INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing

Chin-Chia Hsu*, Che-Yu Wu*, Hsu-Chun Hsiao*† and Shih-Kun Huang‡
*Department of Computer Science and Information Engineering, National Taiwan University, Taiwan
†Research Center for IT Innovation, Academia Sinica, Taiwan
‡Department of Computer Science, National Chiao Tung University, Taiwan

***Abstract*—Empowered by instrumentation, coverage-guided fuzzing monitors the program execution path taken by an input, and prioritizes inputs based on their contribution to code coverage. Although instrumenting every basic block ensures full visibility, it slows down the fuzzer and thus the speed of vulnerability discovery. This paper shows that thanks to common program structures (e.g., directed acyclic subgraphs and simple loops) and compiler optimization (e.g., knowledge of incoming edges), it is possible to accurately reconstruct coverage information by instrumenting only a small fraction of basic blocks. Specifically, we formulate the problem as a path differentiation problem on the control flow graph, and propose an efficient algorithm to select basic blocks that need to be instrumented so that different execution paths remain differentiable. We extend AFL to support such *CFG-aware instrumentation*. Our experiment results confirm that, compared with full instrumentation, our CFG-aware instrumentation only needs to instrument about 20% of basic blocks while offering 1.04–1.78x speedup during fuzzing. Finally, we highlight several technical challenges and promising research directions to further improve instrumentation for fuzzing.**

I. INTRODUCTION

Coverage-guided fuzzing has been shown to be an effective technique for automated vulnerability discovery [1]. At a high level, a fuzz testing tool, or *fuzzer*, feeds a program with random inputs so as to find those that crash the program or trigger exceptions. To increase the chance of finding new crashes, a coverage-guided fuzzer favors inputs that contribute new code coverage (such inputs are called *seeds*) and generates new inputs by mutating the seeds.

Being able to measure code coverage is thus essential to the success of coverage-guided fuzzing, and it is often achieved through instrumentation. By inserting instrumentation code to a program, the fuzzer can track the execution flow exercised by an input and determine whether the input covers a new part of the program (which has not been executed before). However, instrumentation is expensive. Although instrumenting every basic block (referred to as InstrAll) ensures full visibility, it slows down the fuzzer and thus the speed of vulnerability discovery due to the extra time needed to execute the instrumentation code.

It is challenging to reduce instrumentation cost without sacrificing coverage information. AFL can optionally instrument a random subset of basic blocks, but its impact on coverage information requires careful evaluation. Moreover, most existing techniques to reduce instrumentation overhead [2], [8], [6] either rely on simple heuristics or become inefficient when being directly applied to coverage-guided fuzzing. Hence, the core research questions we would like to investigate are: What techniques can be applied to reduce the cost of instrumentation? What is their impact on the effectiveness of coverage-guided fuzzing? As an initial probe to these questions, this work first formulates the problem as the **path differentiation problem** on the control flow graph (CFG). This is, given the control flow graph of a program, we want to identify a subset of basic blocks for instrumentation such that different execution paths remain differentiable.

As a proof of concept, we design and implement efficient **CFG-aware instrumentation** algorithms that take advantage of common program structures (e.g., directed acyclic subgraphs and simple loops) and compiler optimization (e.g., knowledge of the incoming edges). The directed acyclic subgraph, which is a common pattern in CFG, allows us to reduce the number of instrumented basic blocks but still be able to distinguish each execution path. As for simple loops that do not affect the subsequent control flow of the program, we can further reduce the instrumentation cost by only retrieving the information of whether the current execution will step into this loop. Finally, knowing the incoming edges to a basic block provides additional information about multiple previous vertices by instrumenting one vertex only.

Our experiment results confirm that, compared with InstrAll, our algorithm only needs to instrument about 20% of basic blocks without sacrificing the accuracy of the coverage information. In addition, it reduces runtime overhead and provide 1.04–1.78x speedup during fuzzing. Finally, we explore promising future directions to further enhance INSTRIM, including dynamic marking algorithms and cache-aware instrumentation.

Our code is available at https://github.com/csienslab/instrim.

Workshop on Binary Analysis Research (BAR) 2018
18 February 2018, San Diego, CA, USA
ISBN 1-891562-50-9
https://dx.doi.org/10.14722/bar.2018.23014
www.ndss-symposium.org

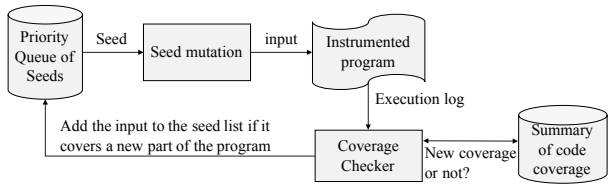


Fig. 1: fuzz testing process.

II. Background and Related Work

Coverage-guided Fuzz Testing. Fuzzing is an automated technique to discover program vulnerabilities. Given a program, a fuzz testing tool (or fuzzer) feeds a large number of inputs into the program so as to find inputs that crash the program. The input selection can be totally random [3], mutational [7], or generational [4].

In a *coverage-guided fuzzer*, new inputs are created by mutating interesting inputs (called *seeds*) in previous fuzz runs. Beginning with an initial set of seeds, the fuzzer mutates the seeds to generate new inputs, and feeds the new inputs to the program. An input is considered interesting if its execution flow explores a new part of the program (e.g., a new basic block). These interesting inputs will then be added to the seed queue. On the other hand, if an input does not contribute new coverage, it becomes less favored in the next fuzz run. This feedback strategy has shown to be effective in finding real-world vulnerabilities [1]. Figure 1 illustrates a typical workflow of coverage-guided fuzz testing.

As a coverage-guided fuzzer needs to be informed of code coverage information in seed selection, one key component of it is accurate and efficient detection of new code coverage. Code coverage is an important metric indicating the percentage of a program being executed in software testing. Code coverage can be quantified by basic blocks, edges, paths, etc. When an input executes a new basic block, edge, or path, the corresponding code coverage will increase. AFL-like fuzzers also consider hit-count coverage, which takes into account the execution times. To determine whether an execution path of a given input contributes new coverage, the fuzzer needs to perform two tasks: (1) tracking the execution path of the input, and (2) checking the current path against the past ones. The former can be tracked by adding instrumentation code to basic blocks in the program, and the latter one requires keeping a compact data structure for efficient checking.

In sum, the support of instrumentation introduces two types of latency: one due to the execution of additional instrumented code, the other by checking and updating the data structure maintaining code coverage information. To avoid outweighing fuzzing's benefit, it is important to ensure low runtime overhead of instrumentation. While our work focuses on reducing the first type of latency, improvement in the first type can also help enhance the performance of the second, because by reducing the amount of information needed to be tracked, we also lessen the overheads associated with the data structure.

Related Work. AFL supports an alternative method (referred to as InstrRand) that instruments only a random subset of basic blocks to alleviate the bitmap saturation problem. However, because such a random strategy may omit important coverage information of non-instrumented blocks, the fuzzer may falsely discard useful seeds that execute new non-instrumented blocks with a high probability, as shown in our evaluation in Table I.

Tikir and Hollingsworth [8] proposed a heuristic to selectively instrument a subset of basic blocks such that the exact execution path can still be recovered. However, their simple heuristic often ends up selecting almost every basic block in practice, as shown in our evaluation in Figure 4.

Ohmann et al. [6] consider binarized coverage and modeled the problem using integer linear programming and proved it is NP-hard. They proposed several approximation algorithms, which still need to mark about a half of the total vertices. Also, their static program analysis requires knowing all the execution paths in advance.

Path profiling aims to count the execution times of each program path. The seminal work of Ball and Larus [2] describes an algorithm to select which edges to instrument and assign a value to each of the instrumented edges such that each distinct path has a unique value sum. However, given a sum, they need to spend extra time to reconstruct the corresponding execution path in order to compute edge coverage, which would be a severe performance obstacle when being applied to fuzzing. In contrast, our approach can be applied to efficiently compute code coverage without explicitly reconstructing the path.

III. Problem Definition

A. Control Flow Graph and Vertex Marking

The control-flow graph of a program is a directed graph that abstracts the possible execution paths of the program. In a control-flow graph, each vertex represents a basic block and each directed edge represents a transition between two basic blocks. An execution path is represented by a sequence of vertices on the graph.

Given a control-flow graph (CFG) of a program, the entry vertex (the entry of the program) has an in-degree of 0 and the exit vertex (the exit of the program) has an out-degree of 0. For ease of description, if there are multiple such vertices, we can link all of them to an extra entry/exit vertex.

A vertex is *marked* if and only if the corresponding basic block is instrumented. The problem of selecting which basic blocks to instrument can be modeled as a vertex marking problem on graphs.

B. Path Differentiation Problem

Based on the above graph model, we define the **path differentiation problem** as follows. Our goal is to mark some vertices on the graph such that, for each pair of paths, we are able to distinguish them by only taking those marked vertices into consideration. That is, a path is now represented by the sequence of marked vertices it traverses instead of all of the traversed vertices.

To explore trade-offs between distinguishability and performance, we also consider the **approximate path differentiation problem**, in which we can tolerate some different paths to be considered identical.

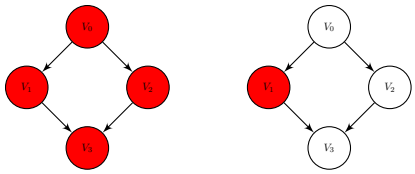


Fig. 2: Two possible markings of an example CFG.

Note that even when two different execution paths lead to the same basic block, they may be subjected to different sets of constraints, and thus may explore different basic blocks in subsequent execution. Hence, this work considers a general definition that also supports path-based information, in which an input is interesting if it produces a new execution path (or a path that has been executed more than a threshold of times) on the control-flow graph.

Figure 2 illustrates two different markings of a CFG. The red nodes are marked. On the fully marked graph, the execution path going through V_0, V_1, V_3 is represented by $[V_0, V_1, V_3]$, and the execution path going through V_0, V_2, V_3 is represented by $[V_0, V_2, V_3]$. On the partially marked graph on the right hand side, only V_1 is marked. The execution path through V_0, V_1, V_3 is represented by $[V_1]$, and the execution path through V_0, V_2, V_3 is represented by [] (an empty list). In this case, we can still distinguish these two execution paths by using only about $\frac{1}{6}$ time and memory.

IV. Algorithm

This section describes InsTrim and InsTrim-Approx, which solve the exact and approximate path differentiation problems, respectively.

InsTrim takes advantage of the following properties:

- **Knowledge of the incoming edges**: By using ϕ nodes, we can know not only the visited basic blocks but also the incoming edge of a visited block. This is a powerful property: suppose a vertex v has n incoming edges from vertices $v_1, v_2, \cdots v_n$, instead of instrumenting $n+1$ blocks, we only need to instrument one (vertex v) block to recover the executed edge to v.
- **Common program structures**: Many control flow graphs are *almost* a directed acyclic graph (DAG) as they can be converted into a DAG by removing few back edges, and a DAG is much easier to process than a general graph. Another common program structure is simple loops, whose execution times does not affect subsequent CFG. We can further reduce instrumentation overhead for simple loops by avoid tracking the number of its iterations.

Listing 1 describes the pseudocode of InsTrim, which consists of two functions:

1) **Marking nodes on a subgraph** (§IV-A): Given a subgraph of the CFG, this marking function traverses it in a topological order and determines whether to mark a vertex in a recursive definition. This function is the heart of our algorithms as it judiciously applies the knowledge of the incoming edges and common program structures to reduce marked vertices.
2) **Dividing CFG into subgraphs** (§IV-B): To further reduce the time complexity of the marking function, this function divides a given CFG into smaller subgraphs based on the dominators of the exit vertex.

The computational complexity is $O((|V|+|E|) \times |V| \lg |V|)$.[1]

In addition, since programs often contain simple loops (e.g., array initialization) whose execution times do not affect the subsequent control flow of a program, we also propose an approximate algorithm called InsTrim-Approx that can reduce the instrumentation cost inside simple loops (§IV-C).

```
1  def mark(G, t, s):
2    G = G.subgraph(s, t)
3    marked = {v for (u, v) in G.E.backedges}
4    G.E erases out G.E.backedges
5    # G becomes directed acyclic graph
6    T = topological_order(G)
7    for x in T:
8      need_marked = False
9      P(x) = {}
10     for u in G.E.from(x):
11       P(x) = P(x) union P(u)
12       for v in G.E.from(x):
13         if u == v: continue
14         if size(intersection(P(u), P(v))) > 0:
15           need_marked = True
16     if need_marked or x in marked:
17       marked += x
18       P(x) = {x}
19   return marked
20 def main(G): # G := control flow graph
21   marked = {}
22   D = dominator_tree(G)
23   # D.idom(v) := immediate dominator of v
24   cur_state = G.final_state
25   while cur_state != G.init_state:
26     marked += mark(G, cur_state, D.idom(cur_state))
27     cur_state = D.idom(cur_state)
28   return marked
```

Listing 1: Pseudocode of our exact algorithm InsTrim

A. Marking Nodes on a Subgraph (Line 1-19)

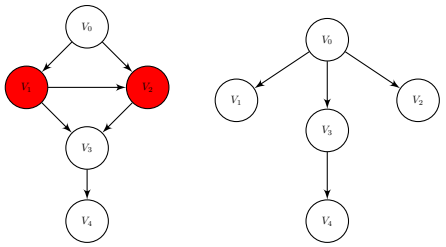
We first define a useful notation. Given a vertex v on the CFG, we define the *previous marked vertex set* of v, $\mathbb{P}(v)$, to be the set of the last visited marked vertices among all possible paths ending at v on the CFG. For example, considering Figure 3a, $\mathbb{P}(V_0) = \phi$, $\mathbb{P}(V_1) = \{V_1\}$, $\mathbb{P}(V_2) = \{V_2\}$, $\mathbb{P}(V_3) = \{V_1, V_2\}$, and $\mathbb{P}(V_4) = \{V_1, V_2\}$.

Since a feasible $\mathbb{P}(v)$ for each vertex v on the subgraph corresponds to one vertex marking assignment, the problem becomes how to compute a feasible $\mathbb{P}(v)$. InsTrim defines $\mathbb{P}(v)$ recursively on a directed acyclic graph (DAG). We convert the CFG (which might contain cycles) to a DAG and then define the base case and recursive case for computing $\mathbb{P}(v)$ in a dynamic-programming-like approach as follows.

First, to convert the CFG to a DAG, we erase all back edges (which can be identified by performing a depth-first search). To achieve this, for each back edge, we directly mark the vertex to which the back edge heads. Since our instrumentation can know the incoming edge of this vertex, we will not lose any coverage information after removing these back edges.

Then, we consider vertices in a topological order (starting from the vertex without any incoming edge) on the DAG and

[1]Constructing the dominator tree costs $O(|V|+|E|)$ and marking vertices costs $O((|V|+|E|) \times |V| \lg |V|)$.



(a) CFG (b) dominator tree of (a)

Fig. 3: Example of a CFG and its dominator tree

model this problem into a dynamic programming-like problem. For those vertices without any incoming edge (the base case of the DP problem), the condition holds because every path ending at it is distinguishable.

For the recursive case, for each node x, we examine its incoming vertices in a pairwise manner. If there exists a vertex k both in $\mathbb{P}(u)$ and $\mathbb{P}(v)$, where u and v are two vertices being able to reach x, then there are at least two paths: one from k to u then to x, and the other from k to v then to x. To distinguish these two paths, marking node x is enough because by marking x, we know the incoming vertex to x. Formally, if x is marked, $\mathbb{P}(x) = \{x\}$. Otherwise,

$$\mathbb{P}(x) = \cup_{v|\text{edge }(v,\text{ x})\ \in\ \text{graph}}\mathbb{P}(v)$$

B. Dividing CFG into Subgraphs (Line 20-29)

To further improve the efficiency of the algorithm, INSTRIM divides the CFG into smaller subgraphs. It is important to note that INSTRIM will compute the same result (i.e., the same set of marked vertices) even without dividing the CFG.

We extract the domination relationship by constructing a *dominator tree* [5] of the CFG. A dominator tree of a directed graph is a tree in which node A is the parent of node B if and only if A immediately dominates B. Building a dominator tree allows us to quickly check the domination relationship between vertices and determine whether a vertex should be marked. Since $D.idom(t)$ (the immediate dominator of t on the CFG and also the parent of t on D) must be executed before t on every possible path reaching t, we can "cut off" the subgraph from $D.idom(t)$ to t, and mark the two subgraphs separately. This cut process can be done repeatedly until the immediate dominator is the root of D, thereby reducing the problem into smaller sub-problems for efficiency.

Consider the CFG in Figure 3a and its dominator tree in Figure 3b. The exit vertex (V_4) and all its ancestors (V_0, V_3) will be excluded from the marked vertices (V_1, V_2). Then, the subgraph cut by V_3 and V_4 will contain vertices V_3, V_4 and edges between them. The subgraph cut by V_0 and V_3 will contain vertices V_0, V_1, V_2, V_3 and edges between them.

C. INSTRIM-APPROX: Improvement for Loops

In INSTRIM, we mark all vertices pointed by back edges. In the presence of a loop, INSTRIM marks the vertex representing the entry of this loop. However, when executing this loop, the instrumented gadget in the entry will be executed repeatedly, which may waste lots of resources especially when the loop is simple—that is, it contains only one basic block (a self-loop)

To mitigate this problem, we modify INSTRIM and propose INSTRIM-APPROX that adds a virtual marked vertex above the entry of the loop. By doing this, we can still distinguish whether the loop has been executed or not, but we trade off the information about the execution times of the loop, as INSTRIM-APPROX will only append the marked vertex once. Our preliminary evaluation in Section VI shows that INSTRIM-APPROX does not lose much accuracy (in terms of falsely included and excluded seeds).

V. IMPLEMENTATION

This section briefly describes our implementation using the LLVM instrumentation.

Marking and labeling basic blocks. We split a function into several basic blocks in LLVM. Each basic block is assigned a unique ID generated by a pseudo random number generator. After applying INSTRIM to each function, we obtain a list of basic blocks to be marked. Our implementation assigns two kinds of labels—node label and edge label—to each marked basic block. The node label of a marked basic block is equal to its ID (which was assigned at the very beginning). For each incoming edge of a marked basic block, we also assign a random edge label. These labels are required to construct a reduced graph in our code coverage recording method described next.

In INSTRIM-APPROX, we also need to insert a virtual basic block before each loop entry, but identifying an edge entering a loop is non-trivial. In our implementation, for each basic block representing a loop entry, we first try to find an incoming edge that 1) originates from outside the loop, 2) points to the basic block, and 3) dominating this basic block (i.e., the edge must be visited before reaching the basic block). Our marking depends on whether such an edge is found. If such an edge is found, we split this edge, insert a new basic block, and mark this new basic block. If no such edge is found, we directly mark the basic block representing the loop entry.

Recording code coverage. Similar to AFL, when executing an instrumented program, we store the node label of the last marked basic block in the Thread Local Storage (TLS) and record the path segment pair `(LastNodeLabel, CurrentIncomingEdgeLabel)` on the coverage map. By computing code coverage over the path segments (rather than original CFG), we avoid the overhead of reconstructing the original execution path. Since there may be more path segments than the edges of original CFG and every unseen path segment between marked basic blocks will contribute new coverage, we may report more seeds having new coverage than the AFL method, but the difference is small as shown in our experiments.

VI. EVALUATION

We compare INSTRIM and INSTRIM-APPROX with *InstrAll* (AFL default, instrumenting every basic block), *InstrRand*, and the *TH algorithm* [8]. To ensure fair comparison, InstrRand instruments $X\%$ of randomly chosen basic blocks, where X is set to be the same percentage as in INSTRIM. We report

the detailed configurations for measuring execution time and code coverage.

Execution time measurement. To evaluate the runtime overhead of different algorithms, we measure their execution time for replaying the same set of test seeds. The reason why we replay seeds instead of running the fuzzer directly is because different algorithms may favor different seeds with varying execution time, which prevents meaningful comparisons.

We directly compared INSTRIM and INSTRIM-APPROX with AFL instrumentation. Both are integrated with the AFL bloom filter by increasing the counter at `(LastNodeLabel ⊕ CurrentIncomingEdgeLabel)`. The InstrRand and TH algorithms are left out in this experiment because the percentage of marked basic blocks is the main factor of overhead, and both of them perform worse than our algorithms on this factor.

Code coverage measurement. In this set of experiments, to focus on evaluating instrumentation, we maintain our own data structure to measure code coverage, avoiding inaccuracy introduced by inconsistent or approximate code coverage measurement. Specifically, we replace the AFL bloom filter with a set data structure to prevent collision and consider binarized edge coverage [6] (i.e., ignoring hit counts) to ensure comparable results. In addition, the new-coverage detection method uses a slightly different way to label vertices in different algorithms. In InstrAll and InstrRand, to behave like AFL without collision, the reported label is the concatenation of the previous and current basic block unique IDs. In INSTRIM and INSTRIM-APPROX, we use the recording method described in the implementation section.

Target programs. We selected five small-sized programs (`libfreetype`, `libxml2`, `libcapstone`, `lame --silent --preset standard`, and `objdump -dg`) and three medium-sized ones containing 10^5–10^6 basic blocks (`libpypy`, `coreclr`, and `libwireshark`). These programs are selected as they are commonly used in prior studies and can be successfully executed on our environment.

Test seeds collection. We fuzzed each program over 24 hours using AFL and collected the generated seeds. Due to high computation overhead of our code coverage measurement, we replayed these seeds to filter out those taking too much time (>10s) to run.

Evaluation metrics. We consider three evaluation metrics:

- Percentage of marked vertices: the ratio between number of marked vertices and total vertices.
- Difference of favored seeds: the number of favored seeds that differ from those favored by InstrAll. An algorithm favors a seed if considering it provides new coverage.
- Execution time of replaying seeds: the time taken by running all test seeds.

A. Experiment Results

Figure 4 shows the number of basic blocks marked by INSTRIM and the TH algorithm. The TH algorithm performs poorly because the number of marked vertices is more than a half of the total vertices for all programs. By contrast,

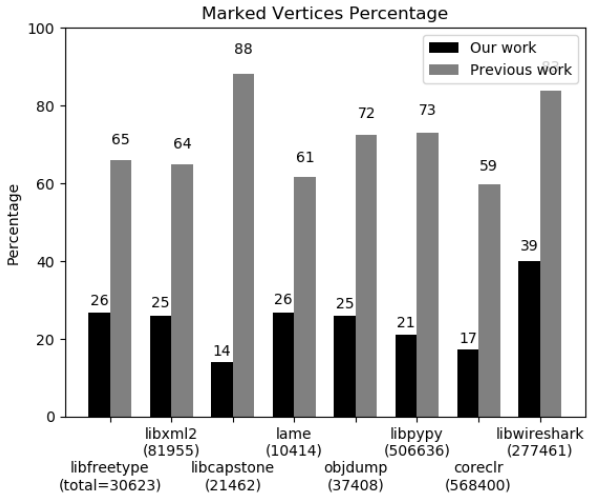


Fig. 4: The percentage of marked vertices.

INSTRIM marks about 20% of total vertices (except for `libwireshark`, but TH algorithm still marks twice as many as INSTRIM), which is significantly lower. Note that INSTRIM and INSTRIM-APPROX mark exactly the same vertices because they only differ in instrumentation implementation.

Table I compares INSTRIM and INSTRIM-APPROX with InstrRand regarding the quality of their favored seeds. We use the seeds selected by InstrAll as a baseline, since InstrAll is similar to the methods used by AFL-like fuzzers. For each algorithm, we calculate the percentage of seeds only favored by it but not by InstrAll (+) and the percentage of seeds only favored by InstrAll (−). We say an algorithm is more accurate if its total difference is smaller. N is the number of seeds favored by InstrAll. For InstrRand, the $X\%$ of each program is determined by the percentage of marked basic blocks in INSTRIM. That is, if INSTRIM selects $X\%$ of total vertices, InstrRand will choose $X\%$ of vertices randomly. The results show that our algorithms outperform the random selection InstrRand. In these cases, INSTRIM reduces more than 20% of difference. Even though INSTRIM-APPROX sacrifices some accuracy, it still performs better than InstrRand.

To measure execution time, we combined our algorithms with the AFL bloom filter and replayed all test seeds five times. Table II shows the average execution time per seed and the speedup comparing to AFL (with 100% instrumentation ratio). The speedup is more significant on medium-sized programs than small-sized ones. Additional measurements are required to determine the root cause of the difference. One possibility is that as the execution time consists of many parts such as executing code, I/O latency, and system calls, in medium-sized programs, the part of executing code is bigger and thus has more instrumentation overhead that can be reduced by our algorithms. Another possibility is that the medium-sized programs in our experiment were shared libraries, and thus AFL needs to perform an extra function call to retrieve the TLS address, which incurs instrumentation overhead.

TABLE I: The percentage of seed difference.

	libfreetype		libxml2		libcapstone		lame		objdump		libpypy		coreclr		libwireshark	
N	775		1197		561		83		882		1107		461		10041	
$X\%$	27%		26%		14%		27%		26%		21%		17%		40%	
	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+
InstrRand	35%	35%	33%	39%	64%	14%	30%	60%	45%	19%	17%	35%	20%	8%	23%	5%
INSTRIM	2%	18%	1%	15%	0%	7%	1%	20%	1%	18%	3%	20%	6%	5%	4%	4%
INSTRIM-APPROX	4%	32%	4%	25%	0%	7%	1%	66%	1%	19%	4%	25%	6%	7%	6%	9%

TABLE II: The execution time of different methods.

	AFL	INSTRIM		INSTRIM-APPROX	
	time (μs)	time	speedup	time	speedup
libfreetype	114	95	1.20	92	1.23
libxml2	48	42	1.14	41	1.17
libcapstone	26	25	1.06	25	1.06
lame	832	801	1.04	756	1.10
objdump	451	432	1.04	417	1.08
libpypy	5682	3444	1.65	3184	1.78
coreclr	6531	4760	1.37	4602	1.41
libwireshark	513	402	1.27	391	1.31

VII. DISCUSSION AND FUTURE DIRECTIONS

There is still much room for improvement in the design space of lightweight instrumentation for coverage-guided fuzzing. We discuss tradeoffs between speed and accuracy, and highlight promising research directions.

Dynamic marking algorithms. So far we only consider static information and static instrumentation. By taking into account the run-time execution information, we can estimate the execution frequency of each basic block, and design a weighted version of INSTRIM that aims to minimize the delay introduced by instrumentation. Another possibility is to dynamically adjust the marked vertices as the fuzzer collects more information during exploration.

Cache-aware instrumentation. Our main future work is to study cache-aware instrumentation, as frequent cache-miss will significantly slowdown fuzzing. One aspect of cache-aware instrumentation is optimizing bitmap access pattern via better labeling algorithms, such that a bigger bitmap can be used to alleviate the saturation problem.

In general, a (block or edge) labeling algorithm for coverage-guided fuzzing should satisfy three requirements: fast to compute, uniqueness (to avoid collision), high locality (to avoid cache miss). For example, LLVM SanitizerCoverage (which is used by libfuzzer and LLVM Trace PC mode) sequentially assigns node labels in a simple traversal order and keeps edge hit-counts in a large array indexed by the labels. Such a simple assignment algorithm already exhibits good locality during program execution, such that array access contributes little overhead despite the relatively huge array size. This also avoids inaccuracy introduced by bitmap collision. To probe the influence of cache-aware label assignment on instrumentation speed, we allocate a big bitmap with 2M slots and directly assign an exclusive slot to each CFG edge. We write an LLVM pass to assign a sequential number for each edge in LLVM's natural traversing order. Our result shows that such cache-aware assignment using 2M slots is as fast as random assignment using 65536 slots.

We are currently studying how to enhance INSTRIM with cache-aware label assignment. Specifically, because INSTRIM represents a path segment by the tuple of the previous marked basic block and the incoming edge to the current marked basic block, the main technical challenge is how to optimally assign both block and edge IDs so as to ensure fast computation, space efficiency, and cache awareness. In fact, integrating INSTRIM and cache-aware assignment is non-trivial, because partial instrumentation creates a large number of path segments that need to be uniquely labeled. To achieve this, we are exploring label assignment algorithms that allow labels to be densely packed within a minimal range, and also reducing the number of path segments, for example, by marking redundant vertices in our marking algorithms.

Path-segment coverage vs. edge coverage. To compute coverage information without needing to reconstruct execution paths, INSTRIM approximates edge coverage using path-segment coverage. Our preliminary experiment suggests these two coverage metrics lead to similar outcomes in coverage-guided fuzzing. An open research problem is whether it is possible to obtain exact edge coverage information without reconstructing execution paths under partial instrumentation.

In addition, though our approximate algorithm can reduce the unnecessary overhead caused by simple loops, it remains unclear how to reasonably approximate the coverage information of the inner loops inside nested loops.

VIII. CONCLUSIONS

This paper explores lightweight instrumentation for coverage-guided fuzzing. We formulate the problem as the path differentiation problem on the control-flow graph, establishing an algorithmic foundation on which we can apply graph theory techniques. Our evaluation results show that the proposed algorithms are promising in accelerating fuzzing by marking fewer blocks and keeping more useful seeds. While prior work in fuzzing often focuses on seed selection and scheduling, we hope this paper can encourage discussion and innovation in instrumentation techniques tailored for coverage-guided fuzzing, thereby helping unleash the full potential of fuzzing-based automated vulnerability discovery.

ACKNOWLEDGMENT

This work was supported in part by Taiwan Information Security Center (TWISC), Academia Sinica, and the Ministry of Science and Technology of Taiwan under grants MOST 106-3114-E-001-001 and 107-2636-E-002-005-, and by Institute for Information Industry under grant 106-EC-17-D-11-1502.

References

[1] "american fuzzy lop," http://lcamtuf.coredump.cx/afl, 2015.

[2] T. Ball and J. R. Larus, "Efficient Path Profiling," in *IEEE/ACM International Symposium on Microarchitecture*, 1996.

[3] W. Dorman, "CERT Basic Fuzzing Framework," https://www.cert.org/vulnerability-analysis/tools/bff.cfm, 2010.

[4] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *SIGPLAN Not.*, vol. 43, no. 6, pp. 206–215, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375607

[5] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," pp. 121–141, 1979.

[6] P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit, "Optimizing Customized Program Coverage," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016.

[7] A. Rebert, S. K. Cha, T. Avgerinos, J. M. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing Seed Selection For Fuzzing," in *USENIX Security Symposium*, 2014.

[8] M. M. Tikir and J. K. Hollingsworth, "Efficient Instrumentation for Code Coverage Testing," *SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 86–96, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/566171.566186

IX. Appendix

A. Code coverage detection in AFL

In order to quickly determine whether an input contributes new coverage, the AFL fuzzer [1] maintains a compact bitmap (a counting Bloom filter with 65,536 single-byte slots by default) that can fit in the L1 cache. AFL uses such a small bitmap to avoid cache miss, as frequent cache-miss would slowdown fuzzing significantly. However, due to the small-sized bitmap, collisions may occur frequently, therefore falsely excluding interesting inputs. For example, a `libpypy` execution trace on average has more than 30000 edges, and with a half saturated bitmap, AFL will suffer from a 0.5 false positive rate when checking the existence of one edge. In addition, such big programs (libpypy, dotnet core) often have around 100K-1M CFG edges.

Note that AFL adopts edge-based code coverage rather than path-based, as the number of possible paths is much more than the number of possible edges and thus the path-based definition could worsen the bitmap saturation problem. For each edge on an execution path, AFL hashes it to a slot on the bitmap and increments the slot. An input is considered interesting if it triggers change in the most significant bit of a slot. Hence, reducing the number of instrumented basic blocks (like InsTrim does) can also alleviate the bitmap saturation problem because a path can be represented by fewer blocks, thereby reducing collisions that quickly overflow the counters.

To mitigate the bitmap saturation problem, AFL provides an instrumentation option (`AFL_INST_RATIO`) of `afl-clang-fast` to control the ratio of instrumented basic blocks. An interesting future direction is to examine the impact of bitmap saturation on AFL, especially on its ability to find new paths. Since a new path may consist of several new edges, the probability of missing one new path could be much lower than it of missing one new edge.

B. Thread Local Storage

AFL assigns each basic block on the CFG a random block label in $[0, MAP\_SIZE)$ at compile time, where the default MAP_SIZE is 65536, and represents an edge from block A to B using an edge label $(A.label >> 1) \oplus B.label$.

While executing a program, AFL stores the label of the previous basic block in Thread Local Storage (TLS) instead of stack. Because TLS is a global variable, this allows AFL to differentiate indirect control-flow transfers across function borders. However, accessing TLS is slower than accessing stack variables: it is about 2x slower than accessing stack variables in our experiment. It would be interesting to further explore the impact of using stack or TLS on AFL's performance.

The AFL experimental Trace PC Mode avoids the need of keeping the previous block information by removing critical edges (whose source and destination blocks have multiple outgoing and incoming edges, respectively) on the CFG. An alternative to removing critical edges is directly labeling edges instead of blocks. Although this approach can be effortlessly implemented under 100% instrumentation, it becomes trickier under partial instrumentation (such as InsTrim) because it is unclear where edge labels should be restored and processed on the partially instrumented graph.