to improve clarity.

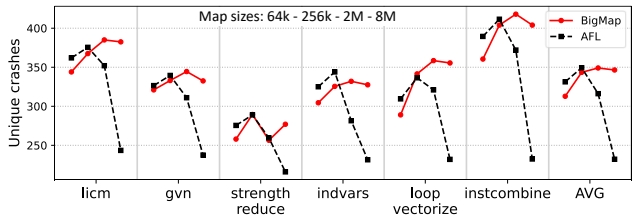


Figure 8: Unique crashes found with varying map sizes. Going from 64kB to 256kB map shows improvement as a result of reduced collisions. AFL suffers for bigger map sizes due to throughput loss.

AFL performed identically for small benchmarks. However, AFL's low throughput on bigger maps prevented it from reaching the plateau for benchmarks with a higher number of discoverable edges. Note that mitigating the hash collision was not particularly beneficial at improving the edge coverage. A public report available on FuzzBench indicates that the metric edge coverage has a relatively small variance across a wide range of fuzzers [22]. Furthermore, AFL authors noted that edge count bucketing provides some protection against collisions [13]. We hypothesize that the inherent small variance and the binning process made the edge coverage relatively insensitive to collisions.

Crashes, on the other hand, are extremely sparse and do not follow any simple pattern. We were able to find crashes on the bloaty and the LLVM benchmarks. For bloaty, we found one unique crash on all configurations except for AFL 8MB. The number of unique crashes found on the LLVM benchmarks is given in Figure 8. From this figure, it is evident that AFL performed its best with a 256kB map. The smaller 64kB map prevented finding more crashes due to collisions, while larger maps of 2MB and 8MB caused excessive runtime overhead, leading to low crash coverage. Interestingly, the *optimal* map size is unknown beforehand and may vary with the target application. Therefore, to find the most crashes, AFL has to run the target application with different map sizes (or have access to an oracle). However, testing with multiple map sizes will consume valuable compute time that may have been better utilized otherwise (e.g., longer runs or multiple instances with cooperative fuzzing). Finding the optimal map size is less of an issue for BigMap as we can choose an arbitrarily large map size with little runtime penalty. This adaptive nature of BigMap makes it an attractive choice.

### C. Evaluating Coverage Metric Composition

Previously we mentioned how BigMap's efficiency with large maps enables an aggressive composition of multiple coverage metrics. In this section, we investigate one such scenario by stacking laf-intel [11] and N-gram [12]. Because the original AFL does not have in-built support for laf-intel and N-gram, we implemented BigMap on a community-maintained version of AFL called AFLPlusPlus [23]. We used all the LLVM fuzzing harnesses available on OSS-Fuzz as benchmarks. The laf-intel transforms each multi-byte comparison into a series of single-byte comparisons. The switch statements and strcmp/memcmp functions are also deconstructed into multiple if-else statements. With laf-intel applied, the resulting LLVM-opt has around 5.5 million static edges. N-gram does not increase the number of static edges but provides a more thorough coverage metric. Unlike AFL's default edge coverage with (src_block, dst_block) tuple, N-gram gets partial path coverage by hashing the last $N$ blocks. We choose $N = 3$ (i.e., the hash of the last three blocks) for this experiment. With stacked laf-intel and N-gram applied, the covered edges vary between 212k - 603k ($\sim$87% collision rate). While laf-intel and N-gram independently showed improvement in terms of edge/crash coverage in small benchmarks, they were not applied to such large benchmarks previously due to excessive hash collisions. To our knowledge, this is the first time the combination of laf-intel and N-gram is applied to benchmarks of this scale. Also, note that the purpose of this experiment is not to scrutinize the effectiveness of the N-gram or laf-intel themselves, rather show that BigMap can effortlessly

Table III: Code Coverage with laf-Intel and N-gram

| Benchmark (with n-gram & laf-intel) | Collision rate | | Edge coverage | | Unique crash | |
|---|---|---|---|---|---|---|
| | 64kB | 2MB | 64kB | 2MB | 64kB | 2MB |
| loop-unswitch | 70.6 | 4.9 | 214,437 | 211,697 | 276 | 325 |
| sccp | 71.1 | 5.2 | 218,473 | 226,084 | 261 | 324 |
| earlycase | 75.3 | 5.8 | 260,008 | 255,295 | 279 | 382 |
| loop-prediction | 75.8 | 6.2 | 265,740 | 270,806 | 202 | 265 |
| loop-rotate | 76.2 | 6.2 | 271,383 | 269,534 | 276 | 384 |
| irce | 77.0 | 6.0 | 281,479 | 262,675 | 226 | 245 |
| licm | 78.5 | 7.1 | 301,490 | 312,943 | 284 | 433 |
| gvn | 79.0 | 7.3 | 309,262 | 324,302 | 295 | 367 |
| simplifycfg | 79.1 | 7.4 | 311,143 | 325,526 | 285 | 412 |
| strength-reduce | 83.1 | 8.4 | 387,462 | 373,813 | 250 | 307 |
| indvars | 84.0 | 9.3 | 409,555 | 414,217 | 271 | 342 |
| loop-vectorize | 87.2 | 11.2 | 512,991 | 510,469 | 233 | 362 |
| instcombine | 86.9 | 13.1 | 588,397 | 602,669 | 295 | 434 |
| AVERAGE | 78.8% | 7.5% | 333,217 | 335,387 | 264 | 352 |

support such combinations with high map pressure. Similar to the setup of Section V-B, we took an average of three runs to reduce the variation caused by random mutation steps.

The results of the experiment are shown in Table III. Here, both the 64kB and 2MB version employs BigMap. By mitigating collision with a bigger map, the unique crashes found improved by 33% on average. However, the edge coverage remained unaffected, similar to what we observed in our previous experiment. We conclude that for large applications and/or when applying extensive coverage metrics, crash coverage can benefit from collision mitigation.

### D. Evaluating the Scalability with Parallel Fuzzing

Every fuzzing instance uses one CPU core. As a result, a system with $n$ physical cores can run $n$ concurrent fuzzing instances with virtually little performance penalty (assuming there is minimal contention for other system resources) while gaining about $n$ times more throughput. This linear scaling property is achievable by programs with a small memory footprint, where the program and the used portion of their bitmap fit within faster cache levels. In this section, we evaluate how scaling fares when a large bitmap (i.e., 2MB) is used. For this experiment, we ran 4, 8, and 12 concurrent instances in the master-secondary configuration. In this configuration, a single master instance performs the deterministic fuzzing steps before proceeding to random fuzzing. The rest are secondary instances that skip the deterministic step. The output corpus is periodically synchronized between these instances. This configuration is standard for all real-world parallel fuzzing sessions.

Figure 9(a) shows the resultant throughput. Each benchmark's throughput is normalized to the corresponding single-run version to visualize the scaling effect better. The black line is added as a theoretical reference for 1:1 scaling, where $k$ instances gain $k$ times the throughput. The bold red line is the average execution rate across all benchmarks. It is evident that both BigMap and AFL cannot maintain 1:1 scaling with large maps. The reason is, with multiple instances and large maps, the working set is much more likely to exceed the last-level cache capacity. Note that the last-level cache is shared across all the fuzzing instances. BigMap performs relatively well since it does not access the full map, therefore having a smaller effective memory footprint. AFL scales poorly. The throughput of AFL has a negative slope above four instances, meaning the total number of executions actually went down as the number of instances was increased. The per benchmark speedup attained by BigMap is given in Figure 9(b). This speedup is measured by taking the ratio of *total* test cases generated by BigMap and AFL with equal number of instances. As AFL scales poorly with the number of instances compared to BigMap (demonstrated in Figure 9(a)), it is expected for the speedup to show a super-linear behavior. On average,

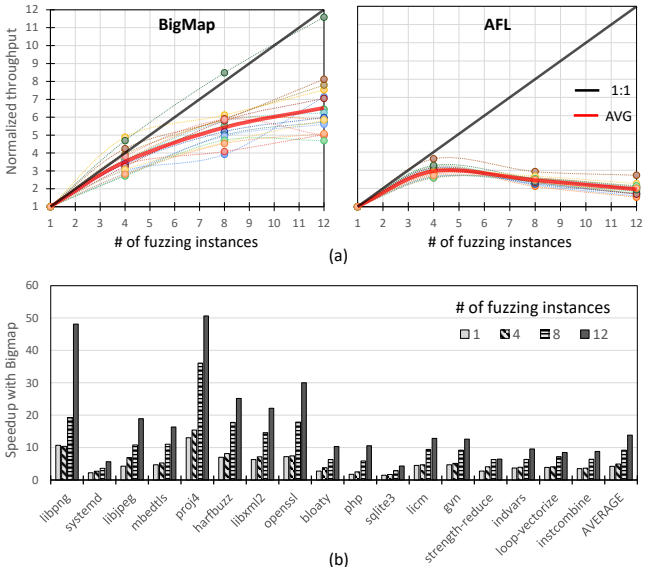


Figure 9: (a) Throughput (normalized to the single instance) vs. the number of fuzzing instances. Dotted lines represent the individual benchmarks from (b), and the solid red line is their average. The solid black line shows 1:1 scaling as a reference. (b) Speedup attained by BigMap over AFL. The coverage map is fixed to 2MB for both (a) and (b).

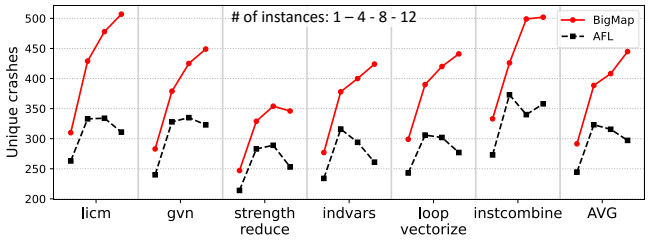


Figure 10: Unique crashes found with a varying number of fuzzing instances. The coverage map is fixed to 2MB.

BigMap achieved a speedup of 4.9x, 9.2x, and 13.8x for the 4, 8, and 12 concurrent runs, respectively.

Figure 10 depicts a similar trend in the number of unique crashes found. AFL suffers due to the drop in execution throughput. For 4, 8, and 12 instances, BigMap found 20%, 36%, and 49% more unique crashes on average. If we compare the best configurations available on our hardware (e.g., 12 instances for BigMap and 4/8 instances for AFL), BigMap shows an average speedup of 9.2x and uncovers 37% more crashes.

## VI. Related Work

Fuzzing as an evolutionary process was first introduced by Sidewinder in 2006 [24]. Since then, most successful fuzzers have followed this path [6]–[8], [16], [17], [23], [25]–[28]. A critical component in this evolutionary process is the fitness function that determines what inputs will be used as seeds for future fuzzing rounds. AFL and AFL

based fuzzers [8], [16], [23] use coarse edge hit counts as the fitness function. Any test vector that exercises a yet-unseen edge or a seen edge with a different hit count is considered an interesting input. On the other hand, libFuzzer based fuzzers [6], [7], [25] leverage compiler support such as SanitizerCoverage [18] to utilize basic block coverage as the fitness function. Angora [17] combines function calling context with edge coverage to differentiate between interesting test cases covering the same sets of edges but have unique execution paths. PerfFuzz [29] considers both execution count and code coverage. Ankou [28] queries behavioral similarity between a new test case and the current seeds in the seed pool to determine if it should be considered as interesting. All of these approaches use some form of code coverage as the fitness function. BigMap is orthogonal to these approaches and can be adopted to improve their fitness functions' accuracy by reducing collision.

In addition to seed selection, fuzzers can leverage coverage information for scheduling seeds from the seed pool. AFL schedules "favored" entries more frequently, and these entries are determined based on the edge coverage. AFLFast [16] selects seeds that cover the least frequently traveled paths. VUzzer [25] uses control-flow graphs to model the execution path and prioritizes inputs that visit deeper blocks. Cerebro [30] employs a multi-objective algorithm that takes code coverage, complexity, and execution time into account during scheduling the seed. FairFuzz [31] prioritizes seeds based on rare branch coverage. The intuition being that rare branches are more likely to hide hard to trigger bugs. NeuFuzz [32] trains a deep neural network model to differentiate between a vulnerable path from a clean path and prioritizes the vulnerable one. AFLGo [33] measures branch distance to select seeds that are closer to predetermined targets. Since these approaches use coverage feedback in their scheduling mechanism, hash collisions can obscure the seeds' priority.

The expressiveness of the coverage metric is another factor that influences the collision rate. Angora's context-sensitive coverage puts up to eight times more pressure on the bitmap [17]. Coverage metrics such as N-gram (hash of last $N$ branches), memory-access-aware branch coverage, and memory-write-aware branch coverage also exhibits higher map pressure than simple edge coverage [12]. Control-flow transformations such as laf-intel [11] or CmpCov [34] can increase the map pressure as well, necessitating collision mitigation.

Fuzzers do not need to fixate to a particular coverage metric or scheduling algorithm. An ensemble of different fuzzing mechanisms is proven to be an effective strategy [12], [35]. Ensemble fuzzers run multiple fuzzing instances with different metrics and periodically cross-pollinate the inputs. However, unlike BigMap, they do not stack the coverage metrics together, which is still subjected to increased hash collisions. Comparing BigMap with ensemble fuzzing is not covered in this work and can be an interesting avenue for future research.

CollAFL [9] is the state-of-the-art technique for mitigating hash collisions in coverage bitmap. It leverages static analysis to distribute edge IDs with a link-time compiler pass. Blocks with a single incoming edge are assigned IDs statically. For other blocks, injected instrumentation generates the IDs at runtime. It adapts to indirect edges by considering all blocks with no incoming edges as the potential branching target. One shortcoming of CollAFL is that it cannot be extended for coverage metrics other than the block or edge coverage (e.g., N-gram, Angora). In addition, it expands the bitmap to fit all the statically assigned IDs. Our experimental findings (presented in Table II) indicate that only a fraction of the static edges are visited during a fuzzing campaign, making the increase in map size a source of unnecessary runtime overhead. While both BigMap and CollAFL aim to solve the hash collision issue, they are orthogonal mitigation techniques. BigMap can be used independently of CollAFL to reduce hash collisions. It can also be used in combination with CollAFL to completely eliminate collisions while providing more efficient access to the map. Furthermore, BigMap supports any form of coverage metric as long as it is recorded in a coverage bitmap, making it applicable to a wide variety of fuzzers.

## VII. Conclusion

We investigated the common belief that enlarging bitmaps to mitigate hash collisions necessarily results in the deterioration of both throughput and quality in fuzzing campaigns. Our key observation is that the primary source of overhead stems from frequent map operations performed on the full bitmap, although only a fraction of the map is under active use. We proposed BigMap, a two-level bitmap that adds an extra level of indirection to limit the map operations on the map's active regions. Our evaluation results showed 0.98x-33.1x throughput gain over AFL as we increased the map size from 64kB to 8MB. BigMap also demonstrated better scalability with the number of concurrent fuzzing instances. Furthermore, BigMap's compatibility with most coverage metrics, along with its efficiency on large maps, enabled exploring aggressive compositions of coverage metrics and fuzzing algorithms, uncovering 33% more unique crashes. By making the use of large maps practical and open-sourcing BigMap, we hope to enable and spur further research into the design space of coverage metrics.

## Acknowledgement

This work was supported in part by CRISP, one of the six centers of JUMP, a Semiconductor Research Corporation program sponsored by DARPA. This work was also supported in part by DARPA under contract no. W911NF-18-C-0019 and FA8750-20-C-0507. The authors also wish to thank the anonymous reviewers for their time and valuable feedback.

## REFERENCES


[1] C. Cadar *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008, pp. 209–224.

[2] A. Ahmed and P. Mishra, "QUEBS: Qualifying event based search in concolic testing for validation of RTL models," in *ICCD*, 2017, pp. 185–192.

[3] Y. Lyu, A. Ahmed, and P. Mishra, "Automated activation of multiple targets in rtl models using concolic testing," in *DATE*, 2019, pp. 354–359.

[4] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on rtl models," in *DATE*, 2018, pp. 1538–1543.

[5] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," in *URL: https://github.com/google/oss-fuzz/*, 2020.

[6] ——, "Continuous fuzzing with libfuzzer and addresssanitizer," in *Cybersecurity Development*, 2016, pp. 157–157.

[7] R. Swiecki, "Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options," *URL: https://github.com/google/honggfuzz*, 2020.

[8] M. Zalewski, "American fuzzy lop, v2.52b," in *URL: https://lcamtuf.coredump.cx/afl/*, 2020.

[9] S. Gan *et al.*, "CollAFL: Path sensitive fuzzing," in *Security and Privacy*, 2018, pp. 679–696.

[10] "Fuzzbench: Fuzzer benchmarking as a service," in *URL: https://github.com/google/fuzzbench*, 2020.

[11] "laf-intel: Circumventing fuzzing roadblocks with compiler transformations." in *URL: https://clang.llvm.org/docs/SanitizerCoverage.html*, 2020.

[12] J. Wang *et al.*, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *RAID*, 2019, pp. 1–15.

[13] M. Zalewski, "Technical whitepaper for afl-fuzz," in *URL: https://github.com/google/AFL/blob/master/docs/technical_details.txt*, 2019.

[14] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *Security and Privacy*, 2019.

[15] "Ankou benchmark sources," in *URL: https://github.com/SoftSec-KAIST/Ankou-Benchmark*, 2019.

[16] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, pp. 489–506, 2017.

[17] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Security and Privacy*, 2018, pp. 711–725.

[18] "Clang sanitizer coverage," in *URL: https://clang.llvm.org/docs/SanitizerCoverage.html*, 2020.

[19] J. Naus, "Probabilities for a generalized birthday problem," *Journal of the American Statistical Association*, pp. 810–815, 1974.

[20] "opt - llvm optimizer," in *URL: https://llvm.org/docs/CommandGuide/opt.html*, 2020.

[21] B. Nagy, "Crashwalk: Bucket and triage on-disk crashes," in *URL: https://github.com/bnagy/crashwalk*, 2020.

[22] "Fuzzbench report," in *URL: https://www.fuzzbench.com/reports/2020-08-23/index.html*, 2020.

[23] M. Heuse *et al.*, "American fuzzy lop plus plus (afl++)," in *URL: https://github.com/AFLplusplus/AFLplusplus*, 2020.

[24] S. Embleton, S. Sparks, and R. Cunningham, "Sidewinder: An evolutionary guidance system for malicious input crafting," *Black Hat, August*, 2006.

[25] S. Rawat *et al.*, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, 2017, pp. 1–14.

[26] I. Yun *et al.*, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *USENIX*, 2018, pp. 745–761.

[27] D. She *et al.*, "Neuzz: Efficient fuzzing with neural program smoothing," in *Security and Privacy*, 2019, pp. 803–817.

[28] V. J. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference."

[29] C. Lemieux *et al.*, "Perffuzz: Automatically generating pathological inputs," in *SIGSOFT*, 2018, pp. 254–265.

[30] Y. Li *et al.*, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *ESEC/FSE*, 2019, pp. 533–544.

[31] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *ASE*, 2018, pp. 475–485.

[32] Y. Wang *et al.*, "Neufuzz: Efficient fuzzing with deep neural network," *IEEE Access*, pp. 36 340–36 352, 2019.

[33] M. Böhme *et al.*, "Directed greybox fuzzing," in *CCS*, 2017, pp. 2329–2344.

[34] M. Jurczyk, "Comparecoverage," in *URL: https://github.com/googleprojectzero/CompareCoverage*, 2020.

[35] C. Salls *et al.*, "Exploring abstraction functions in fuzzing," in *CNS*, 2020, pp. 1–9.