KRAKEN: Program-Adaptive Parallel Fuzzing

ANSHUNKANG ZHOU, Hong Kong University of Science and Technology, China
HEQING HUANG*, City University of Hong Kong, China
CHARLES ZHANG, Hong Kong University of Science and Technology, China

Parallel fuzzing, which utilizes multicore computers to accelerate the fuzzing process, has been widely used in industrial-scale software defect detection. However, specifying efficient parallel fuzzing strategies for programs with different characteristics is challenging due to the difficulty of reasoning about fuzzing runtime statically. Existing efforts still use pre-defined tactics for various programs, resulting in suboptimal performance.

In this paper, we propose Kraken, a new program-adaptive parallel fuzzer that improves fuzzing efficiency through dynamic strategy optimization. The key insight is that the inefficiency in parallel fuzzing can be observed during runtime through various feedbacks, such as code coverage changes, which allows us to adjust the adopted strategy to avoid inefficient path searching, thus gradually approximating the optimal policy. Based on the above insight, our key idea is to view the task of finding the optimal strategy as an optimization problem and gradually approach the best program-specific strategy on the fly by maximizing certain objective functions. We have implemented Kraken in C/C++ and evaluated it on 19 real-world programs against 8 state-of-the-art parallel fuzzers. Experimental results show that Kraken can achieve 54.7% more code coverage and find 70.2% more bugs in the given time. Moreover, Kraken has found 192 bugs in 37 popular open-source projects, and 119 of them are assigned with CVE IDs.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Greybox Fuzzing, Parallel Fuzzing

ACM Reference Format:
Anshunkang Zhou, Heqing Huang, and Charles Zhang. 2025. KRAKEN: Program-Adaptive Parallel Fuzzing. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA013 (July 2025), 25 pages. https://doi.org/10.1145/3728882

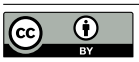
1 Introduction

Fuzzing is one of the most effective ways of detecting vulnerabilities in software systems. It works by feeding a large number of inputs to the target program to trigger unintended behaviors such as crashes, hangs, or assertion failures. Traditional techniques usually focus on maximizing code coverage or the number of detected bugs in a single fuzzer by prioritizing inputs [8, 9, 26, 50], controlled input generation [61, 70], combining with symbolic execution [17, 38, 79], and optimized instrumentation [15, 54, 82].

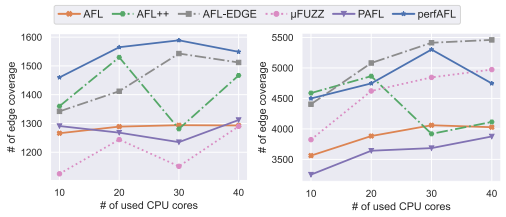
Despite numerous advances, most existing fuzzers still require more than 24 hours to thoroughly test the target programs to achieve satisfactory code coverage or bug detection results [7, 32, 42, 64]. Recently, as cloud-based computing and multicore computers have become more and more prevalent, parallel fuzzing [28, 29, 47, 75, 77, 83] has emerged as a new direction for improving fuzzing efficiency. It works by running multiple fuzzers simultaneously with information sharing between instances.

*Corresponding author

Authors' Contact Information: Anshunkang Zhou, Hong Kong University of Science and Technology, Hong Kong, China, azhouad@cse.ust.hk; Heqing Huang, City University of Hong Kong, Hong Kong, China, heqhuang@cityu.edu.hk; Charles Zhang, Hong Kong University of Science and Technology, Hong Kong, China, charlesz@cse.ust.hk.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2994-970X/2025/7-ARTISSTA013
https://doi.org/10.1145/3728882



(a) libpng (b) libtiff

Fig. 1. Accumulated edge coverage of fuzzing two Magma [32] programs with a different number of CPU cores for 3 hours.

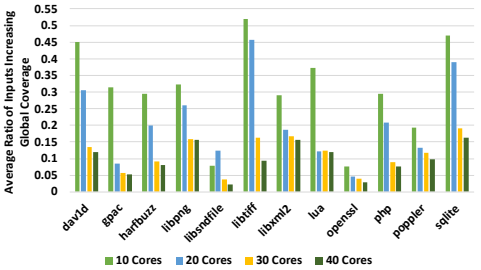


Fig. 2. Average contribution ratio of each instance to the global code coverage of fuzzing programs used in § 5.2 with AFL++ [3] for 3 hours.

Parallel fuzzing has great potential to speed up the testing process by reducing the required fuzzing time from several days to several hours while still achieving similar code coverage [7, 83].

However, we found that existing parallel fuzzers still suffer from a fundamental problem: they are not *adaptive* to programs with different characteristics because their fuzzing strategies are usually statically determined and remain the same across various projects. Figure 1 shows the edge coverage of using existing parallel fuzzers to test two Magma [32] programs with different numbers of CPU cores for 3 hours. As shown in the figure, existing parallel fuzzers still have unstable performance across different projects, i.e., performing well on some programs while badly on others. We mainly observe two root causes.

First, more computation cores do not necessarily result in better or worse performance. Intuitively, given the same amount of time, the code coverage should increase monotonically with the number of computation cores used. However, we found that such intuition may not hold in real-world cases. All existing parallel fuzzers still use a pre-defined number of CPU cores (parallelism degree) for fuzzing, making their performances vary across projects. For example, perfAFL [77] performs the best with 30 cores in Figure 1, while μFuzz [16] reaches the best performance at 40 cores. However, for `libpng`, μFuzz performs the best with 20 or 40 cores.

Second, the same input selection strategy does not consistently perform well. Some fuzzers [1, 77] let each instance adopt the same input selection strategy as the single-node fuzzing. They evaluate all the inputs and "intensify" the most promising ones for mutation, meaning that all instances tend to fuzz a similar set of inputs. Several recent approaches [47, 75] have designed new task distribution strategies that focus on maximizing search diversity in different instances. They partition the program into fragments, such as paths [47, 75] and functions [62], and let instances focus on inputs that cover different code parts. However, as illustrated in Figure 1, merely performing diversified or intensified input selection still could not consistently achieve good results. For example, perfAFL outperforms AFL-EDGE when fuzzing `libpng` while AFL-EDGE beats perfAFL on `libtiff`.

In this paper, we propose Kraken, a new program-adaptive parallel fuzzer that improves fuzzing efficiency through dynamic strategy optimization. Our key insight to solve the above problems is that the inefficiency in parallel fuzzing can be observed through various runtime feedbacks such as code coverage changes, which allows us to dynamically adjust the adopted strategy to avoid inefficient path searching, thus gradually approximating the optimal strategy. Since precisely reasoning about fuzzing runtime behaviors is extremely difficult, it is impossible to determine the real "optimal strategy". Instead, following numerous efforts on solving undecidable problems [8, 11, 50], our key idea is to view the problem of finding optimal strategy as an optimization problem, and gradually approach the best program-specific strategy on the fly by maximizing certain objective functions.

Specifically, Kraken features two algorithms to overcome the two limitations above, i.e., suboptimal parallelism degree and ineffective input selection. Kraken leverages the runtime feedback, such as coverage changes, to measure the "fitness" of the current strategy and dynamically adjusts it to achieve better ones. First, to find the optimal number of parallelized fuzzers, Kraken models the parallel fuzzing process as a multinomial-Dirichlet compound distribution and leverages runtime statistical data to update its hyper-parameters constantly. The probability model is used to estimate the fuzzing efficiency, and Kraken uses a simulated annealing-inspired algorithm to activate/deactivate fuzzing instances to maximize the efficiency dynamically. Second, to balance the intensification and diversification of the input selection in sub-instances, Kraken partitions the program flow graph into a compound hierarchical representation and instruments the target binary to obtain runtime information about search diversity in sub-instances. By evaluating how the current strategy affects the coverage results of already selected inputs, Kraken adopts a metaheuristic, the ant colony optimization (ACO) algorithm, to dynamically adjust the diversity degree of the input selection in sub-instances.

We evaluated Kraken on 19 real-world programs against 8 state-of-the-art parallel fuzzers. Experimental results show that Kraken achieves 54.7% more code coverage and finds 70.2% more bugs on average in the given time. Moreover, Kraken has found 192 bugs in 37 popular open-source projects, and 119 of them are assigned with CVE IDs. We will open source Kraken to facilitate future research and bug detection. In summary, this paper makes the following contributions:

- We propose a new program-adaptive parallel fuzzer Kraken together with two new algorithms to automatically optimize the fuzzing strategy during runtime.
- We conduct large-scale experiments to show that Kraken has better performance than existing works in terms of both code coverage and bug detection.
- We have found 192 bugs in 37 popular open-source projects with 119 CVE IDs assigned.
- We will open source Kraken to facilitate future research and bug detection.

2 Background and Motivation

In this section, we first introduce how state-of-the-art parallel fuzzing approaches work (§ 2.1). Then, we illustrate the limitations of existing techniques (§ 2.2). Finally, we summarize the challenges we attempt to resolve and present our solutions (§ 2.3).

2.1 Existing Efforts

Parallel fuzzing [29, 47, 75, 77] expedites the fuzzing process by running multiple fuzzer instances simultaneously on multi-core machines.

Naive parallel mode. Most existing single-node fuzzers [1, 9, 26, 71, 76] such as AFL natively support a parallel mode, which runs multiple instances of the same fuzzer concurrently and performs synchronization between instances periodically to exchange local findings.

Reducing overhead. Some approaches focus on reducing the execution or synchronization overhead to speed up parallel fuzzing. For example, perfAFL [77] designs new system primitives to solve performance bottlenecks of parallel fuzzing. They replace the `fork()` system call with a lightweight `snapshot()` and substitute the disk file system with the in-memory file system. μFuzz [16] breaks the serial fuzzing loop into four self-contained microservices to address the CPU cycle-wasting issue and partitions the fuzzing state among workers to avoid synchronization delay.

Principled task distribution. Some methods [28, 47, 62, 75] propose principled task distribution strategies to reduce duplicated path searching in parallelized instances. Since the naive parallel mode lets sub-instances explore program paths independently with similar strategies, they likely perform redundant or repeated path searches, causing performance degradation. To solve this

problem, PAFL [47] divides the bitmap into intervals and makes different fuzzer instances only focus on specific areas. AFLTeam [62] captures the program call graph dynamically and divides it into sub-graphs evenly with graph segmentation algorithms, and these sub-graphs serve as guidance for input selection or mutation in sub-instances. AFL-EDGE [75] designs a greedy algorithm on the program control-flow graph to divide already queued inputs into mutually exclusive while similarly-weighted sub-sets, which will be assigned back to fuzzing instances as guidance for input mutation. glibFuzzer [28] divides the global corpus into groups based on multiple features such as the seed size, execution speed, and the degree of difference in the coverage metric. Then, it randomly selects multiple seeds from the groups as the local corpora for sub-instances.

Collaborative fuzzing. Other approaches run different combinations of fuzzers instead of individual ones to improve the parallel fuzzing efficiency, which is also called collaborative fuzzing [13, 24, 29, 57]. EnFuzz [13] ensembles a number of diverse fuzzers by self-defined diversity heuristics. Cupid [29] generalizes the selection intuition and proposes an automated data-driven selection method. autofz [24] proposes an automated meta-fuzzer to make fine-grained adjustments to the resource allocation of base fuzzers. CollabFuzz [57] proposes a framework for scheduling test cases among different fuzzers by centralized analysis.

2.2 Limitations of Existing Techniques

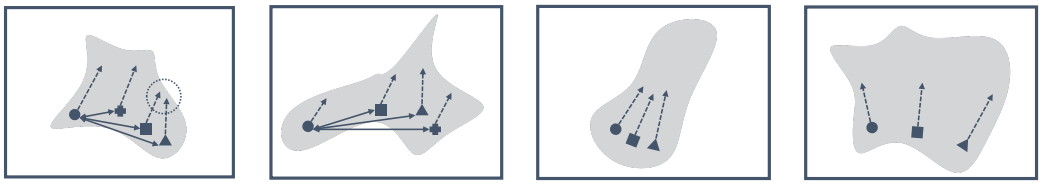
Most existing parallel fuzzers still suffer from a fundamental problem: they are not *adaptive* to programs with different characteristics because their fuzzing strategy is statically determined and remains the same across various projects. Using a fixed strategy for all programs could result in unstable performance, i.e., performing well on some programs while badly on others (Figure 1).

Fuzzing can be viewed as a random search process [10, 49], where the search space is the whole input space and the solutions are inputs that can cover new program code. Although the search space is infinite for any program, the solution space could vary across programs. Similarly, parallel fuzzing can also be viewed as a parallel random search process on the input space. There are two essential aspects of the parallel random search process: **the number of simultaneous searches** and **the search direction**, which corresponds to the **parallelism degree** (the number of parallelized fuzzers) and **the input selection method** (which inputs to select in sub-instances) in parallel fuzzing. We use Figure 3 to illustrate how these two aspects influence the fuzzing efficiency on different programs.

Aspect 1: parallelism degree. First, *for different programs, the fuzzing efficiency varies across the number of parallelized instances.* Adding more CPU cores for fuzzing could increase the probability of finding solutions; however, if the solution space is narrow, newly found solutions could collide, reducing the overall efficiency. For example, in Figure 3a, the square and triangle find conflict solutions and are both synchronized with the circle instance, causing redundancy and thus affecting the overall efficiency. On the contrary, using more cores has positive effects if the solution space is large. For example, in Figure 3b, all four instances can find different solutions; therefore, the efficiency can be improved.

Limitations of existing efforts. All existing parallel fuzzers discussed in § 2.1 still assume a configuration that specifies the number of available CPU cores, and will exhaustively use all cores for fuzzing, which makes them unable to adapt to various programs. As illustrated in Figure 1, for the `libtiff` project, perfAFL [77] performs best with 30 cores, while μFuzz [16] reaches the best performance at 40 cores. However, for `libpng` project, μFuzz [16] performs best with 20 or 40 cores.

Aspect 2: input selection. Second, *the input selection method should vary across different programs.* Intuitively, if the solution space is wide, it is beneficial to perform more diversified searches in different directions. However, if the search space is narrow, the best strategy would be to intensify the search in the same direction for deeper solutions. For example, consider different solution



(a) Conflict solutions result in inefficiency. (b) Discrete solutions advocate more instances. (c) Narrow solution space needs intensification. (d) Wide solution space needs diversification.

Fig. 3. Dots with different shapes are instances. The dotted and solid lines are the search/synchronization.

spaces in Figure 3c and Figure 3d; a narrow but deep solution space needs intensification, while a wide solution space needs diversification.

Limitations of existing efforts. As discussed in § 2.1, naive parallel mode [1, 9, 26, 71, 76] runs multiple instances of the same fuzzer, which always "intensifies" the most promising inputs for mutation, therefore, all instances tend to focus on fuzzing a similar set of inputs. Approaches [28, 47, 62, 75] that feature principled task distribution strategies focus on maximizing search diversity in different instances. Those methods, while innovative, have two significant drawbacks. On the one hand, they primarily focus on reducing task conflict [28, 46], overlooking the potential benefits of repeated searching. On the other hand, they divide the program based on paths [28, 47] or functions [62], which could be either too fine-grained or coarse-grained.

Example 1. *Consider the control-flow graph (CFG) in Figure 4, if one input covers the path* $P_1 = \langle A, C, E, I, J, K, M \rangle$ *and the other one covers the path* $P_2 = \langle A, C, E, I, J, L, M \rangle$*, although* P_1 *and* P_2 *are highly related (only* K *and* L *are different), they are still marked as different tasks, which could result in possible solution conflicts in different instances because* P_1 *is easy to reach from* P_2 *and vice versa. However, if the program is partitioned based on functions, then inputs that cover any blocks inside the CFG will not be distinguishable.*

Since base fuzzers that have different properties and advantages can complement each other on different programs, collaborative fuzzing [13, 24, 29, 57] could achieve program adaption to some extent. However, existing collaborative fuzzers still suffer from two major drawbacks. First, they only realize a coarse-grained program-adaptive method since the number of available fuzzing strategies is limited by the number of adopted base fuzzers. Second, they still assume a fixed number of CPU cores and try to distribute resources among the base fuzzers, hence overlooking possible inefficiency caused by suboptimal parallelism degree.

We also studied the average contribution of instances to the global code coverage of fuzzing programs used in § 5.2 with AFL++ using 10, 20, 30, and 40 CPUs for 3 hours. Figure 2 shows the results, and we draw two conclusions. First, Using more CPUs increases the chance of solution conflicts and decreases global coverage. As shown in Figure 2, nearly all projects show a monotonic decrease trend of average contribution ratio when more CPUs are adopted. For example, in `libpng`, the average contribution ratios are 32.2%, 26%, 15.8%, and 15.6% when using 10, 20, 30 and 40 cores, respectively. Therefore, even using 30 cores achieves higher local coverage (edges covered by sub-instances), most of them conflict and do not contribute to global coverage, causing the decrease in global coverage from 20-core to 30-core in Figure 1a. Second, the probability of covering new program code increases when more CPUs are used, causing the see-sawing pattern. For example, in `libpng`, the average contribution ratio of 40-core is close to that of 30-core (15.6% vs. 15.8%), and 40-core has higher local coverage. Therefore, 40-core reaches higher global coverage in Figure 1a.

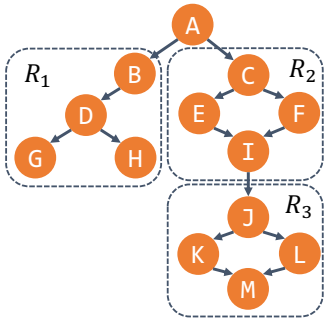


Fig. 4. An example CFG. R_n marks code regions.

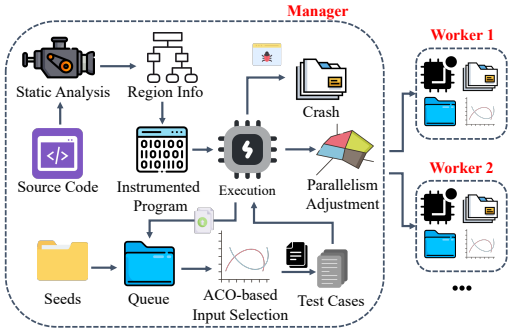


Fig. 5. Overview of KRAKEN

However, it is challenging to manually specify suitable parallel fuzzing strategies for different programs in advance because it is hard to statically determine the solution space, which could only be revealed during the fuzzing process.

2.3 Our Technique

Insight. Our key insight to solve limitations of existing efforts is that the inefficiency in parallel fuzzing can be observed during runtime through various feedbacks such as code coverage changes, which allows us to dynamically adjust the adopted strategy to avoid inefficient path searching, thus gradually approximating the optimal strategy.

For example, suppose the initial input selection strategy for the program in Figure 3c guides the search direction to be diverse. Since the solution space is narrow, the inefficiency can be observed in the search results, which can be improved by changing the search directions to be more intensified. However, there are two challenges for efficiently approximating the optimal strategy.

Challenge 1: How to efficiently evaluate the current strategy? Simply evaluating code coverage changes at a certain moment is not appropriate because the coverage increase is not at a constant speed throughout the fuzzing process. Therefore, a bad strategy could also result in a seeming coverage increase.

Challenge 2: How to efficiently adjust the fuzzing strategy to obtain the optimal one? The granularity and frequency of the adjustment could significantly affect the final results. For example, although Figure 3d needs search diversity, if search directions become so diverse that they go out of the solution space, the fuzzing results could even get worse.

This paper proposes KRAKEN, a new parallel fuzzer that achieves better program adaption by optimizing the two aspects of parallel fuzzing with two new algorithms.

Algorithm 1: dynamic parallelism adjustment (§ 3.2). We propose a dynamic parallelism adjustment algorithm for optimizing the number of CPU cores used. To tackle **challenge 1**, it describes the overall fuzzing process with a mathematical model and leverages the Bayesian method [33] to estimate its parameters with runtime statistical data. To tackle **challenge 2**, the algorithm leverages the estimation result to compute a single objective function and tries to maximize it with a simulated annealing-inspired process [8].

Algorithm 2: ACO-based dynamic input selection (§ 3.3). Our second algorithm aims to optimize the input selection. To tackle **challenge 1**, the algorithm measures how the search diversity affects the fuzzing outcomes of selected inputs. The algorithm partitions the program into regions, a compound hierarchical representation of the program [41]. For example, in Figure 4, the CFG can be split into R_1, R_2, and R_3 based on branches. By instrumenting the program, KRAKEN can obtain

region coverage information during runtime. Various feedbacks of fuzzed inputs, such as region and branch coverage, are aggregated to measure the outcomes of inputs. To tackle **challenge 2**, the algorithm adopts the ant colony optimization (ACO) [22] to adjust the input selection for maximizing code coverage.

3 Design

In this section, we first introduce Kraken's workflow (§ 3.1). We then detail our methods for dynamic parallelism adjustment (§ 3.2) and ACO-based input selection (§ 3.3).

3.1 Overview of Kraken

Figure 5 shows the workflow of Kraken. It takes the project code, initial seed inputs, and the maximum number of allowed computation cores as inputs.

Kraken contains two kinds of fuzzing instances: the manager and the worker node. Both of them implement essential components for grey-box fuzzing, such as input mutation, feedback collection, and crash saving. There is only one manager node, and all the remaining ones are worker nodes that can only be launched by the manager node. The manager node synchronizes inputs from all worker nodes, while workers only fetch inputs from the manager. The parallel fuzzing process of Kraken is mainly controlled by two components that leverage different optimization algorithms for searching optimal strategy: parallelism degree adjustment (§ 3.2) and ACO-based input selection (§ 3.3).

Those optimization algorithms are well-suited for program-adaptive strategy searching in parallel fuzzing because of their strengths in handling uncertainty, optimizing exploration-exploitation trade-offs, and leveraging data-driven learning [50, 59, 80].

Parallelism degree adjustment. Kraken only launches the manager node and one worker node at the beginning. During runtime, the manager node periodically collects statistical data from workers and iteratively updates a mathematical model that represents the fuzzing efficiency. Based on estimation results, the manager node leverages a simulated annealing-inspired algorithm to activate/deactivate workers to maximize an objective function.

ACO-based input selection. Apart from the regular coverage instrumentation, Kraken performs an additional region analysis and instruments the binaries to get region coverage at runtime to measure the path search diversity. Each node (manager and worker) contains a separate ACO-based decision engine and generates new inputs independently according to runtime feedback.

3.2 Parallelism Degree Adjustment

The first part of Kraken's technique aims to find the optimal parallelism degree for parallel fuzzing. Our key idea is to use the Bayesian method [33] to estimate the fuzzing efficiency with runtime statistical data and dynamically adjust the number of instances for maximizing the overall efficiency. The algorithm uses a mathematical model to describe the parallel fuzzing process and leverages the runtime feedback to estimate the efficiency under a specific parallelism degree. Then, it uses a simulated annealing-inspired process to dynamically activate/deactivate instances to maximize the efficiency. Therefore, Kraken could try different parallelism degrees during the fuzzing process and select the most promising one.

3.2.1 Efficiency Estimation. There exists a set of underlying parameters that control the output of the fuzzing process [7, 82]. For single-node fuzzing, each input only has two outcomes: finding new paths or examining old ones. Since parallel fuzzing could generate multiple inputs simultaneously, apart from the above two outcomes, inputs could also find new paths but collide with others.

We use a mathematical model to describe the parallel fuzzing process. Let p_{new}, p_{old}, and p_{coll} be the probabilities of one input finding new edges, not finding new edges, and finding new edges but colliding with other inputs. Then, the number of paths found by parallel fuzzer is assumed to be independently distributed and can be modeled by a *multinomial distribution* with parameter vector $\vec{p} = \{p_{new}, p_{old}, p_{coll}\}$ and parameter $K = 3$ as follows:

$$p(\vec{n}|\vec{p}, N) = \binom{N}{\vec{n}} \prod_{k=1}^{K} p_k^{n^{(k)}} \triangleq \text{Mult}(\vec{n}|\vec{p}, N) \tag{1}$$

with the multinomial coefficient $\binom{N}{\vec{n}} = \frac{N!}{\prod_k n^{(k)}!}$. Further, the elements of $\vec{p}$ and $\vec{n}$ follow the constraints $\sum_{k=1}^{K} p_k = 1$ and $\sum_{k=1}^{K} n^{(k)} = N$. The multinomial distribution governs the multivariate variable $\vec{n}$ with elements $n^{(k)}$, which counts the occurrences of the event k within N total trials. The multinomial coefficient counts the number of configurations of individual trials that lead to the total. For this model, if we can estimate $\vec{p}$, we can estimate the fuzzing process.

We employ the widely adopted Bayesian likelihood estimation [33] to estimate $\vec{p}$ from a set of observed data $\mathbb{D}$. Specifically, $\mathbb{D}$ is the set of occurrence counts of three outcomes during the parallel fuzzing process: $\mathbb{D} = \{d_{new}, d_{old}, d_{coll}\}$.

For the parameter $\vec{p}$ of the multinomial distribution, the conjugate prior is the *Dirichlet distribution*, which generalizes the beta distribution from 2 to K dimensions:

$$Dir(\vec{p}|\vec{\alpha}) \triangleq \frac{\Gamma(\sum_{k=1}^{K} \alpha_k)}{\prod_K \Gamma(\alpha_k)} \prod_{k=1}^{K} p_k^{\alpha_k - 1} \tag{2}$$

with parameters vector $\vec{\alpha}$ that controls the shape of the probability mass function, which can be estimated from the training set of count vectors $\mathbb{D}$. The likelihood is [33]:

$$p(D|\vec{\alpha}) = \prod_i p(x_i|\vec{\alpha}) = \prod_i \left(\frac{\Gamma(\sum_{k=1}^{K} \alpha_k)}{\Gamma(n_i + \sum_{k=1}^{K} \alpha_k)} \prod_k \frac{\Gamma(n_{ik} + \alpha_k)}{\Gamma(\alpha_k)} \right) \tag{3}$$

The gradient of the log-likelihood is [33]:

$$g_k = \frac{d \log p(D|\vec{\alpha})}{d\alpha_k} = \sum_i \Psi(\sum_k \alpha_k) - \Psi(n_i + \sum_k \alpha_k) + \Psi(n_{ik} + \alpha_k) - \Psi(\alpha_k) \tag{4}$$

Its maximum is obtained using the fixed-point iteration [53]:

$$\alpha_k^{new} = \alpha_k \frac{\sum_i \Psi(n_{ik} + \alpha_k) - \Psi(\alpha_k)}{\sum_i \Psi(n_i + \sum_k \alpha_k) - \Psi(\sum_k \alpha_k)} \tag{5}$$

The value of $\vec{p}$, which is the probability of finding new paths (p_{new}), covering old paths (p_{old}) and colliding with other new paths (p_{coll}), can be computed as [33]:

$$p_i = \frac{\alpha_i + d_i - 1}{\sum_{j=1}^{k} (\alpha_j + d_j - 1)} \tag{6}$$

where i corresponds to the three outcomes of each input, i.e., $i = \{new, old, coll\}$. α_i is from $\vec{\alpha}$ and d_i is from $\mathbb{D}$.

Example 2. *Initially, $\vec{\alpha} = \{1.0, 1.0, 1.0\}$. Suppose at time t, there are 10,000 inputs generated in total, and 5,000 of them trigger new code coverage while 1000 of them increase the global code coverage. Then, the observed data $\mathbb{D}$ is constructed as $\mathbb{D} = \{1,000, 4,000, 5,000\}$. Then, we could update $\vec{\alpha}$ with*

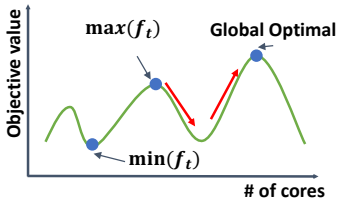


Fig. 6. An illustration of simulated annealing process.

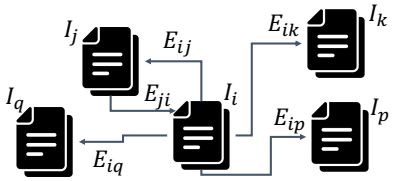


Fig. 7. Input selection as graph traversal.

$\mathbb{D}$ *using Equation 3 - Equation 5, and* $\vec{\alpha} = \{0.765361, 1.65205, 1.89488\}$. *Further applying Equation 6,* $\vec{p} = \{0.0999634, 0.400013, 0.500024\}$

3.2.2 Parallelism Adjustment. Algorithm 1 presents the parallelism adjustment process. At a high level, it uses a simulated annealing [8] inspired process to adjust the number of used CPUs to maximize the objective function f_t that represents the fuzzing efficiency.

Example 3. *Figure 6 illustrates the simulated annealing process, where the x-axis is the number of used cores and the y-axis is the value of the objective function. As illustrated in the figure, the algorithm can gradually approach the maximum value of the objective function* f_t. *At the same time, it prevents Kraken from falling into a local best result and allows it to find the global optimum result.*

More specifically, at the beginning, Kraken only launches the manager node and one worker node. To let Kraken make sufficient progress, the manager node invokes the `AddOrKillDecision` function inside Algorithm 1 for parallelism adjustment after each round of fuzzing on one input, i.e., mutating and running the input multiple times [9]. It first performs a round of synchronization with all workers to collect newly found inputs that can increase the global code coverage (Line 3). Moreover, the manager node counts the number of inputs that fall into the three outcomes: finding new coverage, covering old paths, and colliding with other inputs on new paths. Then, it constructs a set of observed values $\mathbb{D}$ for efficiency estimation (Line 4).

Line 7 - Line 10 illustrates the fixed-point computation for updating the parameters $\vec{\alpha}_c$ for the current number of running instances c. To ensure its convergence, we limit the maximum number of iterations to a configurable value *MaxIter* (20 in our implementation). If the total changes in parameter $\vec{\alpha}_c$ is less than the converge criteria ε (0.001 in our implementation), we also break the iteration. The value of $\vec{p}_c$, i.e., the probability of finding new paths (p_{new}), covering old paths (p_{old}) and colliding with other new paths (p_{coll}), can be computed with Equation 6 given $\vec{\alpha}_c$ (Line 12).

Leveraging the obtained $\vec{p}_c$, Kraken computes the objective function f_t to measure the "fitness" of the current strategy. The objective function f_t is designed such that the code coverage is maximized while the path collision is minimized and is defined as follows:

$$f_t = \frac{p^c_{new}(t) + 1}{p^c_{coll}(t) + 1} \times \sum_{i=0}^{c} avgSpeed_i(t) \tag{7}$$

where $p^c_{new}(t)$ ($p^c_{coll}(t)$) represents the probability of finding new paths (colliding with other inputs) for c CPUs at time t. At time t, the number of new paths found by the parallel fuzzer with c cores can be computed as $p_{new} \times \sum_{i=0}^{c} avgSpeed_i(t)$, which is the probability of finding new paths times the number of newly generated inputs at time t by all instances. Therefore, we add the two probability values by one to normalize them and further times their fraction with the sum of the fuzzing speed of all instances ($avgSpeed_i(t)$).

The parallelism manager dynamically adjusts the number of worker nodes by leveraging the result of the efficiency estimation discussed above. Specifically, at time t, suppose we have obtained

Algorithm 1: Dynamic Parallelism Adjustment

Input: *List of running worker nodes* $\mathbb{L}_w$*, and the number of running instances* c
Output : *Decision of adding or killing sub-instances*

```
1  Function AddOrKillDecision():
2      foreach w in L_w do                                   ▷ synchronize all workers
3          Sync(w)
4      D = {c_new, c_old, c_coll} ← getObs()                 ▷ runtime data
5      if α_c is empty then
6          α_c ← {1.0, 1.0, 1.0}
7      iter ← 0; α'_c ← α_c
8      while iter ≤ MaxIter and abs(Σ α_c − Σ α'_c) > ε do
9          α' ← α
10         DirMulEstimation(α_c, D)                          ▷ estimation
11         iter ← iter + 1
12     p_c = p_new, p_old, p_coll ← DirMulMode(α_c)          ▷ prob
13     V_{f_t} ← f_t(p_c, L_w)                               ▷ compute objective function
14     if V_{f_t} ≥ max(f_t) then
15         return ADD
16     else
17         q ← random(0, 1)
18         d ← max(f_t) − min(f_t)                           ▷ normalize
19         n ← max(max(f_t) − f_t, f_t − min(f_t))
20         if c < c_{max(f_t)} then
21             return q < n/d ? ADD : KILL
22         else
23             return q < n/d ? KILL : ADD
```

the probability vector $\vec{p}_c(t)$. Kraken uses a simulated annealing [8] inspired process to maximize f_t (Line 13 - Line 23 in Algorithm 1). If the current parallelism configuration c increases f_t, the manager node adds another worker. Otherwise, it adds or kills a worker based on probability proportional to the value of f_t. For example, the rule for adding a new instance is as follows:

$$add = \begin{cases} \dfrac{max(max(f_t) - f_t, f_t - min(f_t))}{max(f_t) - min(f_t)}, & \text{if } c < c_{max(f_t)} \\ \dfrac{min(max(f_t) - f_t, f_t - min(f_t))}{max(f_t) - min(f_t)}, & \text{otherwise} \end{cases}$$

where f_t is the current value of the objective function. $max(f_t)$ is the maximum value of f_t so far. c is the current number of used CPU cores. $c_{max(f_t)}$ is the number of CPU cores whose objective function is evaluated to $max(f_t)$.

As illustrated in Algorithm 1, if the current number of running CPU cores c is smaller than the number of CPU cores $c_{max(f_t)}$ that achieves the maximum f_t so far, Kraken gives more probability to adding another worker (Line 21). Otherwise, it gives more probability of killing an existing worker (Line 23). The above process allows us to gradually approach the global optimal value of f_t and c. When the manager decides to kill one instance, it kills the last one added. The basic

insight here is that the instance that runs with the shortest time is more likely to explore shallow paths, therefore, killing the last instance has the least chance of missing deeper new paths. When it decides to add a new worker, it starts a fresh instance with an input queue that is constructed from the manager input queue, thus preventing it from re-exploring already covered shallow paths.

3.3 ACO-based Input Selection

The second part of the design aims to optimize the input selection. The key idea is to measure the search diversity in sub-instances and evaluate how the current input selection strategy affects the coverage of selected inputs. By regulating the input selection with an effective meta-heuristic, the *Ant Colony Optimization* (ACO) [20], Kraken can gradually approach the optimal strategy. Unlike previous work, which merely maximizes diversification or intensification, our method lets sub-instances decide searching directions based on the code coverage results of already selected inputs. Therefore, the input selection purely depends on the actual runtime results instead of pre-defined rules.

3.3.1 Diversity Measurement. To measure the search diversity in sub-instances, existing work examines whether inputs cover different paths [28, 47] or functions [62]. However, as shown in § 2.2, existing measurement is ineffective since their program partition schema could be either too fine-grained or coarse-grained. Different from them, Kraken partitions each function in the target program into *single-entry single-exit* (SESE) regions [12, 41, 45], which form a hierarchical representation of the control structure of a program that allows us to exploit the sparsity of the flow graphs. We now formally define the SESE region:

Definition 1. *Let $G = (N, E, \rho)$ be a flow graph, where N is a set of nodes, $E \in N \times N$ is a set of directed edges, and ρ is the entry node. Given that $N_1 \subseteq N$, $E_1 \subseteq E$, and $\rho_1, e_1 \in N$, a connected subgraph, $R = (N_1, E_1, \rho_1)$, is called a SESE region of G with entry node ρ_1 and exit node e_1 if and only if on every path $(n_1, ..., n_k)$, where $n_1 = \rho$ and $n_k \in N_1$, there is some $i < j \leq k$ such that: $n_i = \rho_1$, $n_j = e_1$, $n_{i+1}, ..., n_j \in N_1$, and $(n_i, n_{i+1}), (n_{i+1}, n_{i+2}), ..., (n_{k-1}, n_k) \in E_1$.*

The above definition ensures that every path from the region entry node ρ_1 passes through its exit e_1; in other words, e_1 post-dominates ρ_1 while ρ_1 dominates e_1. Moreover, regions cannot have any partial overlap. If two regions have any nodes in common, they are either nested or in tandem. The sparsity of regions allows us to reason about searching diversification effectively by examining whether two inputs cover different regions.

Algorithm 2 presents the region partition algorithm. First, it uses a linear-time greedy algorithm [41] *getAllSESERegions* to partition the flow graph of $\mathbb{F}$ into a set of regions $\mathbb{R}$ (Line 2). However, the results are still not suitable for guiding parallel fuzzing and need further refinement. For instance, regions with only one block are not worth exploring, but it is still valuable to explore if the block contains other callees. Moreover, if two sequential regions are only connected by one edge, then distinguishing them would not be necessary.

Example 4. *In Figure 4, there is no need to distinguish R_2 and R_3 because R_2 dominates R_3. So, we further refine the result $\mathbb{R}$ into $\mathbb{R}'$ to eliminate unnecessary regions.*

Specifically, the refinement step skips basic blocks that lack path divergence such as singe successor (Line 7) and "if" statement without "else" branches (Line 10). Then, for each $\mathbb{B}$'s successor that is the entry of a region R_{new}, the algorithm tries to iteratively expand R_{new} by joining this region with its predecessors (Line 15). Newly found regions are added to the results set $\mathbb{R}'$ if it is "valuable", i.e., whether it contains sufficient code for exploring. If blocks inside R_{new} contain at least one callee, we view it as valuable; otherwise, if the number of blocks is less than 3 (cannot form a simple flow graph with branches), we discard it (Line 17).

Algorithm 2: Region Analysis

```
Input: Function 𝔽
Output : The result of region partition ℝ
Function RegionPartition(𝔽):
    ℝ ← getAllSESERegions(𝔽)                          ▷ initial region partition
    ℝ′ ← ∅
    foreach 𝔹 in 𝔽 do
        𝒮_𝔹 ← Succ(𝔹)                                  ▷ successors of basic blocks
        if |𝒮_𝔹| == 1 then
            continue
        else if |𝒮_𝔹| == 2 then
            if postDominate(𝒮_𝔹[1], 𝒮_𝔹[0]) or postDominate(𝒮_𝔹[0], 𝒮_𝔹[1]) then
                continue
        foreach 𝔹 in 𝒮_𝔹 do
            R_new ← getRegionFor(𝔹, ℝ)
            E ← getEntry(R)
            if 𝔹 == E then
                R_new ← expandRegion(R_new)
                if isValuableRegion(R_new) then
                    ℝ′ ← ℝ′ ∪ R_new
    return ℝ′
```

Based on region analysis results, Kraken instruments the program binaries to give each region a unique ID so that they can be identified during the fuzzing process. Since the entry block ρ_1 of a region R dominates all basic blocks inside R, we only instrument ρ_1 to avoid redundant tracing. During the execution, if ρ_1 is covered, we record it in a shared bitmap to notify the fuzzer that R has been covered.

3.3.2 ACO-based input selection. Single-node fuzzing usually prioritizes ("intensifies") the so far most promising input for mutation. However, parallel fuzzing also needs to take care of possible path collisions. The main challenge here is how to prioritize inputs for fuzzing to balance intensification and diversification to improve the overall efficiency (as illustrated in § 2.2).

Our key idea is to view the problem of input selection as a combinational optimization problem [35, 67, 81] and solve it optimally by existing meta-heuristics, which can be used to guide local search algorithms towards promising regions of the search space containing high-quality solutions.

We leverage an effective meta-heuristic, the *Ant Colony Optimization* (ACO) [20] to regulate the input selection. Specifically, we model the input selection as a graph traversal problem by making each input an individual node and organizing them with a fully connected graph G. The graph is symmetric, meaning that each input is connected from and to all the others. Then, selecting the next input from the current one can be viewed as moving on the edge between them. Each edge is annotated with a value that controls the probability of choosing it from all the other neighbors.

Example 5. *In Figure 7, choosing the next input I_j from I_i can be viewed as moving on the edge E_{ij}.*

The edge E_{ij} is associated with a value $a_{ij}(t)$ that controls the probability of choosing E_{ij} from all the neighbor edges of I_i, which is defined as follows [21]:

$$a_{ij}(t) = [\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta} \tag{8}$$

where $\tau_{ij}(t)$ is the amount of the pheromone value on E_{ij} at time t, η_{ij} is the heuristic value that denotes the benefit of choosing input I_j. α and β are two parameters controlling the relative weight of the pheromone and heuristic values. In our current implementation, we let both of them be 1.

We set all incoming edges of input I to have the same pheromone value, denoting that the probability of choosing I from any other inputs is the same, i.e., $\forall j, k, a_{ij}(t) = a_{ik}(t)$.

We define the value η_{ij} to reflect the favorableness of input I_j itself. Fuzzing favors inputs with faster execution speed and smaller size [1], and we define η_{ij} as follows:

$$\eta_{ij} = \frac{max(factor) - factor}{max(factor) - min(factor)} \tag{9}$$

where $factor = T(I_j) \times Len(I_j)$, T_j and L_j are the execution time and the size of input I_j. We use the max-min normalization [25] to obtain η_{ij}.

The parameter $\tau_{ij}(t)$ represents the learned desirability of choosing I_j from I_i and is updated dynamically during the fuzzing process. The update is performed in two steps by applying the rule:

$$\tau_{ij}(t) \leftarrow (1-\rho)\tau_{ij}(t) + \rho\Delta\tau_{ij}(t) \tag{10}$$

where $\rho \in (0, 1]$ is a parameter governing pheromone decay. It helps to forget previously poor decisions and prevent Kraken from falling into the local minimum.

$\Delta\tau_{ij}(t)$ is the reward factor that depends on how well the chosen input I_j has performed. The definition of $\Delta\tau_{ij}(t)$ consists of two parts. First, we consider the number of new solutions found by mutating I_j (denoted as $Child(I_j)$). Second, we consider the unique region coverage achieved by I_j and $Child(I_j)$. Specifically, for input I_j in a fuzzing node F_n, we retrieve the region coverage of I_j and the overall region coverage of all other nodes except F_n. We then compute the number of regions uniquely covered by I_j, denoted as RC_j. We aggregate above definitions into the $factor$ defined as follows:

$$factor = |Child(I_j)| + RC_{I_j} + \sum_{n}^{i=0} RC_{Child(I_j)_i} \tag{11}$$

, and we normalize $factor$ to compute $\Delta\tau_{ij}(t) = \frac{factor}{factor+1}$. Using $\Delta\tau_{ij}(t)$ allows Kraken to adaptively select inputs to balance diversification and intensification to maximize the overall code coverage. If intensification brings more code coverage, Kraken leans to intensify similar inputs; otherwise, Kraken tends to perform more diversified path searching.

Input selection rule. We use the *pseudo random proportional* [22] rule to make the next move from an already chosen input I_i. Let q be a random variable uniformly distributed over $[0, 1]$, and $q_0 \in [0, 1]$ be a tunable parameter. Then the probability of choosing the next node I_j at time t is the following if $q \leq q_0$:

$$p_{ij}(t) = \begin{cases} 1 & \text{if } j = argmax\ a_{ij} \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

otherwise, when $q > q_0$:

$$p_{ij}(t) = \frac{a_{ij}}{\sum_{l \in \mathcal{N}_i} a_{il}(t)} \quad \forall j \in \mathcal{N}_i \tag{13}$$

Table 1. Magma programs.

Program	Version	Command-line
libpng	1.6.38	libpng_read_fuzzer
libsndfile	1.2.0	sndfile_fuzzer @@
libtiff	4.1.0	tiff_read_rgba_fuzzer -M @@
libxml2	2.9.10	xmllint @@
lua	5.4.0	lua
openssl	3.0.0	asn1parse
php	8.0.0	php-fuzz-exif
poppler	0.88.0	pdf_fuzzer
sqlite	3.32.0	sqlite3_fuzz

Table 2. Real-world programs.

Program	Version	Command-line
harfbuzz	2.7.2	hb-subset-fuzzer @@
gpac	20200801	MP4Box -stdb -diso @@
dav1d	0.7.1	dav1d -o /dev/null –demuxer ivf –muxer yuv -i @@
bento4	1.6.0	mp4info –show-layout @@
faad2	2.10.0	faad -w -b 5 @@
faust	2.30.5	faust -lang ocpp -e -lcc -exp10 -lb -rb -mem -sd @@
jasper	2.0.20	jasper –output-format pnm –input @@
gravity	0.8.1	gravity @@
libjpeg	2020021	jpeg -oz -h -s 1x1,2x2,2x2 @@ /dev/null
nasm	2.15	nasm -fmacho64 -g -o /dev/null @@

This decision rule has a double function: when $q \leq q_0$, the decision rule exploits the knowledge available about the problem by choosing the best local solution. When $q > q_0$, it operates a biased exploration according to Equation 13. Tuning q_0 allows us to modulate the degree of exploration and choose whether to concentrate the system's activity on the best solutions or to explore the search space. In our implementation, we let $q_0 = 0.5$ to give them equal chances.

Example 6. *In Figure 7, if the current input is I_i, and it has four neighbors: I_j, I_k, I_p and I_q. Each edge that connects from I_i to one of its neighbors is annotated with a value computed by Equation 8. Suppose the values of E_{ij}, E_{ik}, E_{ip}, and E_{iq} are 1, 2, 3, and 4, respectively. Then, at time t, if $q \leq q_0$, Kraken will choose I_q as the next input; otherwise, the probability of choosing I_j, I_k, I_p, and I_q are 0.1, 0.2, 0.3, and 0.4, respectively.*

4 Implementation

Kraken is implemented in C/C++ with over 10K lines of code and its overall architecture is similar to AFL [1]. Kraken implements most of the AFL's features such as input mutation, forkserver executor, coverage instrumentation, etc. Similar to existing parallel fuzzers, Kraken runs multiple fuzzer instances with individual processes simultaneously on multi-core machines. There exists only one manager node, which is responsible for invoking multiple worker nodes as described in § 3.2. To start a new worker instance, the manager instance calls `fork()` to create a new process and then calls `execve()` to run the fuzzing command. The manager will record the `PID` of all the involved workers and call `kill()` to stop specific workers on demand. Our implementation replaces the original seed prioritization heuristics employed by AFL with the ACO-based input selection algorithm described in § 3.3. Kraken implements the region analysis and instrumentation based on the `RegionInfoPass` [5] provided by the LLVM [43].

5 Evaluation

Our evaluation aims to answer the following questions.

- **RQ1**: Can Kraken achieve higher code coverage than than existing parallel fuzzers?
- **RQ2**: How is the bug detection capability of Kraken?
- **RQ3**: What are the contributions of Kraken's main components?

RQ1 assesses the coverage performance of Kraken, which is the main metric for evaluating a fuzzer [32]. We answer **RQ2** to show that Kraken is practical for real-world bug detection. **RQ3** aims at a finer evaluation of how the parallelism adjustment and ACO-based input selection help improve Kraken's effectiveness and show the importance of dynamic fuzzing strategy optimization.

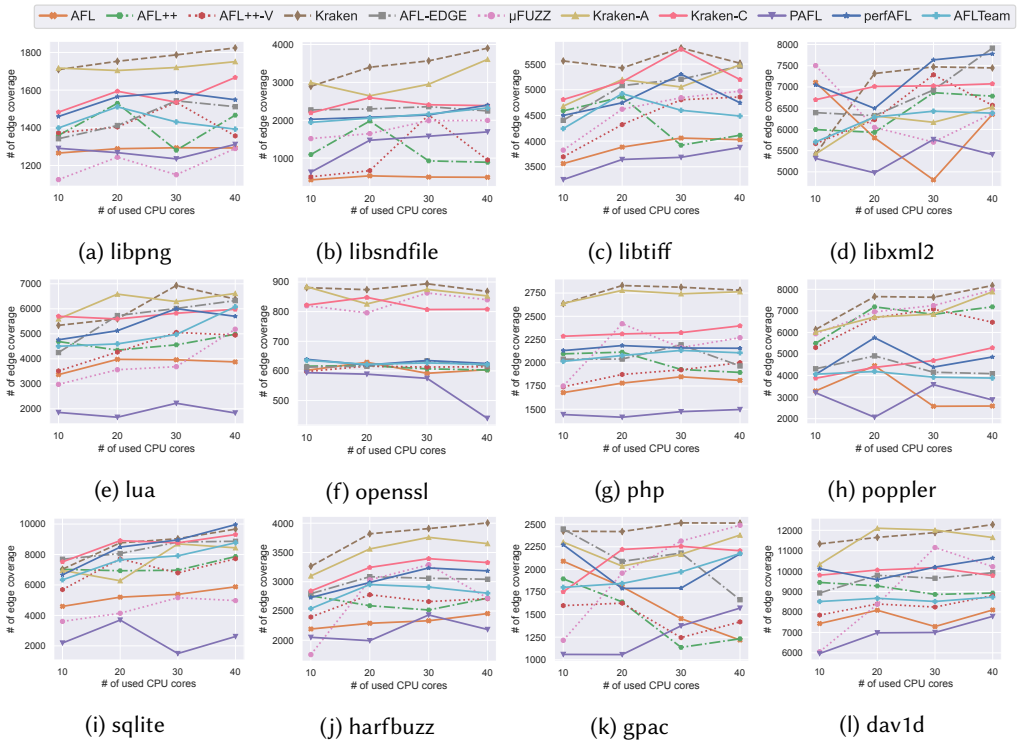


Fig. 8. Accumulated edge coverage by fuzzing with different number of CPU cores for 3 hours. AFL++-V uses different power schedule strategies in sub-instances. Kraken-A is Kraken without ACO-based input selection. Kraken-C is Kraken without parallelism degree adjustment.

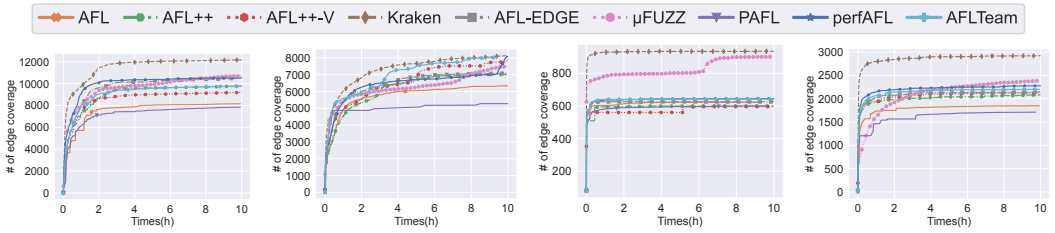
5.1 Experimental Setup

Environment Setup. We evaluate Kraken on an Intel Xeon(R) computer (80 cores) with an E5-1620 v3 CPU and 64GB of memory running Ubuntu 16.04 LTS.

Baseline Approaches. We compared Kraken with 8 state-of-the-art parallel fuzzers:

- AFL [1] is a well-known fuzzer with industrial strength. We follow its official document [2] to set up parallel fuzzing by running only one main node and numerous secondary nodes.
- AFL++ [3] is a re-engineered fork of AFL and is actively maintained. By default, it also runs parallel fuzzing with one main node and multiple secondary nodes.
- As suggested by AFL++'s official document [4], we also configure a variant of AFL++ that uses different power scheduling methods [9] in secondary nodes, denoted as AFL++-V.
- perfAFL [77] is built up on AFL with newly designed system primitives.
- PAFL [47] modifies the original AFL by dividing the bitmap into intervals and makes different fuzzer instances only focus on specific areas.
- AFL-EDGE [75] distributes mutually-exclusive but similarly-weighted tasks to sub-instances.
- AFLTeam [62] also divides the parallel fuzzing task into instances, it differs from AFL-EDGE in that its program partition algorithm is based on the call graph.
- μFuzz [16] is a most recent work that redesigns parallel fuzzing with microservice architecture.

Apart from existing parallel fuzzers, we also compare Kraken with two of its ablations to study the contribution of Kraken's key components:



(a) dav1d (b) libxml2 (c) openssl (d) php

Fig. 9. Edge coverage changes in 10 hours using 20 cores.

- Kraken-A is a variant of Kraken that adopts the single-node input selection strategy in all sub-instances, similar to the naive parallel mode [1, 9, 26, 71, 76] discussed in § 2.1.
- Kraken-C is a variant of Kraken that does not perform parallelism degree adjustment, i.e., it invokes all fuzzing instances at the beginning, similar to existing work [28, 29, 47, 75, 77, 83].

Benchmarks. We evaluate Kraken with both standard benchmarks and real-world programs. We choose programs based on the following features: popularity in the community, frequency of being used in the fuzzing evaluation, functionality diversity, and size of the codebase. For standard benchmarks, we adopt all 9 programs in the popular fuzzing benchmark Magma [32], which has been widely used by previous work [16, 24, 37, 39]. We also include 10 real-world programs that are popular in the community and frequently used in fuzzing evaluation [11, 31, 44, 52, 56]. The detailed versions and command line options are shown in Table 1 and Table 2. Programs whose command-line options contain "@@" are general-purpose applications that read inputs from files, while those that do not use "@@" are drivers specifically designed as fuzzer harnesses [32] and read inputs from the `stdin`. In total, we use 19 programs in the evaluation.

Configurations. To evaluate the code coverage, we run each fuzzer with 10, 20, 30, and 40 cores for 3 hours (wall time, which is the real-world time elapsed) using the same initial inputs, which equals 30, 60, 90, and 120 hours of single-node fuzzing campaign. We use a maximum of 40 cores, considering both the hardware limit and the duration of the experiments. Although previous work usually uses 24 hours as the time budget, we found that using multiple CPU cores could significantly reduce such 24-hour requirements. Following prior work [62], we also include the results of running compared fuzzers with 20 cores for 10 hours to support our claim. For the bug detection experiments, we let each fuzzer run for 4 hours with 20 cores. We repeat each experiment 5 times and plot the median value in the charts to mitigate the impact of randomness.

Metrics. We follow many prior studies [11, 38, 66] by using the edge coverage as the code coverage. We also measure the widely adopted metrics [9, 50], i.e., the number of unique crashes, to compare the performance of each fuzzer. We filter duplicate crashes by manually analyzing whether the stack trace provided by AddressSanitizer [65] is unique.

5.2 RQ1: Code Coverage

In this section, we compare Kraken with 8 state-of-the-art parallel fuzzers in terms of code coverage. We run each fuzzer for 3 hours using different numbers of CPU cores (10, 20, 30, and 40) with the same initial seed corpus and compare their achieved edge coverage. Figure 8 shows the accumulated edge coverage results on 12 benchmark programs. The results for Kraken's two ablations will be discussed in § 5.4. In total, Kraken covers 54.7% more edges than all baseline fuzzers. Specifically, Kraken achieves 75.9%, 29.5%, 32.8%, 34.2%, 36.3%, 158.9%, 29.5%, and 40.3% higher code coverage

Table 3. Bug detection results by running 4 hours with 20 cores. #Crash represents the number of unique crashes found by each fuzzer. #CVE shows the number of CVEs contained in the unique crashes.

Program	Kraken		AFL		AFL++		AFL++-V		perfAFL		AFL-EDGE		PAFL		AFLTeam	
	#Crash	#CVE	#Crash	#CVE	#Crash	#CVE	#Crash	#CVE	#Crash	#CVE	#Crash	#CVE	#Crash	#CVE	#Crash	#CVE
bento4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	**3**	**3**
faad2	**6**	**4**	2	1	5	3	**6**	**4**	4	**4**	5	**4**	0	0	4	**4**
faust	2	0	1	0	0	0	2	0	2	0	2	0	**3**	0	1	0
gpac	**2**	**2**	1	1	0	0	0	0	1	1	0	0	1	1	0	0
jasper	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
gravity	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
libjpeg	8	**2**	5	1	8	1	**10**	2	7	2	8	2	4	1	7	2
nasm	**10**	0	4	0	4	0	2	0	4	0	3	0	3	0	3	0
Total	**30**	**10**	15	5	19	6	22	8	20	9	20	8	13	4	18	9

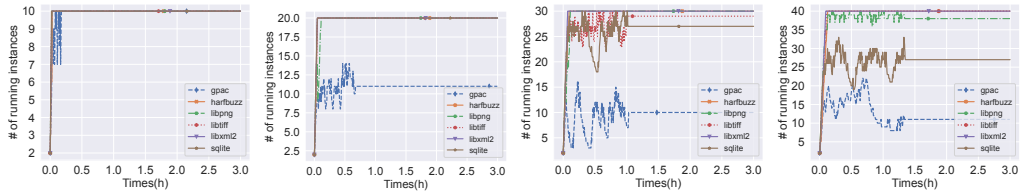
than AFL [1], AFL++ [3], AFL++-V, AFL-EDGE [75], μFuzz [16], PAFL [47], perfAFL [77], and AFLTeam [62] respectively on average. Kraken demonstrates superior performance across projects since it is able to adjust the fuzzing strategy according to programs adaptively.

Parallel fuzzers such as AFL, AFL++, and perfAFL adopt the single-node fuzzing strategy in all instances. As shown in the figure, they all show unstable performance across programs. For example, AFL++ outperforms AFL on `poppler` while AFL achieves higher edge coverage than AFL++ on `gpac`. Several fuzzers, such as AFL-EDGE and PAFL, focus on maximizing the search diversity in sub-instances. However, such a strategy does not fit all programs. For example, AFL-EDGE performs better than all the other baselines when fuzzing `lua`; however, it performs worse than AFL++ and μFuzz on `poppler`. μFuzz redesigns parallel fuzzing with micro-service architecture to improve the fuzzing efficiency. However, its input selection strategy is still based on the single-node fuzzing ones and is purely static. Therefore, it only outperforms other baselines on several projects such as `poppler` and `php`. On `sqlite`, μFuzz only performs better than PAFL. Although AFL++-V adopts different power schedule strategies in instances, the strategies remain unchanged throughout the fuzzing process. Therefore, it still suffers from the same limitations as other baselines.

Moreover, we observe that the performance of Kraken increases nearly linearly with the number of maximum CPU cores adopted, and different initial configurations do not affect the final results too much. Such results demonstrate the benefit of the program-adaptive technique, i.e., the fuzzing strategy can truly adapt to different programs. For other baselines, the performance could vary between configurations. For example, several fuzzers achieve monotonic performance increase on `sqlite`. The coverage achieved by μFuzz on `php` reaches a maximum of 20 cores but drops with more CPU cores. The coverage achieved by AFL-EDGE and AFL on `gpac` even decreases monotonically with the used CPUs.

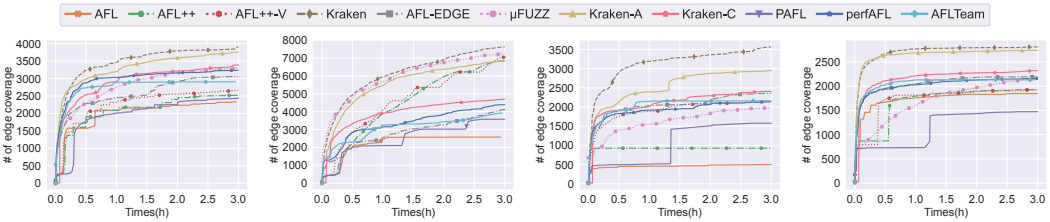
Although our method outperforms other fuzzers on most of the evaluated programs, it is still possible for Kraken to be defeated by others; for example, it does not achieve the best results on `libxml2`. As discussed in § 1, since determining the real "optimal strategy" is an undecidable problem, Kraken actually tries to find the best solution by evaluating the objective functions.

10-hours evaluation. We also evaluate Kraken against its baselines in terms of longer-term experiments. Following prior work [62], we let each fuzzer run with 20 cores for 10 hours and compare the final edge coverage, and the results are shown in Figure 9 (complete data is shown in [6]). In total, Kraken covers 43.9% more edges than other baselines. Specifically, Kraken achieves 96.9%, 35.2%, 35.2%, 26.8%, 6.3%, 105.2%, 19.7%, and 26.6% higher code coverage than AFL [1], AFL++ [3], AFL++-V, AFL-EDGE [75], μFuzz [16], PAFL [47], perfAFL [77], and AFLTeam [62] respectively on average. The results also support our claim that it is feasible to evaluate parallel fuzzing with a shorter time budget (§ 5.1) because the edge coverage nearly saturates after 3 hours. On average, the edge coverage only increases about 15% during the fuzzing campaign between 3 to 10 hours.



(a) 10 cores. (b) 20 cores. (c) 30 cores. (d) 40 cores.

Fig. 10. Parallelism degree changes within 3 hours under different number of maximum allowed CPUs.



(a) harfbuzz (b) poppler (c) libsndfile (d) php

Fig. 11. Edge coverage changes in 3 hours using 30 cores.

As shown in Figure 9a, the edge coverage achieved by all fuzzers on `dav1d` only increases about 4.7% on average after 3 hours. For `openssl`, the edge coverage even remains unchanged for most baselines after 3 hours.

5.3 RQ2: Bug Detection

Kraken has found 192 new vulnerabilities in 37 real-world projects, and 119 of them are assigned with CVE IDs[1]. Target projects contain widely tested ones such as `FFmpeg`, system libraries like `Ncurses`, and popular third-party libraries, including `libjpeg`.

In this section, we compare Kraken with baseline fuzzers in terms of their bug detection ability. We omit the result of μFuzz here since it does not support saving crash inputs to the disk, and we are unable to examine crashes manually. We adopt 8 popular real-world programs for the experiment. The detailed versions and command line options are shown in Table 2. Kraken has found numerous bugs in those programs, and some of them are assigned with CVE IDs. To avoid possible duplicate crashes, we manually examine all crash inputs found by each fuzzer and keep those that have unique stack traces. Table 3 shows the results of bug detection by running each fuzzers with 20 cores for 4 hours. As shown in the Table, Kraken finds 30 unique crashes in 4 hours, while AFL, AFL++, AFL++-V, perfAFL, AFL-EDGE, PAFL, and AFLTeam find only 15, 19, 22, 20, 20, 13, and 18 crashes, respectively. Among all the identified crashes, Kraken finds 10 CVEs, while AFL, AFL++, AFL++-V, perfAFL, AFL-EDGE, PAFL, and AFLTeam find only 5, 6, 8, 9, 8, 4, and 9 CVEs, respectively. Moreover, most baselines can only find the subset of CVEs found by Kraken, except for AFLTeam, which finds one more CVE than Kraken on `bento4`. On average, Kraken can find 70.2% more crashes and 103.2% more CVEs than other baselines.

[1]Details are shown in [6].

5.4 RQ3: Ablation Study

In this section, we study the contribution of Kraken's key components to the final fuzzing result.

Parallelism adjustment. To understand the contribution of parallelism adjustment, we configure Kraken to invoke all fuzzing instances at the beginning, denoted as Kraken-C. As shown in Figure 8, Kraken achieves 21% more edge coverage than Kraken-C on average, showing that the parallelism adjustment algorithm lets Kraken adapt to different programs effectively.

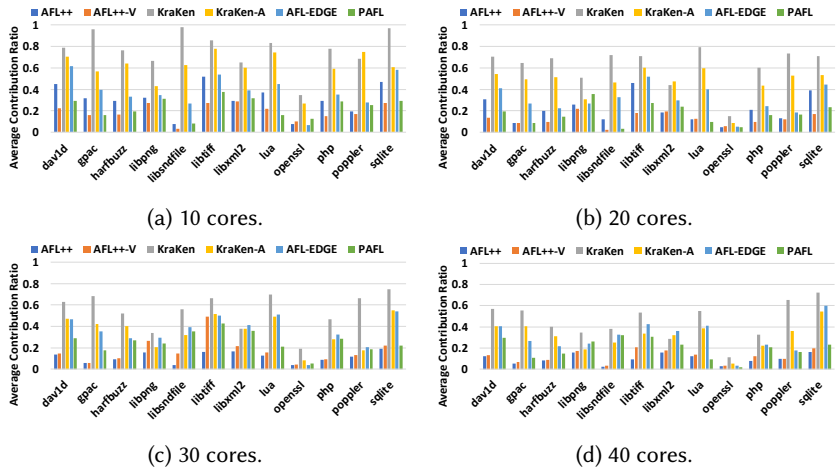
We further analyze the change in the number of running instances during fuzzing programs used in § 5.2. Due to space limitation, we only show the results for 6 programs in Figure 10, grouped by the maximum CPUs allowed (complete data is in [6]). We draw two main conclusions from the results.

First, the algorithm is able to adaptively optimize the parallelism degree to achieve better performance throughout the whole fuzzing process consistently. As shown in Figure 10, while Kraken uses all available CPUs for programs under the 10-core configuration (Figure 10a), with more available CPUs, the figure shows evident simulated annealing processes as the algorithm adjusts the number of CPUs according to the estimated efficiency. For example, Figure 10 shows that the optimal CPU number for fuzzing `gpac` is around 10 to 11. Although Kraken does not take up all the CPU resources when testing `gpac`, it still achieves higher code coverage than others (Figure 8k). For `sqlite`, Kraken estimates that the optimal number of cores is around 25. Therefore, it uses the maximum number of CPUs for fuzzing under the 10- and 20-core configurations (Figure 10a and Figure 10b), while using the same number (27) of CPUs under the 10- and 20-core configurations (Figure 10c and Figure 10d). This phenomenon again demonstrates the benefit of the program-adaptive technique, i.e., the fuzzing strategy can truly adapt to different programs, which is the key to solving the unstable performance issues discussed in § 2.2.

Second, the simulated annealing process described in § 3.2.2 is able to converge quickly, and the time spent for convergence is proportional to the maximum number of allowed CPUs. Intuitively, a larger number of CPUs means a larger search space for the algorithm, which requires more time to explore and find the global optimal solution. Our experimental results in Figure 10 show that the algorithm only takes about 30 minutes more to converge when there are ten more available CPUs, which is practical enough for real-world usage. Take the results for `gpac` as an example. Under the 20-core configuration, the algorithm converges with less than 0.7 hours (Figure 10b). The time spent on finding the optimal solution under the 30-core configuration is about one hour (Figure 10c), and the time for 40 cores is about 1.5 hours (Figure 10d). Moreover, the algorithm is performing the hill-climbing process at first because increasing the core number at the beginning has a more positive effect on the fuzzing process. Then, it explores around the global optimal solution to prevent falling into a local minimum. After all possible core numbers are tested, the optimal core number shows obvious superiority compared with others, so the number of cores does not change. As shown in Figure 10, under all configurations, the core number increases very quickly at the beginning and slows down afterward, which fits the illustration in Figure 6.

We also measured the runtime overhead of the parallelism adjustment. Specifically, we counted the total time spent on adding/killing instances and efficiency estimation under the 40-core configuration. Within 3 hours, 24 seconds are spent on parallelism adjustment on average, which only accounts for 0.22% of the total running time.

ACO-based input selection. To understand the contribution of the ACO-based input selection, we configure Kraken to adopt the single-node input selection strategy in all sub-instances, denoted as Kraken-A. As shown in Figure 8, Kraken achieves 6.2% more edge coverage than Kraken-A on average, showing that our algorithm could adapt to different programs effectively. Figure 11 shows the edge coverage changes within 3 hours using 30 CPU cores on 4 programs used in



(a) 10 cores.

(b) 20 cores.

(c) 30 cores.

(d) 40 cores.

Fig. 12. Average contribution ratio by each instance of fuzzing programs used in § 5.2.

Figure 8 (complete data is shown in [6]). As shown in the figure, at the beginning, the edge coverage increases similarly between different fuzzers. However, without the ACO-based input selection, the coverage increase gets slower with time, and the final coverage achieved by KRAKEN-A is still lower than KRAKEN.

We further studied the contribution of each instance to the global code coverage to demonstrate how the input selection algorithm reduces solution overlaps. Specifically, we adopt 6 fuzzers (AFL++, AFL++-V, PAFL, AFL-EDGE, KRAKEN-A, and KRAKEN) that perform input synchronization in a centralized way, i.e., there exists only one instance that fetches inputs from all the other ones while the remaining instances only fetch inputs from the central node. This allows us to reason about whether the solutions found by each instance could increase the global code coverage. Figure 12 shows the results of the average contribution ratio of each instance of fuzzing programs used in § 5.2 with 10, 20, 30, and 40 CPU cores for 3 hours. The y-axis represents the average portion of inputs that increase the global coverage in all instances.

The figure shows that KRAKEN achieves an average contribution ratio of 59.7%, outperforming AFL++, AFL++-V, AFL-EDGE, PAFL, and KRAKEN-A by 41.5%, 44.4%, 25.8%, 38.2%, and 16%, respectively. AFL-EDGE achieves a higher average ratio than AFL++ (33.8% vs. 18.1%) because it focuses on maximizing the search diversity in sub-instances by splitting the program based on paths. Although PAFL also tries to improve search diversity, its diversity measurement is purely based on the bitmap, which is much less effective than that of AFL-EDGE. Therefore, PAFL performs the worst among compared fuzzers, with an average contribution ratio of 21.4%. Thanks to the parallelism adjustment algorithm, both KRAKEN-A and KRAKEN avoid much more redundancy than all the other compared fuzzers. KRAKEN further outperforms KRAKEN-A by 16% on average because the ACO-based input selection could better balance search diversification and intensification.

In terms of runtime overhead, within 3 hours, 28 seconds are spent on ACO-based input selection on average, which only accounts for 0.26% of the total running time.

6 Related Work

Apart from existing parallel fuzzers discussed in § 2.1, this section surveys other related work. They all focus on improving the efficiency of single-node fuzzing, which is orthogonal to our work and can be integrated with KRAKEN for better performance.

Controlled input generation. Existing fuzzers adopt different approaches to generate high-quality inputs for testing the target programs. They could utilize existing grammar specifications [30, 58] or file format information [60] to generate inputs that are both syntactically and semantically correct. Some methods leverage dynamic taint analysis [11, 63, 73] or lightweight mutation-based taint inference [23, 40, 48, 61, 78] to track which input parts affect program instructions for guiding input mutation. Some studies [18, 36] make use of sophisticated program static analysis techniques [68, 69] to generate useful test cases. Other fuzzers [14, 17, 38, 79] use symbolic execution or concolic execution to generate inputs that can get through complex program conditions.
Input scheduling. Existing work also tries to improve the input scheduling algorithm to facilitate fuzzing. Some approaches [26, 51] design more fine-grained fitness functions to evaluate and schedule inputs. Some methods [9, 27, 50, 74] leverage various optimization algorithms or heuristics to prioritize promising inputs for increasing code coverage.
Instrumentation optimization. Several static binary instrumentation methods [19, 55, 80] are proposed to perform binary-only coverage-guided fuzzing. Other approaches [34, 54, 72, 82] reduce instrumentation costs by selective or on-demand coverage instrumentation.

7 Conclusion

We propose a new program-adaptive parallel fuzzer Kraken, which achieves higher code coverage and detects more bugs than existing work. It also detects hundreds of real-world bugs.

Data Availability

We will open source Kraken at https://github.com/seviezhou/Kraken. We will also include the raw data of evaluations in § 5.2, § 5.3 and § 5.4, including a pdf format supplemental material.

Acknowledgments

We thank the anonymous reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. This work is funded by research donations from Huawei, TCL, and Tencent.

References

[1] 2017. AFL. http://lcamtuf.coredump.cx/afl/. [Online; accessed 29-January-2022].
[2] 2017. AFL. https://github.com/google/AFL/blob/master/docs/parallel_fuzzing.txt. [Online; accessed 29-January-2022].
[3] 2019. AFLplusplus Documents. https://github.com/AFLplusplus/AFLplusplus/. [Online; accessed 29-January-2023].
[4] 2019. AFLplusplus Documents on Using Multiple Cores. https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing_in_depth.md. [Online; accessed 29-January-2023].
[5] 2024. RegionInfoPass. https://llvm.org/doxygen/classllvm_1_1RegionInfoPass.html.
[6] 2025. Supplement Material. https://github.com/seviezhou/Kraken/blob/main/supplement/supplement-material.pdf.
[7] Marcel Böhme and Brandon Falk. 2020. 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*. 713–724.
[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.
[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.
[10] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. Jigsaw: Efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 18–35.
[11] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.
[12] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. 2003. Code compaction of matching single-entry multiple-exit regions. In *International Static Analysis Symposium*. Springer, 401–417.

[13] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers.. In *USENIX Security Symposium*. 1967–1983.

[14] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596.

[15] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 633–645.

[16] Yongheng Chen, Rui Zhong, Yupeng Yang, Hong Hu, Dinghao Wu, and Wenke Lee. 2023. μFuzz: Redesign of Parallel Fuzzing Using Microservice Architecture. In *USENIX Security Symposium*.

[17] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-level constraint solving for hybrid fuzzing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 515–530.

[18] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 677–693.

[19] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.

[20] Marco Dorigo and Christian Blum. 2005. Ant colony optimization theory: A survey. *Theoretical computer science* 344, 2-3 (2005), 243–278.

[21] Marco Dorigo, Gianni Di Caro, and Luca M Gambardella. 1999. Ant algorithms for discrete optimization. *Artificial life* 5, 2 (1999), 137–172.

[22] Marco Dorigo and Luca Maria Gambardella. 1997. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation* 1, 1 (1997), 53–66.

[23] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13.

[24] Yu-Fu Fu, Jaehyuk Lee, and Taesoo Kim. 2023. autofz: Automated Fuzzer Composition at Runtime. *arXiv preprint arXiv:2302.12879* (2023).

[25] Vatsal Gajera, Rishabh Gupta, Prasanta K Jana, et al. 2016. An effective multi-objective task scheduling algorithm using min-max normalization in cloud computing. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*. IEEE, 812–816.

[26] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[27] Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2021. Cerebro: Static Subsuming Mutant Selection. *arXiv preprint arXiv:2112.14151* (2021).

[28] Taotao Gu, Xiang Li, Shuaibing Lu, Jianwen Tian, Yuanping Nie, Xiaohui Kuang, Zhechao Lin, Chenyifan Liu, Jie Liang, and Yu Jiang. 2022. Group-based corpus scheduling for parallel fuzzing. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1521–1532.

[29] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. In *Annual Computer Security Applications Conference*. 360–372.

[30] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*.

[31] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. 2018. Enhancing memory error detection for large-scale applications and fuzz testing. In *Network and Distributed Systems Security (NDSS) Symposium 2018*.

[32] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[33] Gregor Heinrich. 2005. *Parameter estimation for text analysis*. Technical Report. Citeseer.

[34] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, Vol. 40.

[35] Heqing Huang, Hung-Chun Chiu, Qingkai Shi, Peisen Yao, and Charles Zhang. 2023. Balance Seed Scheduling via Monte Carlo Planning. *IEEE Transactions on Dependable and Secure Computing* (2023).

[36] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–50.

[37] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. 2024. Titan: efficient multi-target directed greybox fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1849–1864.

[38] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1613–1627.

[39] Heqing Huang, Anshunkang Zhou, Mathias Payer, and Charles Zhang. 2024. Everything is Good for Something: Counterexample-Guided Directed Fuzzing via Likely Invariant Inference. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 142–142.

[40] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: using input type inference to improve fuzzing. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 505–517.

[41] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 171–185.

[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.

[43] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[44] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. 3559–3576.

[45] Yong-Fong Lee, Barbara G Ryder, and Marc E Fiuczynski. 1995. Region analysis: A parallel elimination method for data flow analysis. *IEEE Transactions on Software Engineering* 21, 11 (1995), 913–926.

[46] Sisi Li, Ruilin Li, Jiaxi Ye, and Chaojing Tang. 2020. Adaptive Parallel Fuzzing with Multi-candidate Task Scheduling. *Journal of Physics: Conference Series* (aug 2020).

[47] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 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*. 809–814.

[48] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. Pata: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17.

[49] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. 2023. Reachable Coverage: Estimating Saturation in Fuzzing. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23), 17-19 May 2023, Australia*.

[50] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers.. In *USENIX Security Symposium*. 1949–1966.

[51] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1024–1036.

[52] Matteo Marini, Daniele Cono D'Elia, Mathias Payer, and Leonardo Querzoni. [n. d.]. QMSan: Efficiently Detecting Uninitialized Memory Errors During Fuzzing. ([n. d.]).

[53] Thomas Minka. 2000. Estimating a Dirichlet distribution.

[54] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.

[55] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. 1683–1700.

[56] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level directed fuzzing for {use-after-free} vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 47–62.

[57] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2021. Collabfuzz: A framework for collaborative fuzzing. In *Proceedings of the 14th European Workshop on Systems Security*. 1–7.

[58] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1629–1642.

[59] Long Pham, Feras A Saad, and Jan Hoffmann. 2024. Robust resource bounds with static analysis and Bayesian inference. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 76–101.

[60] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 543–553.

[61] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.

[62] Van-Thuan Pham, Manh-Dung Nguyen, Quang-Trung Ta, Toby Murray, and Benjamin IP Rubinstein. 2021. Towards systematic and dynamic task allocation for collaborative parallel fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1337–1341.

[63] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.
[64] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. *arXiv preprint arXiv:2405.10220* (2024).
[65] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.
[66] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. MTFuzz: fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 737–749.
[67] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective seed scheduling for fuzzing with graph centrality analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2194–2211.
[68] Wensheng Tang, Dejun Dong, Shijie Li, Chengpeng Wang, Peisen Yao, Jinguo Zhou, and Charles Zhang. 2024. Octopus: Scaling Value-Flow Analysis via Parallel Collection of Realizable Path Conditions. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–33.
[69] Chengpeng Wang, Wenyang Wang, Peisen Yao, Qingkai Shi, Jinguo Zhou, Xiao Xiao, and Charles Zhang. 2023. Anchor: Fast and precise value-flow analysis for containers via memory orientation. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–39.
[70] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.
[71] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Jingzhou Fu, Zhuo Su, Qing Liao, Bin Gu, Bodong Wu, and Yu Jiang. 2024. Data coverage for guided fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 2511–2526.
[72] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. 2022. Odin: on-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1010–1024.
[73] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.
[74] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS*.
[75] Yifan Wang, Yuchen Zhang, Chenbin Pang, Peng Li, Nikolaos Triandopoulos, and Jun Xu. 2021. Facilitating parallel fuzzing with mutually-exclusive task distribution. In *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part II 17*. Springer, 185–206.
[76] Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. 2023. Enhancing Coverage-Guided Fuzzing via Phantom Program. In *2023 The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[77] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2313–2328.
[78] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 769–786.
[79] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.
[80] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 659–676.
[81] Yiru Zhao, Xiaoke Wang, Lei Zhao, Yueqiang Cheng, and Heng Yin. 2022. Alphuzz: Monte carlo search on seed-mutation tree for coverage-guided fuzzing. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 534–547.
[82] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 858–870.
[83] Xu Zhou, Pengfei Wang, Chenyifan Liu, Tai Yue, Yingying Liu, Congxi Song, Kai Lu, Qidi Yin, and Xu Han. 2020. UltraFuzz: Towards Resource-saving in Distributed Fuzzing. *arXiv preprint arXiv:2009.06124* (2020).

Received 2024-10-24; accepted 2025-03-31