Towards Systematic and Dynamic Task Allocation for Collaborative Parallel Fuzzing

Van-Thuan Pham
University of Melbourne
thuan.pham@unimelb.edu.au

Manh-Dung Nguyen
Montimage
manhdung.nguyen@montimage.com

Quang-Trung Ta
National University of Singapore
taqt@comp.nus.edu.sg

Toby Murray
University of Melbourne
toby.murray@unimelb.edu.au

Benjamin I.P. Rubinstein
University of Melbourne
benjamin.rubinstein@unimelb.edu.au

***Abstract*—Parallel coverage-guided greybox fuzzing is the most common setup for vulnerability discovery at scale. However, so far it has received little attention from the research community compared to single-mode fuzzing, leaving open several problems particularly in its task allocation strategies. Current approaches focus on managing micro tasks, at the seed input level, and their task division algorithms are either ad-hoc or static. In this paper, we leverage research on graph partitioning and search algorithms to propose a systematic and dynamic task allocation solution that works at the macro-task level. First, we design an attributed graph to capture both the program structures (e.g., program call graph) and fuzzing information (e.g., branch hit counts, bug discovery probability). Second, our graph partitioning algorithm divides the global program search space into sub-search-spaces. Finally our search algorithm prioritizes these sub-search-spaces (i.e., tasks) and explores them to maximize code coverage and number of bugs found. The results are collected to update the graph and guide further iterations of partitioning and exploration. We implemented a prototype tool called AFLTeam. In our preliminary experiments on well-tested benchmarks, AFLTeam achieved higher code coverage (up to 16.4% branch coverage improvement) compared to the default parallel mode of AFL and discovered 2 zero-day bugs in FFmpeg and JasPer toolkits.**

***Index Terms*—Vulnerability Discovery, Parallel Fuzzing**

I. Introduction

Coverage-based Greybox Fuzzing (CGF) is a well-established automated testing technique. Given a set of sample inputs (a.k.a seed corpus), CGF slightly mutates them, feeds newly generated inputs to an instrumented version of the program under test, and leverages the code coverage feedback (e.g., branch coverage) from the instrumented program to evolutionarily enlarge the seed corpus and discover bugs. This technique is implemented in popular fuzzers like American Fuzzy Lop (AFL) [1] and libFuzzer [2], which are effective in bug finding [3]–[6]. Different approaches have been proposed to improve the effectiveness and efficiency of CGF [7]–[18] and expand its reach to more challenging target programs like network protocols [19] and smart contracts [20]. However, existing works mainly focus on single-mode fuzzing and rely on off-the-shelf and ineffective parallel fuzzing setups (e.g., AFL's default parallel mode) to run the fuzzers at scale on multiple cores in a single computer or over interconnected computers [21]. Research on improving parallel fuzzing techniques surprisingly enjoys much less attention.

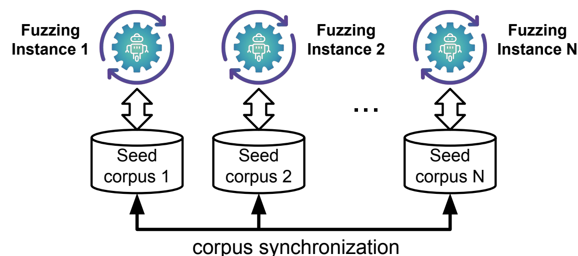

Figure 1. AFL's default collaborative parallel fuzzing setup.

Based on the interaction between fuzzing instances, we categorize parallel fuzzing setups into non-collaborative and collaborative fuzzing. In a non-collaborative setup, fuzzing instances explore the same target program independently so they may end up trying to cover the same branch(es). This setup might lead to resource and time inefficiency. For instance, we ran five instances of AFL in this non-collaborative setup in 4 hours to fuzz the LibPNG [22] library and our analysis shows that the overlapping ratio of the branch coverage is approximately 95%. It would be better for the fuzzing instances to share results with each other. This is the idea behind the collaborative setup implemented in AFL and libFuzzer, as depicted in Figure 1. More specifically, AFL supports collaborative parallel fuzzing via a shared corpus directory so that any new interesting inputs (a.k.a seeds) found by one fuzzing instance will be available to the other instances. However, this setup also has two major problems.

Missing fuzzing information. Valuable dynamic information (e.g., path frequency, branch hit counts) that supports single-mode optimizations [7], [12] is not simultaneously synchronized among fuzzing instances in this parallel mode. Instead, such information is periodically collected and used by each instance. This leads to the underperformance of optimized single-mode fuzzers like AFLFast and FairFuzz, compared to AFL, in parallel fuzzing [23].

Task conflicts. AFL has no explicit task allocation strategies that divide tasks and assign them to the fuzzing instances.

https://t.me/learningnets

It relies on the stochastic nature of fuzzing to do implicit task scheduling. However, given the same (synchronized) seed corpus and the same algorithm (e.g., seed selection and mutations), different fuzzing instances would still select the same seeds and produce similar test cases.

Several works have recently been completed to address the aforementioned problems (PAFL [23], P-Fuzz [24], Uni-Fuzz [25]). These works share a similar idea of adding new hierarchical data structures [23] and/or building a centralized database [24], [25] to synchronize dynamic fuzzing information and construct a global seed corpus. The benefit of this solution is twofold. First, each fuzzing instance has a local view as well as a global view of the fuzzing information to support its optimized algorithms [7], [12]. Second, seeds are selected from the global corpus before being dispatched to fuzzing instances statically in batches [23] or dynamically upon requests [24], [25]. It could ensure one seed is fuzzed by only one instance (at a time) and hence reduce task conflicts.

However, by considering a single seed input as a task, these approaches only work at the *micro-task* level which is not ideal for task management. Moreover, their algorithms to divide tasks might be ineffective since they do not consider structural information (e.g., module dependency graph, call graph, control flow graph) of the input program. We argue that collaborative parallel fuzzing should also have a systematic task allocation approach that works at the *macro* level and supports fuzzing instances to control the micro tasks. To the best of our knowledge, there are no such available solutions.

In this paper, we leverage research on graph partitioning and search algorithms to propose a systematic and dynamic task allocation solution. First, we design an attributed graph to capture both the program structures (e.g., program call graph) and fuzzing information (e.g., branch hit counts, bug discovery probability). Second, our partitioning algorithm divides the global program search space into sub-search-spaces. Finally our search algorithm prioritizes these sub-search-spaces (i.e., tasks) and explores them to maximize code coverage and number of bugs found. The results are collected to update the graph and guide further iterations of partitioning and exploration. Specifically, our paper makes the following contributions:

- ***Systematic and Dynamic Task Allocation.*** We propose a novel graph partitioning-based task allocation approach. The approach works in a self-reinforcing manner to evolve and reach optimal state.
- ***Extendable Framework.*** We design an extendable framework that provides an interface for plugging in other task allocation algorithms. This framework allows us to leverage state-of-the-art research in task scheduling [26] and graph partitioning [27]. Moreover, specific challenges in fuzzing may also pose interesting research questions for researchers working on these topics.
- ***Tool.*** We implemented a prototype called AFLTeam (https://github.com/MelbourneFuzzingHub/aflteam). Compared to AFL, AFLTeam achieved higher branch coverage (up to 16.4%) in our preliminary experiments.

II. Background and Related Work

To fuzz-test a target program using AFL, the program needs to be instrumented with additional instructions to collect the code coverage information at run-time. To that end, the AFL lightweight instrumentation pass injects the piece of code in Figure 2 at each branch point in the program. The variable `cur_loc` identifies the current basic block. The array `bit_map[]` is a shared memory region, which is accessible by both the fuzzer and the instrumented program. Each byte in the array stores the hit count for a particular branch of the program so it tells AFL how many times the branch has been taken by the current input. If the hit count is zero, it means the input does not cover that branch. Given two basic blocks A and B, the bitwise shift operation in line 3 preserves the directionality [(A, B) versus (B, A)].

```
1: cur_loc = < COMPILE_TIME_RANDOM >;
2: bit_map[cur_loc ^ prev_loc]++;
3: prev_loc = cur_loc >> 1;
```

Figure 2. AFL's instrumentation

As shown in Algorithm 1, AFL takes the instrumented program P and a seed corpus S to start its fuzzing process. In each iteration of the main fuzzing loop (lines 1–13), AFL chooses an input from the seed corpus (line 2) and calculates fuzzing energy for that input (line 3). The fuzzing energy [7] decides how many new inputs a fuzzer should produce from the selected seed. AFL uses different mutation operators (e.g., bit flipping, block deletion/insertion) to modify a given input (line 5). It then executes the program P with each newly generated input and monitors the program behaviors (lines 6–10). If P crashes, the crash-triggering input will be saved for further analysis (lines 7–8). Moreover, AFL analyzes the shared coverage bitmap to check if this input can uncover interesting program behaviors (e.g., new branches have been taken). If so, the input is retained for further rounds of fuzzing (lines 9–10). Otherwise, AFL just discards it.

Input: Instrumented Program P, Seed Corpus S
Output: Crashing Inputs $S_{✗}$
1: **repeat**
2: $s = \text{CHOOSE_NEXT}(S)$
3: $p = \text{ASSIGN_ENERGY}(s)$
4: **for** i from 1 to p **do**
5: $s' = \text{MUTATE_INPUT}(s)$
6: $\text{EXECUTE}(P, s')$
7: **if** s' crashes **then**
8: add s' to $S_{✗}$
9: **else if** $\text{IS_INTERESTING}(s')$ **then**
10: add s' to S
11: **if** $in_parallel_mode$ **then**
12: $\text{SYNC_FUZZERS}(S)$
13: **until** *timeout* reached or *abort*-signal

Algorithm 1. Main Fuzzing Loop of AFL/AFLTeam. The original algorithm is from the AFLFast paper [7]

https://t.me/learningnets

In AFL's default collaborative parallel fuzzing mode (AFL-P), each AFL fuzzing instance periodically checks the shared corpus directory for any new interesting inputs found by other fuzzing instances and copies them to its own seed corpus (lines 11–12). Task conflicts happen because different fuzzing instances might work on the same seeds, explore the same program search spaces, and produce similar test cases.

P-Fuzz and UniFuzz [24], [25] change the CHOOSE_NEXT function in Algorithm 1 so that each fuzzing instance can request the central node/database for the next task i.e., the next seed input. On the other hand, PAFL [23] aims to increase the diversity among fuzzing instances and drive them towards different areas in the program space. To achieve that goal, PAFL divides the coverage bitmap into continuous equal-size regions and statically assigns one bitmap region to each fuzzing instance as a task. The CHOOSE_NEXT function is modified so that each fuzzing instance focuses on mutating the inputs related to its assigned task. However, this solution appears to be problematic. First, since the indices of bytes in the bitmap array are randomly decided at instrumentation time (Figure 2), adjacent bytes in the bitmap might not correspond to connected branches on the control flow graph. Hence, each of PAFL's task is normally a collection of likely unrelated micro tasks. Second, because it employs a static allocation strategy, PAFL is unlikely to always achieve optimal results.

III. Our Approach

Our approach shares an observation with PAFL – we also aim to drive fuzzing instances towards different areas (i.e., sub-search-spaces) in the program space. However, unlike PAFL's ad-hoc implementation, our solution is *systematic and dynamic*. We build and dynamically update an attributed call graph of the program under test, partition it into sub-graphs, and consider each sub-graph as a task for a fuzzing instance.

A. Framework Overview

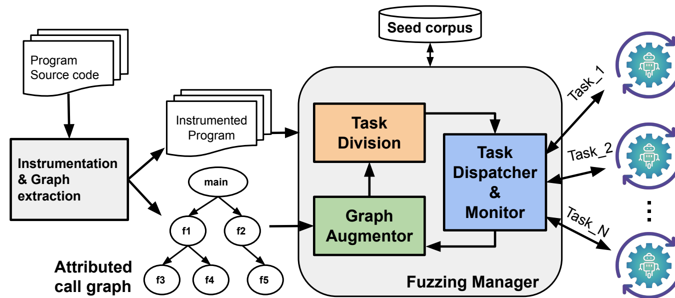

Figure 3. AFLTeam's workflow.

We design a modular framework for collaborative fuzzing called AFLTeam (Figure 3). Given an input program, AFLTeam instruments it and extracts a call graph from its LLVM Bitcode [28], [29]. The call graph is updated by the *Graph Augmentor* component with the function-call profiling information of the seed corpus (see Section III-B). Note that, since the seed corpus is continuously enlarged during fuzzing, the call graph is also getting more complete.

The *Task Division* component takes the most updated call graph and generates tasks in the form of sub graphs (see Section III-B). Figure 4 shows a sample partitioned call graph of a program that captures the main logic of media processing libraries like LibPNG [22]. The program takes a filename as input, reads the file, checks its validity (e.g., correct signature and checksums), extracts specific data chunks (e.g., chunk_A and chunk_B), and processes those chunks (e.g., data decoding). Our approach could partition the given call graph into four tasks based on different criteria (e.g., potential coverage improvement). While the first task focuses on fuzzing the logic of reading file and checking the file header, the second task checks if the data chunk parsing code is working correctly. The third and the fourth tasks stress test the data chunk processing code (e.g., data decoding).

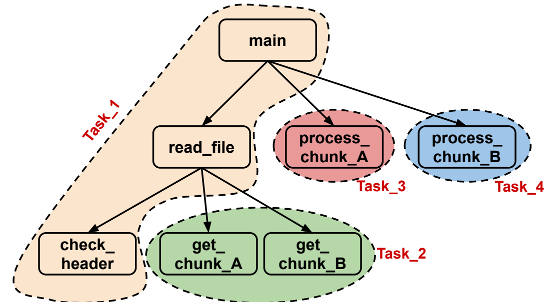

Figure 4. A sample partitioned callgraph.

As described in structure-aware fuzzing papers [10], [15], [30], program inputs can be highly structured and our approach would let fuzzing instances focus on specific parts of the file inputs (e.g., Task_1 focuses on the file header, Task_2 works on chunks' fields, and Task_3&4 mutate chunks' data). Our task allocation takes into account the fuzzing progress to update the tasks accordingly. For instance, after some time, if almost all branches of functions in Task_1 have been covered, our algorithm could merge them into another task. We could also divide a complicated task into sub tasks if necessary.

The *Task Dispatcher* instantiates new fuzzing instances and dispatches the tasks. The fuzzing instances should be task-aware so that they can guide the search towards functions in the assigned tasks (see Section III-C). The *Monitor* component collects results and statistical information from all fuzzing instances, makes the decision when to stop those instances, and asks *Task Division* to decide on a new set of tasks.

B. Systematic and Dynamic Task Allocation

Attributed call graph construction. Given a target program `P`, we build an attributed call graph $\langle V, E\rangle$ where each vertex in V represents a function of `P` and each edge (`f`, `g`) in E indicates a function call from `f` to `g`. Then, we attach the attributes (e.g., weights, discovery probability [31]) to all the vertices and edges to help our partitioning algorithm decide their importance. Those attributes are initialized at the beginning and periodically updated during a fuzzing campaign using the information collected from fuzzing instances. Specifically, we set the initial weight for each function/vertex based on its

https://t.me/learningnets

number of basic blocks which measures the potential of code coverage improvement if that function is targeted.

Graph partitioning. After constructing the attributed call graph, we partition it into sub-graphs such that (i) the total vertex weight sum is distributed evenly over sub-graphs to balance workload and (ii) the weight sum of edges connecting all different partitions is minimized to decrease parallel overhead. Graph partitioning is known to be a challenging problem, and different approaches have been proposed to solve it, such as the Kernighan-Lin and spectral partitioning algorithms [32], [33]. When being applied to fuzz-test real-world programs, the candidate graph partitioning algorithm must be scalable to handle large call graphs (e.g., the extracted call graph of FFmpeg library has 20,000+ vertices and 50,000+ edges). In our prototype, we supports Lukes' algorithm [34], [35], which is time and space efficient. In the future, we will integrate more advanced algorithms into our extendable framework.

Function call profiling. The completeness of the call graph is vital to the effectiveness of our partitioning algorithm. However, when we statically extract a call graph of a target program, the graph could miss many edges due to indirect calls, such as the use of function pointers. To tackle the issue, we modify AFL's instrumentation pass to enable a lightweight function-call profiling feature. Given a set of inputs, this feature produces a list of function calls collected during program executions which can be used to update the call graph.

C. Task-Aware Fuzzing

We update AFL's fuzzing algorithm (Algorithm 1) to take a task T (i.e., a set of functions) and guide the search towards those functions. Specifically, we keep the algorithm of the main fuzzing loop but make changes to core functions such as CHOOSE_NEXT, MUTATE_INPUT, IS_INTERESTING, and SYNC_FUZZERS. The changes will enable the capabilities to do (i) task-aware seed filtering, and (ii) task-aware mutation. AFLTeam's fuzzing instances use seed filtering to choose only seeds that reach the target functions from the seed corpus and discard others (function CHOOSE_NEXT). Seed filtering is also necessary to retain or import inputs that are interesting with respect to the task (functions IS_INTERESTING and SYNC_FUZZER). A task-aware mutation algorithm only modifies the parts of the seed inputs that might increase code coverage or trigger bugs in the target functions (function MUTATE_INPUT) and keeps the remaining parts unchanged.

Task-aware seed filtering. To enable this capability, we extend the AFL's instrumentation pass (Figure 2) so that the fuzzer can collect the function coverage information in addition to the supported code coverage. Given an input, its function coverage information lets the fuzzer know if that input has executed the target functions.

Task-aware mutation. Taint-based directed fuzzing [36] and information flow analysis [37] are two viable options to enable this capability. However, we are concerned about the high overhead of the former and the imprecision of the latter. We leave this problem for our immediate future work.

IV. Preliminary Evaluation

We developed a prototype of AFLTeam, which consists of approximate 900 lines of code (LoCs) in Python and 200 LoCs in C & C++, and evaluated its effectiveness in comparison with the default collaborative fuzzing mode of AFL (AFL-P). We could not report the performance of PAFL, P-Fuzz, and UniFuzz because these tools were not publicly available at the time of our experiments. Regarding subject programs, we selected the newest versions of four well-tested open source libraries: LibPNG [22], LibJPEG-turbo [38], FFmpeg [39], and JasPer [40] which are widely used in systems like Web browsers, and media processing/streaming services. These subjects have also been used in evaluating previous work [10] and are continuously tested on the Google OSS-Fuzz project [41].

In our experiments, we allocated AFLTeam and AFL-P the same resources on a Google Compute Engine with 32 virtual CPUs and 32GB of RAM. Specifically, for each subject we ran the fuzzers on 10 cores (one fuzzing instance per core) for 10 hours. The fuzzers were run with the same arguments and the same seed corpora as reported in the AFLSmart paper [10].

Fuzz target	Metric	AFL-P	AFLTeam	Improvement
pngimage	#lines	5112	5165	**1.04%**
	#branches	3007	3043	**1.20%**
djpeg	#lines	2577	2807	**8.93%**
	#branches	1493	1738	**16.41%**
jasper	#lines	5560	5609	**0.88%**
	#branches	2837	2897	**2.11%**
ffmpeg	#lines	14490	17557	**21.17%**
	#branches	9036	10505	**16.26%**

Table I. Preliminary results: pngimage (LibPNG), djpeg (LibJPEG-turbo), jasper (JasPer), and ffmpeg (FFmpeg)

Table I shows the code coverage achieved by AFLTeam and AFL-P. AFLTeam outperformed AFL-P in all subjects. Notably in LibJPEG-turbo and FFmpeg, AFLTeam's improvements over AFL-P on branch coverage are 16.41% and 16.26%, respectively. AFLTeam also discovered 2 crash-triggering bugs in FFmpeg and JasPer. We reported these bugs to the developers and both have been fixed.

Due to the page constraint, we provide more analyses (e.g., code coverage improvement over time) and the instructions to run experiments on AFLTeam's GitHub repository at https://github.com/MelbourneFuzzingHub/aflteam.

V. Conclusion and Future Directions

We have presented promising empirical evidence that leveraging graph partitioning and search can improve the effectiveness of collaborative parallel fuzzing. Our immediate future work is to complete the task-aware mutation algorithm, improve the graph partitioning algorithm, and run experiments at a larger scale. Since our solution works at the function call graph level, which is close to human understanding of input programs, we also plan to investigate the possibilities of involving humans in the loop. For instance, humans can monitor the task allocation and leverage their domain knowledge to guide the algorithm towards better decisions.

https://t.me/learningnets

REFERENCES

[1] Website, "American fuzzy lop (afl) fuzzer," http://lcamtuf.coredump.cx/afl/technical_details.txt, accessed: 2021-06-18.

[2] ——, "Libfuzzer: A library for coverage-guided fuzz testing," http://llvm.org/docs/LibFuzzer.html, accessed: 2021-06-18.

[3] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *TSE*, 2019.

[4] M. Boehme, C. Cadar, and A. ROYCHOUDHURY, "Fuzzing: Challenges and reflections," *IEEE Software*, 2020.

[5] Website, "Google clusterfuzz," https://google.github.io/clusterfuzz/, accessed: 2021-06-18.

[6] ——, "Microsoft onefuzz," https://github.com/microsoft/onefuzz, accessed: 2021-06-18.

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*, 2016.

[8] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[9] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for use-after-free vulnerabilities," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020.

[10] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," in *TSE*, 2019.

[11] M. Böhme, V. J. M. Manès, and S. K. Cha, "Boosting fuzzer efficiency: an information theoretic perspective," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[12] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 475–485, 2018.

[13] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, 2018.

[14] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "Mopt: Optimized mutation scheduling for fuzzers," in *USENIX Security Symposium*, 2019.

[15] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," in *NDSS*, 2019.

[16] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *NDSS*, 2019.

[17] Website, "Libprotobuf mutator," https://github.com/google/libprotobuf-mutator, accessed: 2021-06-18.

[18] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: guiding grey-box fuzzing towards combinatorial difference," *ICSE*, 2020.

[19] V. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

[25] X. Zhou, P. Wang, C. Liu, T. Yue, Y. Liu, C. Song, K. Lu, and Q. Yin, "Unifuzz: Optimizing distributed fuzzing via dynamic centralized task scheduling," *arXiv preprint arXiv:2009.06124*, 2020.

[20] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[21] Website, "Distributed fuzzing using afl," https://github.com/richo/roving, accessed: 2021-06-18.

[22] ——, "Libpng," http://www.libpng.org/pub/png/libpng.html, accessed: 2021-06-18.

[23] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "Pafl: extend fuzzing optimizations of single mode to industrial parallel mode," *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[24] C. Song, X. Zhou, Q. Yin, X. He, H. Zhang, and K. Lu, "P-fuzz: a parallel grey-box fuzzing framework," *Applied Sciences*, vol. 9, no. 23, p. 5100, 2019.

[26] J. Du and J. Leung, "Complexity of scheduling parallel task systems," *SIAM J. Discret. Math.*, vol. 2, pp. 473–487, 1989.

[27] U. Elsner, "Graph partitioning - a survey," 2005.

[28] Website, "The llvm compiler infrastructure," https://llvm.org/, accessed: 2021-06-18.

[29] ——, "Whole program llvm," https://github.com/travitch/whole-program-llvm, accessed: 2021-06-18.

[30] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[31] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 713–724.

[32] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[33] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *SIAM J. Sci. Comput.*, vol. 16, pp. 452–469, 1995.

[34] J. A. Lukes, "Efficient algorithm for the partitioning of trees," *IBM Journal of Research and Development*, vol. 18, no. 3, pp. 217–224, 1974.

[35] Website, "Network analysis in python," https://networkx.org/, accessed: 2021-06-18.

[36] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 474–484.

[37] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller, "Detecting information flow by mutating input data," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 263–273.

[38] Website, "Libjpeg-turbo," https://libjpeg-turbo.org/, accessed: 2021-06-18.

[39] ——, "Ffmpeg," https://www.ffmpeg.org/, accessed: 2021-06-18.

[40] ——, "Jasper image processing/coding tool kit," https://github.com/jasper-software/jasper, accessed: 2021-06-18.

[41] ——, "Oss-fuzz: Continuous fuzzing for open source software," https://github.com/google/oss-fuzz, accessed: 2021-06-18.

https://t.me/learningnets