Adaptive Parallel Fuzzing with Multi-candidate Task Scheduling

Sisi Li, Ruilin Li, Jiaxi Ye and Chaojing Tang
College of Electronic Science and Technology, National University of Defense Technology, Changsha, China, 410073
securitylrl@gmail.com

Abstract. Parallel fuzzing is a widely used technique for bug detection. It improves fuzzing performance by taking full advantage of the computing resource. Nevertheless, existing parallel fuzzing approaches fail to extend fuzzing optimizations of single mode to parallel mode, due to the lack of efficient information synchronization and task division. To address the challenge, researchers propose a parallel fuzzing framework that combines guiding information synchronization with task division based on branch bitmap to improve the performance of the fuzzers in parallel mode. However, when augmenting existing fuzzers with the parallel framework, we find two types of limitations caused by some mechanisms of the fuzzers, such as task scheduling mechanism. In this paper, we present an optimized parallel fuzzing approach to improve fuzzing efficiency by making valuable tasks executed as much as possible. On the one hand, we introduce a multi-candidate task scheduling mechanism, which takes multiple tasks corresponding to an input as candidates, rather than one to obtain the chances of executing tasks that are valuable but can be missed. On the other hand, we apply a synchronization information-centric design solution to some mechanisms of the fuzzers and take synchronization information as a good indicator to decide whether the parallel instances ought to shift their running mode, which can improve the adaptive ability of parallel instances and facilitate them to timely execute valuable tasks. We implement a prototype system OPAFL on top of Fairfuzz and PAFL, and evaluate its performance on several real-world software. The experimental results show that OPAFL can execute more tasks valuable and achieve higher branch coverage, compared with original parallel mode of Fairfuzz and Fairfuzz augmented with PAFL.

1. Introduction
Fuzzing [1] is one of the most effective techniques for software vulnerability discovery. It feeds the program with numbers of malformed inputs and monitors the unexpected program behaviors during the execution to uncover program defects. Fuzzing efficiency has been greatly improved by combination with some useful techniques, such as coverage guide, scheduling algorithms, static analysis, and symbolic execution. Specifically, coverage-guided fuzzing represented by AFL [2] can improve fuzzing performance by taking program running information as feedback. Some extensions of AFL are proposed. For example, Fairfuzz [3] achieves higher branch coverage at a faster rate than AFL by targeting branches hit by few inputs; AFLfast [4] improves code coverage by prioritizing seeds exercising low-frequency paths and allocating more testing energies to them.
Parallel fuzzing is another way to improve fuzzing efficiency, which exploits abundant computing resource to run multiple parallel nodes simultaneously and synchronizes some useful information...
among parallel instances, to shorten the time of discovering vulnerabilities and achieve higher code coverage on large real-world software. For example, Google’s OSS-Fuzz [5] platform adopts several state-of-the-art fuzzers, including AFL, libFuzzer [6] and honggfuzz [7], to continuously test open source applications; EnFuzz [8] combines several base fuzzers on the basis of their diversities to improve generalization capability of fuzzing; AFL comes with parallel mode, and its parallel instances conduct the same task and just synchronize the interesting seeds, hence serious task duplication decreases performance improvement; PAFL [9] utilizes efficient information synchronization and task division to extend fuzzing optimizations of single mode to parallel mode. Particularly, it splits the fuzzing task into several subtasks based on branch bitmap, which tactfully avoids duplicate tasks among parallel instances.

Efficient guiding information synchronization, as well as task division based on branch bitmap, has been proven to improve the performance of the fuzzers in parallel mode. However, when augmenting existing fuzzers with the parallel framework, we find that some mechanisms of the fuzzers, such as task scheduling mechanism, may incur two types of limitations. On the one hand, when task scheduling mechanism of the fuzzers combines with task division based on branch bitmap, many valuable tasks would miss the chance to be executed and there exists the unbalanced number of tasks handled by different instances in parallel system. On the other hand, some mechanisms of the fuzzers neglect the guiding significance of synchronization information, as a result, the fuzzer instances fail to make the best of synchronization information to timely execute the valuable tasks.

To address the limitations mentioned above, we present an optimized parallel fuzzing approach, which can be illustrated from two aspects: (1) we put forward a multi-candidate task scheduling mechanism that takes multiple tasks corresponding to an input as candidates, rather than one, which enables lots of valuable tasks to obtain the chance of being executed and helps balance the number of tasks among different parallel instances; (2) we apply a synchronization information-centric design solution to some mechanisms of existing fuzzers to improve the adaptive ability of each fuzzer instance, thus it can timely shift its running mode to execute the valuable tasks according to the guiding information synchronized from other parallel instances, which can improve the possibility of discovering new branches.

We implement a prototype as OPAFL and evaluate its performance on several real-world software. The results show that OPAFL can execute more tasks valuable and achieve higher branch coverage, compared with other parallel systems, which proves the effectiveness of our approach.

2. Background and motivation
In this paper, we take Fairfuzz as an example to illustrate two types of limitations caused by some mechanisms of existing fuzzers, such as task scheduling mechanism, when augmented with the parallel fuzzing framework.

2.1. Mechanisms in Fairfuzz
Fairfuzz prioritizes inputs covering rare branches and adopts a novel mutation strategy to increase the probability of hitting these rare branches, which facilitates to explore hard-to-reach codes guarded by rare branches. Rare branch means a branch exercised by few inputs, and hit count is utilized to record the number of inputs that hit the branch. Here are two crucial mechanisms in Fairfuzz.

Rare Branch Scheduling Mechanism. It should be emphasized that we take test of a rare branch as a task and rare branch scheduling is equivalent to task scheduling. Firstly, Fairfuzz only selects inputs that hit a rare branch to mutate. Secondly, when an input hits multiple rare branches, Fairfuzz selects the rarest one (the one hit by the fewest inputs) to test. Thirdly, if more than one rare branch is the rarest, the one with the largest branch id will be chosen.

Mode Switch Mechanism. Note that Fairfuzz has two running modes, “RB” mode that focuses on testing rare branches, and “AFL” mode. The mode switch mechanism depends on a configurable parameter $q$, which has four alternative values that can be divided into two types. For the first case, the default value of $q$ is 0, which means Fairfuzz runs in “RB” mode all the time. However, without a
switch to “AFL” mode, Fairfuzz could get stuck. What’s more, testing a rare branch can incur more overhead, so it is unnecessary to test rare branches whose hit counts are quite large. For the second case, the other three values denote that Fairfuzz will switch to “AFL” mode if it discovers no new branches during a queue cycle in “RB” mode. Specifically,

- **q=1** means it does not switch to “RB” mode until a new branch is discovered.
- **q=2** means it skips deterministic fuzzing and does not switch to “RB” mode until a new branch is discovered.
- **Figure 1** illustrates the mode switch mechanism that corresponds to **q=3**. Fairfuzz switches to “RB” mode after running for a queue cycle (arrow 2 in figure 1), and it can early return “RB” mode if it discovers a new branch during the cycle (arrow 1 in figure 1).

By taking the characteristics of two modes into full account, in this paper, we choose the mode switch mechanism corresponding to **q=3**.

![Diagram of mode switch mechanism](image)

**Figure 1.** The mode switch mechanism corresponding to **q=3**.

2.2. Task division strategy in PAFL

Parallel fuzzing is widely used for software testing. In order to extend fuzzing optimizations of single mode to parallel mode, a parallel fuzzing framework named PAFL is proposed, which utilizes efficient guiding information synchronization and task division strategy to improve the performance of the fuzzers in parallel mode. Particularly, its task division based on branch bitmap can effectively avoid task duplication among parallel instances, as it equally splits the branch bitmap into multiple areas and makes different fuzzer instances focus on different bitmap areas. Note that Fairfuzz targets rare branches and attempts to explore code regions guarded by them, in hence, the task division strategy in PAFL is naively fit for Fairfuzz. Algorithm 1 shows the task scheduling algorithm of Fairfuzz augmented with PAFL.

As shown in Algorithm 1, branch bitmap is equally split into **n** intervals on Line 1 and each fuzzer instance corresponds to a different range of bitmap positions on the basis of its index on Line 2-3. For a given input, the id of its rarest branch is computed on Line 5. If the branch id computed is not in the corresponding bitmap range, the fuzzer instance will skip the current input and try next one on Line 7. Otherwise, the fuzzer instance performs normal fuzzing process.
2.3. Motivation

As is known, for the parallel fuzzing system, valuable tasks should be executed as much as possible to improve fuzzing performance, and existing fuzzers have their own indicators to decide whether the task is valuable. Let us take Fairfuzz as an example, it is proven that when an input hits rare branches and the rarest one’s hit count is small enough, mutating this input in “RB” mode, rather than “AFL” mode, is more likely to discover new branches. Therefore, we consider test of a rare branch whose hit count is small as a valuable task and conclude that rare branches with quite small hit counts should be tested in “RB” mode as much as possible.

\begin{algorithm}
\caption{Task Scheduling Algorithm}
\textbf{Input:} The number of fuzzer instances: \textit{n},
the index of the fuzzer instance: \textit{m},
input \textit{s},
branch hit count map \textit{branch.hit.count}.
1: \textit{interval} = MAP\_SIZE/\textit{n}
2: \textit{start} = (\textit{m}−1) * \textit{interval}
3: \textit{end} = \textit{m} * \textit{interval}
4: \textit{rare.branches} = FINDCOVEREDRAREBRANCH(\textit{s}, \textit{branch.hit.count})
5: \textit{min.hit.id} = FINDMINHitID(\textit{rare.branches}, \textit{branch.hit.count})
6: \textit{if} \textit{min.hit.id} < \textit{start} or \textit{min.hit.id} >= \textit{end} \textit{then}
7: \quad \text{Try next input}
8: \textit{end if}
\end{algorithm}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig2.pdf}
\caption{The number of rare branches processed by each parallel node.}
\end{figure}
In order to assess the number of tasks executed by each parallel node in “RB” mode, we conducted experiments on Fairfuzz augmented with PAFL using several real-world programs, including `readelf`, `objdump`, `tcpdump`, and `djpeg`, and counted the number of rare branches processed by each fuzzer node. For each run, we start four parallel nodes that last 24 hours, which means the whole task is split into four parts and each node corresponds to one part, hence there will be no task duplication among different nodes. Our experiments are repeated for three times. The experimental results are demonstrated in figure 2. We observe that the fuzzer node with a smaller index tends to process fewer rare branches, especially for `readelf` and `tcpdump`. After in-depth analysis of the rare branches tested, we find a fact that multiple rare branches hit by an input actually have the same hit count and the fuzzer instance merely selects the one with the largest branch id as target branch. Then, rare branches with a larger branch id is more likely to be tested.

We can find two types of limitations caused by some mechanisms of existing fuzzers when augmented with the parallel framework. On the one hand, most of existing fuzzers provide only one candidate task for each input despite there are multiple valuable tasks corresponding to the input. As a result, when task scheduling mechanism of the fuzzer combines with task division based on branch bitmap, many valuable tasks miss the chance to be executed and there exists the unbalanced number of tasks handled by parallel instances, which decreases parallel fuzzing efficiency.

On the other hand, some mechanisms of the fuzzers lack considerations for making use of the guiding function of synchronization information, as a consequence, the fuzzer instance fails to timely shift its running mode to execute the valuable tasks even though it has synchronized the critical information. Take the mode switch mechanism of Fairfuzz as an example, in “AFL” mode, if the fuzzer instance does not finish a queue cycle and finds no new branches, even if it has synchronized rare branches whose hit counts are very small from other parallel instances, it won’t timely return “RB” mode to test those valuable rare branches, hindering the discovery of new branches.

In summary, for the parallel fuzzing system, valuable tasks ought to be executed as much as possible to increase the possibility of discovering new branches. Therefore, some crucial mechanisms of existing fuzzers, such as task scheduling mechanism need to be optimized to better fit for the parallel framework and to execute more tasks valuable, hence improving fuzzing performance.

3. Design of OPAFL
In order to address the aforementioned limitations, we propose an optimized parallel fuzzing approach. It adopts a multi-candidate task scheduling mechanism and applies a synchronization information-centric design solution to some mechanisms of the fuzzers, which can make parallel instances execute more tasks valuable and improve parallel fuzzing efficiency.

3.1. Multi-candidate task scheduling mechanism
In this section, we introduce a multi-candidate task scheduling mechanism, which always takes multiple tasks corresponding to an input as candidates, rather than one, to acquire the chances of executing tasks that are valuable but would be missed and help balance the number of tasks processed by parallel instances.

In specific, we take Fairfuzz as an example to show the multi-candidate task scheduling strategy in detail. It is known that multiple rare branches hit by an input always have the same hit count, meanwhile we are agnostic about the programs, hence we assume that these rare branches have the value of equal importance. In order to test valuable rare branches as much as possible, we sort rare branches an input hits in descending order by branch ids and take all of them as candidates to be tested, rather than the only one with the largest branch id. Algorithm 2 illustrates the multi-candidate task scheduling algorithm of Fairfuzz in parallel system.

As shown in Algorithm 2, for a given input, its rare branches are sorted in descending order by branch ids on Line 5. If the rare branch with the largest branch id is not in the corresponding bitmap range, the fuzzer instance selects the one with the second largest branch id, and so on. If all the rare branches are beyond the bitmap range, the fuzzer instance skips the current input and tries next one on Line 15.
Otherwise, it performs normal fuzzing process. It can be seen that the multi-candidate task scheduling algorithm is able to make the fuzzer instance execute lots of valuable tasks that would be missed and help balance the number of tasks among different parallel instances, thus improving the possibility of discovering new branches.

**Algorithm 2** Multi-candidate Task Scheduling Algorithm

**Input:** The number of fuzzer instances: \( n \);
the index of the fuzzer instance: \( m \);
input \( s \);
branch hit count map \( branch\_hit\_count \).

1. \( interval = MAP\_SIZE/n \)
2. \( start = (m - 1) \times interval \)
3. \( end = m \times interval \)
4. \( rare\_branches = FINDCOVEREDRAREBRANCH(s, branch\_hit\_count) \)
5. \( Array = SORTIDBYDESCENDINGORDER(rare\_branches) \)
6. \( num = GETARRAYSIZE(Array) \)
7. for \( i = 0; i < num; i + + \) do
8. \( branch\_id = Array[i] \)
9. if \( branch\_id >= start \) and \( branch\_id < end \) then
10. goto final
11. end if
12. end for
13. final
14. if \( i = num \) then
15. Try next input
16. end if

3.2. Synchronization information-centric mode switch

In this section, we present a synchronization information-centric design solution, which would take synchronization information as a good indicator to decide whether the parallel instances ought to shift their running mode. By applying it to some mechanisms of existing fuzzers, we can improve the adaptive ability of parallel instances and achieve higher fuzzing performance.

**Figure 3.** The optimized mode switch mechanism.
Let us take the mode switch mechanism of Fairfuzz as an example to illustrate the synchronization information-centric design solution. The original mode switch mechanism lacks considerations for making use of the guiding function of synchronization information, thus delaying the executions of valuable tasks. To address the limitation, we apply a synchronization information-centric design solution to the existing mode switch mechanism.

Figure 3 shows the optimized mode switch mechanism. Arrow 3 in figure 3 denotes the mode switch strategy we proposed, that is, in “AFL” mode, even if the fuzzing instance does not finish a queue cycle and finds no new branches, it will switch to “RB” mode once it has synchronized the seeds that discover a new branch. It can be empirically found that the newly discovered branches by the fuzzing instances usually have quite small hit counts, meanwhile, rare branches with small hit counts are considered to be valuable. Therefore, we regard seeds covering new branches as a good indicator to decide whether the fuzzing instance ought to return “RB” mode to test the valuable rare branches. With the synchronization information-centric design solution, the parallel instance can timely shift its running mode to execute valuable tasks according to synchronization information, which improves the adaptive ability of the fuzzing and facilitates to discover more new branches, thus improving the efficiency of parallel fuzzing.

4. Evaluation

4.1. Experimental setting
We implemented OPAFL on top of Fairfuzz and PAFL, and evaluated it on five real-world programs, which came from binutils-2.32 [10], libjpeg-turbo-1.5.1 [11], and tcpdump-4.9.2 [12], including readelf, objdump, nm, djeg, and tcpdump. To verify the effectiveness of OPAFL, we compared it with original parallel mode of Fairfuzz and Fairfuzz augmented with PAFL. Each experiment was run for 24 hours with four parallel instances. In order to avoid the potential randomness, we repeated each experiment for ten times and reported the mean values. Note that we did not use advanced seeds and AddressSanitizer [13] during the test.

All our experiments were conducted on a server with an Intel Xeon Gold 6154 CPU@3.0GH (72 cores) and 512GB memory, running 64-bit Ubuntu 18.04 LTS system.

4.2. Evaluation metrics
We use two metrics to evaluate OPAFL. The first metric is the number of rare branches processed by each fuzzing instance. OPAFL aims to execute valuable tasks as much as possible to improve the possibility of discovering new branches, hence the first metric can be used to evaluate the effectiveness of OPAFL. The second metric is branch coverage, namely the number of branches covered, which is often utilized to estimate fuzzing performance.

4.3. Results and analysis
The Number of Rare Branches Processed. Table 1 shows the average number of rare branches processed by each fuzzing node after ten times of experiments. For all the programs, as shown in table 1, OPAFL can test a larger number of rare branches on four parallel nodes respectively, compared with Fairfuzz augmented with PAFL. Moreover, the variable \textit{Var} denotes the variance of the number of rare branches processed by the four parallel nodes. As we can see from the last column of the table 1, the \textit{Var} of PAFL is larger than that of OPAFL on all the programs, especially for \textit{nm} and tcpdump, which means OPAFL can make the number of rare branches processed much closer to each other among the four nodes, thus effectively balancing the number of tasks among parallel nodes.

OPAFL modifies the original rare branch scheduling mechanism and takes multiple rare branches hit by an input as candidates to be tested, rather than the only one with the largest branch id, hence when the one with the largest branch id is not in the corresponding bitmap range, the others still have the chance of being tested. Moreover, OPAFL adds a synchronization information-centric mode switch strategy to original mode switch mechanism, so the parallel instances can timely return “RB” mode to
test valuable rare branches according to synchronization information. As a result, OPAFL obtains more opportunities to execute valuable tasks and finally tests a larger number of rare branches. To emphasize, we didn’t compare our tool with original parallel mode of Fairfuzz on the first metric, as the later actually executed large numbers of repeating tasks among parallel instances.

### Table 1. The number of rare branches processed.

| Programs | Node1 | Node2 | Node3 | Node4 | Total | Var |
|----------|-------|-------|-------|-------|-------|-----|
| readelf  | PAFL  | 47    | 94    | 112   | 133   | 386 | 1007 |
|          | OPAFL | 236   | 258   | 283   | 261   | 1038| 277  |
|          |        |       |       |       |       | (1) |
| objdump  | PAFL  | 8     | 101   | 85    | 183   | 377 | 3862 |
|          | OPAFL | 124   | 215   | 178   | 221   | 738 | 1491 |
|          |        |       |       |       |       | (2) |
| nm       | PAFL  | 7     | 146   | 181   | 291   | 625 | 10288 |
|          | OPAFL | 84    | 259   | 226   | 297   | 866 | 6483 |
|          |        |       |       |       |       | (3) |
| djpeg    | PAFL  | 2     | 16    | 41    | 103   | 162 | 1497 |
|          | OPAFL | 28    | 77    | 67    | 135   | 307 | 1466 |
|          |        |       |       |       |       | (4) |
| tcpdump  | PAFL  | 64    | 171   | 239   | 378   | 852 | 12967 |
|          | OPAFL | 522   | 620   | 618   | 592   | 2352| 1574 |
|          |        |       |       |       |       | (5) |

The number of branches covered. Table 2 shows the number of branches covered by OPAFL and Fairfuzz in parallel mode, which are separately denoted as \( N_{\text{opafl}} \) and \( N_{\text{rb}} \), and we define a variable \( Ga \), denoted as \( (N_{\text{opafl}} - N_{\text{rb}}) / N_{\text{rb}} \), to describe the performance of OPAFL. As shown in table 2, compared with original parallel mode of Fairfuzz, OPAFL achieves higher branch coverage on all the benchmarks and it covers 4.39% more branches in total, which proves the effectiveness of OPAFL.

### Table 2. The number of branches covered.

| Programs | Fairfuzz | OPAFL | \( Ga \) |
|----------|----------|-------|--------|
| readelf  | 7723     | 7883  | 2.07% |
| objdump  | 2415     | 2662  | 10.23%|
| nm       | 1972     | 2045  | 3.7%  |
| djpeg    | 1729     | 1770  | 2.37% |
| tcpdump  | 8381     | 8835  | 5.42% |
| Total    | 22220    | 23195 | **4.39%** |

However, there are two possible limitations that threaten the performance of OPAFL. Firstly, although OPAFL can execute more tasks valuable, it performs badly in discovering new branches when
executing those tasks, which instead brings a performance loss and decreases fuzzing efficiency. Secondly, as we are agnostic about the programs, it is challenging to distinguish the importance of different tasks, thus we equally treat all the tasks to be executed, which results in an unreasonable distribution of energy, decreasing the effectiveness for OPAFL.
In a nutshell, the experimental results indicate that OPAFL outperforms original parallel mode of Fairfuzz and Fairfuzz augmented with PAFL. In hence, we can conclude that the optimized parallel fuzzing approach we proposed is capable of improving fuzzing efficiency.

5. Conclusion and future work
When augmenting existing fuzzers with the parallel fuzzing framework PAFL, we find two types of limitations caused by some mechanisms of the fuzzers, such as task scheduling mechanism. To address the limitations, in this paper, we propose an optimized parallel fuzzing approach. On the one hand, we present a multi-candidate task scheduling mechanism and take multiple tasks corresponding to an input as candidates, rather than one, enabling more tasks valuable to obtain the chance of being executed. On the other hand, we utilize a synchronization information-centric design solution to improve the adaptive ability of existing fuzzers in parallel mode, which takes synchronization information as a good indicator to decide whether the parallel instance ought to shift its running mode, facilitating the fuzzer instance to timely execute valuable tasks and improving fuzzing efficiency. We implement a prototype system OPAFL based on Fairfuzz and PAFL and conduct evaluation using some real-world software. The experimental results show that OPAFL can execute more tasks valuable and achieve higher branch coverage compared with original parallel mode of Fairfuzz and Fairfuzz augmented with PAFL.
However, there is a limitation in our approach. We didn't distinguish the importance of different tasks and equally treated all the tasks to be executed, thus rendering an unreasonable distribution of energy and low performance improvement. In the future work, we will adopt some other metrics to measure the importance of different tasks. Here are two possible metrics. Firstly, we can compute the number of basic blocks reachable from each rare branch by analyzing program structures, and the greater the number of basic blocks, the higher the value of rare branches. Secondly, we can utilize some program analysis techniques to calculate the number of vulnerable locations in code regions reachable from each rare branch, and the greater the number of vulnerable locations, the more important rare branches are. Based on the above two metrics, we are able to rank the tasks. Finally, we will prioritize tasks that are more important and allocate more testing energies to them, thus improving the possibility of discovering new branches.

6. References
[1] Miller B P, Fredriksen L and So B 1990 An empirical study of the reliability of UNIX utilities Communications of the ACM pp 32-44 https://doi.org/10.1145/96267.96279
[2] American fuzzy lop Retrieved from http://lcamtuf.coredump.cx/afl/
[3] Lemieux C and Sen K 2018 FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering pp 475-485 New York USA: ACM Press https://doi.org/10.1145/3238147.3238176
[4] Bohme M, Pham V-T and Roychoudhury A 2019 Coverage-Based Greybox Fuzzing as Markov Chain IEEE Transactions on Software Engineering pp 489-506 https://doi.org/10.1109/TSE.2017.2785841
[5] OSS-Fuzz Retrieved from https://github.com/google/oss-fuzz
[6] LibFuzzer Retrieved from https://github.com/Dor1s/libfuzzer-workshop
[7] Honggfuzz Retrieved from http://honggfuzz.com
[8] Chen Y, Jiang Y, Ma F, Liang J, Wang M, Zhou C and Jiao X 2018 EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers *ArXiv E-Prints* Retrieved from http://arxiv.org/abs/1807.00182

[9] Liang J, Jiang Y, Chen Y, Wang M, Zhou C and Sun J 2018 PAFL: extend fuzzing optimizations of single mode to industrial parallel mode *In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* pp 809-814 New York USA: ACM Press https://doi.org/10.1145/3236024.3275525

[10] Binutils Retrieved from http://ftp.gnu.org/gnu/binutils/

[11] Libjpeg-turbo Retrieved from https://sourceforge.net/projects/libjpeg-turbo/files/

[12] Tcpdump Retrieved from http://www.tcpdump.org/

[13] Serebryany K, Bruening D, Potapenko A and Vyukov D 2012 AddressSanitizer : A Fast Address Sanity Checker *In USENIX ATC ’12: Proceedings of the 2012 USENIX conference on Annual Technical Conference* p 28

**Acknowledgments**

We are grateful to the anonymous reviewers for their hard work, insightful comments and suggestions.