SNOWCAT: Efficient Kernel Concurrency Testing using a Learned Coverage Predictor

Sishuai Gong
Purdue University, USA

Dinglan Peng
Purdue University, USA

Deniz Altınbüken
Google DeepMind, USA

Pedro Fonseca
Purdue University, USA

Petros Maniatis
Google DeepMind, USA

Abstract

Random-based approaches and heuristics are commonly used in kernel concurrency testing due to the massive scale of modern kernels and corresponding interleaving space. The lack of accurate and scalable approaches to analyze concurrent kernel executions makes existing testing approaches heavily rely on expensive dynamic executions to measure the effectiveness of a new test. Unfortunately, the high cost incurred by dynamic executions limits the breadth of the exploration and puts latency pressure on finding effective concurrent test inputs and schedules, hindering the overall testing effectiveness.

This paper proposes SNOWCAT, a kernel concurrency testing framework that generates effective test inputs and schedules using a learned kernel block-coverage predictor. Using a graph neural network, the coverage predictor takes a concurrent test input and scheduling hints and outputs a prediction on whether certain important code blocks will be executed. Using this predictor, SNOWCAT can skip concurrent tests that are likely to be fruitless and prioritize the promising ones for actual dynamic execution.

After testing the Linux kernel for over a week, SNOWCAT finds ~17% more potential data races, by prioritizing tests of more fruitful schedules than existing work would have chosen. SNOWCAT can also find effective test inputs that expose new concurrency bugs with higher probability (1.4×~2.6×), or reproduce known bugs more quickly (15×) than state-of-art testing tools. More importantly, SNOWCAT is shown to be more efficient at reaching a desirable level of race coverage in the continuous setting, as the Linux kernel evolves from version to version. In total, SNOWCAT discovered 17 new concurrency bugs in Linux kernel 6.1, of which 13 are confirmed and 6 are fixed.

CCS Concepts: • **Security and privacy → Operating systems security**; *Software and application security*; • **Software and its engineering → Software defect analysis**.

Keywords: Kernel concurrency bugs, Operating systems security, Software testing and debugging, Concurrency programming

ACM Reference Format:
Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2023. SNOWCAT: Efficient Kernel Concurrency Testing using a Learned Coverage Predictor. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3600006.3613148

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613148

1 Introduction

Finding kernel concurrency bugs is challenging due to the extensive search space of kernel test inputs and thread interleavings. Finding a concurrency bug first requires choosing effective concurrent test inputs—a pair or more sequential test inputs that concurrently invoke sequences of system calls [29]. Second, finding a concurrency bug requires choosing error-inducing schedules of the concurrently-executing kernel threads [16, 17, 39, 41, 43]. Hence, the search space is at least quadratic in the number of sequential inputs, and exponential in the number of instructions that can interleave among concurrent threads [39, 40], making kernel concurrency testing particularly daunting.

Although generating good kernel concurrent tests is a broadly explored area of research and practice [17, 19, 25, 68], deciding how *fruitful* such a generated test is towards a testing goal is a common and important challenge. Typically, testing campaigns target a reward such as an increase in code coverage, data-race discovery, or triggered undesirable behaviors (e.g., kernel panics and deadlocks) [19, 21, 25, 28, 68]. However, computing such a reward by executing the candidate test is expensive. These kernel executions run in heavyweight environments (e.g., VMs) with expensive instrumentation [2, 19–21, 28, 36, 53, 71], yielding particularly low execution throughput (e.g., tens to hundreds of executions per minute).

This naturally limits the efficiency of concurrency testing since the vast majority of random tests do not increase

coverage [23, 30, 46]. In fact, even sequential test input generation, which generally has a smaller search space, is haunted by efficiency challenges. For example, Syzkaller, a mature feedback-based kernel fuzzing tool, may need to execute thousands of random sequential inputs before finding one that increases coverage [53, 61]. Moreover, the chance of finding tests that increase the overall coverage drops significantly throughout a fuzzing campaign, making the test execution stage even more wasteful [30, 53, 61].

Thus, there is an opportunity to *prioritize interesting concurrency tests and filter out less interesting ones* to curtail the large waste in CPU and wall-clock times from fruitless dynamic executions. Instruction schedules likely to exercise previously uncovered code blocks can be prioritized over schedules that exercise previously observed blocks. This is the quest we embark on in this work. In prior work, one plan of attack is to choose likely-to-be-fruitful concurrency tests at construction, and another is to estimate a reward metric for a test after construction but before execution to filter appropriately. We summarize examples of each.

Since it is known that effective concurrent inputs should exercise diverse kernel inter-thread data flows [19, 36, 42, 68], static analysis is sometimes used to reason about data flows that would be triggered by an input. Unfortunately, because real-world kernels are complex, traditional analysis approaches face limitations in either accuracy [12, 25, 44, 60] or scalability [7, 23, 30, 37, 48, 70].

Thus, heuristics that do not require heavy analysis are used [19, 34, 68]. For example, Snowboard [19], our previous system, prioritizes the tests of concurrent test inputs whose two constituent sequential inputs both touched the same memory when executing single-threaded, as those are likely to exhibit inter-thread data flow when run together. Finding effective schedules is also challenging due to the massive interleaving space [39, 41, 42]. A kernel concurrent test can run concurrently tens of thousands of instructions from each thread [17], making it infeasible to enumerate all possible interleavings. Hence, a targeted approach is necessary that finds and prioritizes interleavings exercising unique concurrent behaviors. Exhaustively analyzing the consequences (e.g., coverage, data flows, etc.) of interleaved instructions from multiple threads requires formal approaches, such as model checking [33, 58, 67]. However, these approaches do not scale well to low-level, complex systems, such as the kernel. In practice, constrained random schedulers [6, 17, 18] that only invoke a fixed number of thread switches per execution are commonly used. The limit on thread switches helps prune the interleaving space, enabling more systematic exploration.

Our work is inspired by the dramatic advances of machine learning (ML) towards code understanding. We propose general and automatic techniques that estimate whether a concurrency test is likely to be fruitful. ML approaches have been used before for software and hardware testing. Neuzz [50] showed that neural networks can learn and predict application edge coverage given the test input. Given the byte sequences of the test input, Neuzz identifies promising bytes that should be mutated for higher coverage. Design2vec [56] further shows that the control and data flow graph of the testing target (hardware in Verilog RTL [66]) can be used by a model to predict test coverage. The success of Neuzz and Design2vec on hardware designs and small-scale applications suggests that ML models may accurately and efficiently predict the execution of concurrent kernels.

However, new challenges arise when applying an ML approach to analyze concurrent kernels. First, representing kernel test inputs—recall, these are userspace programs invoking sequences of system calls—in byte sequences, as Neuzz and Design2vec do, would make it hard for the model to learn because of the extremely long execution paths of kernel APIs. Before analyzing the consequences of interleaved instructions, the model would have to infer the system call specification [22], entry points of different APIs [1, 8, 30], execution paths of system calls [4, 30, 67], and then potential interactions between threads [19], among others, exclusively from the plain input byte sequence. Every task in this pipeline is known to be notoriously challenging and often requires specialized approaches to address; it is unrealistic to expect current ML techniques to solve them all in one shot.

Second, presenting the whole kernel's control and data flow graph, which contains millions of code blocks[1], to a model, as Design2vec does, will incur severe scalability and latency problems. As shown previously [69], when the input graph is large (e.g., over 2M nodes), one model inference can take almost 3 seconds. This time cost is already close to the time of a dynamic execution, which takes about 2.8 seconds per run (§5.2.2), and would cancel most of the benefits of a predictive technique. These two challenges motivate a new design of both input representation and model to target the unique case of kernel concurrency testing.

This paper proposes SNOWCAT, a kernel concurrency testing framework that prioritizes schedules and test inputs using a learned kernel coverage predictor. The predictor uses a graph neural network model trained to predict whether certain concurrency-sensitive kernel blocks will execute, if the kernel runs a concurrent test input under a given schedule. Importantly, the predictor is designed to be efficient so that it can perform many predictions in the time it takes to execute a single concurrency test, and can, therefore, yield higher end-to-end testing effectiveness than state-of-art tools, even when considering the model training cost.

SNOWCAT identifies a set of concurrency-sensitive code blocks that are particularly interesting to predict. Our key observation is that traditional analysis approaches usually struggle to analyze *uncovered reachable blocks* when a concurrent test input runs under different interleavings. These

[1] A compiled 6.1 kernel has 2.7M blocks

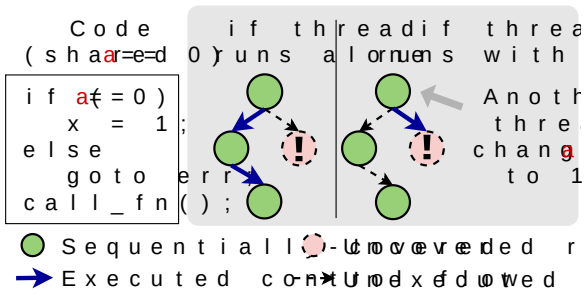


Figure 1. Example of how *uncovered reachable blocks* can be triggered under concurrent executions.

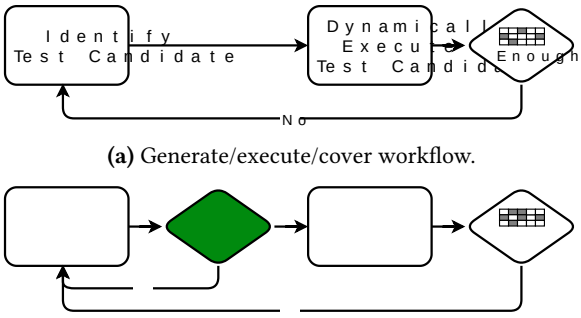
are kernel code blocks that are *reachable* but not actually reached when the constituent sequential input runs single-threaded: a (short) control-flow path to them exists, but is not triggered in a single-threaded execution of the test input. Naturally, such blocks might be reached when two sequential inputs run concurrently and interfere with each other, as illustrated in Figure 1. Coverage prediction on such *uncovered reachable blocks* is useful because it can guide a concurrency testing tool to prioritize tests that exercise control flows different from those covered in sequential executions. Similarly, SNOWCAT predicts when *sequentially-covered blocks* are *not* covered in a concurrent test since this also signifies a new behavior.

The coverage predictor enables SNOWCAT to efficiently address the challenges in both concurrent input and schedule generation. Thanks to its fast inference (§5.2.2), SNOWCAT can consider a large pool of candidates and only select a few likely-to-be-fruitful tests to execute; without a coverage predictor, for the same budget of executions, only a far smaller pool of candidates could be considered, potentially missing bugs. Furthermore, when the testing target is a specific part of the kernel (e.g., two instructions that potentially cause a data race), the coverage predictor enables directed testing to find triggering test inputs faster.

SNOWCAT is evaluated on mature versions of the Linux kernel. First, SNOWCAT can select more effective schedules given random concurrent inputs—it finds 17% more potential data races than the state-of-art approach on Linux kernel 6.1 in a week-long search. Second, SNOWCAT can find concurrent test inputs that trigger bugs faster or with higher probability. For 6 new concurrency bugs in Linux kernel 6.1, SNOWCAT can find them with a probability 1.4×~2.6× higher on average than related work. For 6 known data races in Linux kernel 5.12, SNOWCAT can find the error-inducing test inputs 15× faster on average than an existing tool. More importantly, an analysis of the end-to-end time cost (including model training) shows SNOWCAT is more efficient: it reaches the same amount of potential data races nearly a hundred hours faster than the state-of-art.

This paper makes the following contributions:

- **A kernel block coverage predictor for concurrent executions.** This paper designs a new representation method for concurrent test inputs and schedules, a model architecture that can learn to predict kernel code block coverage when the given concurrent test input is executed under the given schedule.
- **A new workflow for efficient kernel concurrency testing.** SNOWCAT proposes a new kernel testing workflow in which the newly-generated test candidates are first evaluated based on the predicted coverage. Then only interesting test candidates will be selected for dynamic executions.
- **An evaluation on the new workflow.** We evaluate SNOWCAT to find interesting schedules given a concurrent input, and to find test inputs that trigger unique kernel executions or reproduce data races. SNOWCAT discovered 17 new concurrency bugs in Linux kernel 6.1.



(a) Generate/execute/cover workflow.

(b) Filtering out dynamic tests that are unlikely to be fruitful.

Figure 2. Comparison between the general workflow and a predictor-based workflow.

SNOWCAT is publicly available[2].

2 Motivation

We consider the problem of coverage-driven concurrency testing for large software systems; we instantiate the problem specifically to the Linux kernel due to its pervasive adoption and utmost significance.

In particular, consider the general workflow shown in Figure 2a. In this workflow, Sequential Test Inputs (STIs) are identified (i.e., what each thread will execute), put together into a Concurrent Test Input (CTI), and a candidate interleaving schedule is chosen; then a dynamic test is run with the chosen Concurrent Test (CT) (i.e., CTI + interleaving). After the dynamic execution, some termination condition is checked, for example a race detector in bug-finding use cases, or a cumulative coverage collector, in coverage-driven testing, or a time limit otherwise. If the termination condition is not met, another candidate is considered.

In this workflow, the dynamic-test execution consumes most of the computational resources (§1). Yet, most dynamic tests are not fruitful: they might not get closer to the termination condition, either because they do not increase the

[2]SNOWCAT artifact: https://github.com/rssys/snowcat

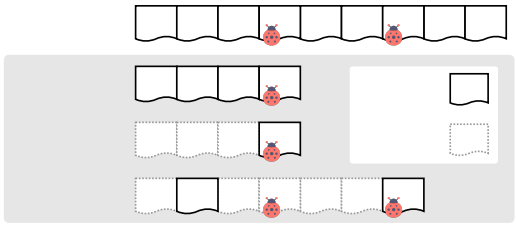


Figure 3. Filtering a stream of test candidates with a perfect and an imperfect filter.

cumulative coverage metric, or they do not trigger some exceptional code blocks (e.g., an assertion). Our work seeks to reduce the number of dynamic tests that need to be executed before the termination condition is met, moving to a workflow that looks like Figure 2b instead.

To be useful, the green diamond must meet certain challenges: **High Speed.** It must be fast. In particular, it must be faster than a dynamic execution. Otherwise, it would be more cost effective (and simpler) to just execute the dynamic test. **Low False Positives.** When it deems a dynamic test helpful, it must be correct much of the time, otherwise it would cause fruitless dynamic executions, defeating the purpose for such a component. **Low False Negatives.** Such a component would be fast indeed if it always said that a test will be fruitless. However, that would preclude any dynamic tests from ever executing, and consequently would make no progress towards the termination condition.

We are considering such a component built using machine learning. ML models typically require considerable resources to (a) collect training data and (b) train the model. This leads to an additional challenge: **Low End-to-End Cost.** Training data generation and training time must be low enough for the end-to-end cost of the approach to be lower than the original workflow.

To put these challenges in perspective, consider this workflow in a simplified setting (that models a race-driven exploration), in which an infinite stream of test candidates is available, and the termination condition is a simple boolean predicate, e.g., "was a target data race triggered during the execution?" (see Figure 3). The (current) exhaustive approach would execute all candidate tests, resulting in 4 test executions in this scenario. An omniscient model that perfectly predicts whether a test will be fruitful (positive in this scenario) would only execute a single test. A more realistic model would have some false positives (executing test 2 even though it is fruitless), and some false negatives (*not* executing test 4, even though it would have been fruitful), and would keep going until the second fruitful test 7 is encountered. §A.6 explores this analytically.

Finally, we are considering the steady state of keeping Linux kernels properly tested as the code evolves from version to version. It is not only important to get to a high level of test validation quality for a single kernel version; it is also important to be able to adapt quickly to the next version, and the one after that. This brings the final challenge: **Generalization.** Especially with frequent kernel updates, the cost of training can add up. An ML-based test evaluator should be able to generalize from version to version, with limited additional data-gathering and training cost, possibly by building upon prior training datasets and models.

To summarize, we seek to build a rejection filter for candidate concurrency tests. To be effective given a goal (e.g., target code coverage), such a filter must be cheap enough to build and update as new kernels arrive to be worth the reduced cost from the skipped dynamic tests that it filters.

3 Design

To address the challenges outlined in §2, we consider the design of a *learned coverage predictor* for CTs. In particular, we build our system around the following design principles:

Train on multimodal data. Many types of kernel information including syntax, semantics, and single-thread executions can be used to train effective models for concurrency testing. Notably, much of these data sources is the byproduct of other kernel testing procedures such as kernel fuzzing and static analysis, making their collection cost-efficient.

Predict block coverage. Although several coverage metrics exist (e.g., alias coverage [68]), in kernel testing, which involves copious error-handling code [35] and assertions, the basic code-block coverage can be representative of exceptional kernel behavior [20].

One interleaving at a time. Much kernel-testing work considers separately the choice of STIs to execute on different threads, or the interleaving to combine them into a CT. Even when exploring interleavings, some are useful and some are not, providing an opportunity to save wasted effort. Therefore, we focus on predicting the coverage of a CT—a CTI plus a target schedule.

Predict coverage of *uncovered reachable blocks* and *sequentially-covered blocks*. ML systems are usually constrained by the size of their inputs and outputs, so it is important to reduce the context of any ML prediction. In the case of a block-coverage predictor, a single CT is likely to cover only a small amount of code blocks of the entire kernel. On the other hand, all we know about a CT is that its constituent STIs already covered a limited set of *sequentially-covered blocks*, but the major point of concurrency testing is to see *what else* we can get those tests, run concurrently, to cover. Therefore, we structure our learning objective as one of *also* predicting the coverage of *uncovered reachable blocks* (*URBs*) and *sequentially-covered blocks* (*SCBs*) from the test's code. "Reachable" means blocks that could be reached via a constrained control flow path from the code run by the test threads. "Uncovered" means blocks that are not covered by each test thread when run sequentially. *Sequentially-covered blocks* are still interesting to predict because they can be

affected by concurrency as well (e.g., a function skipped because of a changed control flow). This design choice limits the size of the task examples to just the blocks covered during sequential runs and those reachable within a small number of hops, making our predictor feasible and efficient.

Putting all these design directions together, SNOWCAT trains an ML coverage predictor that takes as input a CT (two STIs and a target interleaving), and predicts a Boolean ("covered" or "not covered") for every code block, including *SCBs* and *URBs* of the two STIs. In a sense, *SNOWCAT learns to predict the block coverage of a CT from the block coverage of its constituent STIs and the target interleaving.*

Specifically, SNOWCAT fits in the following workflow, typical with feedback-based fuzzing systems [19, 20, 25, 36, 68]:

1. Similar to SKI [17], Razzer [25], and Snowboard [19], it assumes a source of STIs, generated by a tool such as Syzkaller [20] or fsstress [3].
2. Similar to Snowboard and Razzer, it uses information already collected during the single-thread execution of STIs (e.g., control flow) to prime a downstream CT generator.
3. It uses a static analysis tool to build a control flow graph (CFG) of the whole kernel, so that *uncovered reachable blocks* (within k hops) can be easily identified from those sequential executions.
4. It collects a training dataset of concurrent executions, by running an existing tool, such as SKI, to generate and execute CTIs under target schedules. Before the execution, it records the test input and the target schedule. After the execution, it collects the block coverage.
5. Given this training dataset, it trains a *per-interleaving coverage* predictor model (*PIC*) that, given the CT candidate including its assembly code, dataflow, controlflow, and schedule edges, predicts the coverage of *URBs* and *SCBs*.
6. At run time, the *PIC* model is used to filter out candidate CTs, by predicting their coverage, and using a strategy to judge whether that coverage would be fruitful towards some goal (e.g., towards increasing overall coverage, or towards reaching an interesting block).
7. When a new version of the kernel is available, data generation and training steps are repeated to fine-tune the existing *PIC* model to the new kernel version.

We describe the details of our approach next.

3.1 CT Data Representation

At the core of SNOWCAT lies an ML model that predicts the coverage of blocks during the execution of a CT. We now explain how we present a CT to the model.

Since we use a variety of information (code, static analysis, dynamic behavior) about a CT, we chose a graph representation. This is similar to prior ML work that focuses on testing,

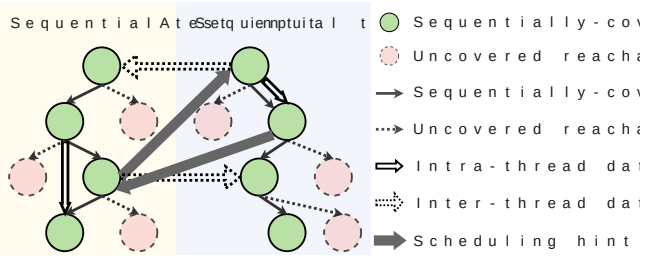


Figure 4. The SNOWCAT graph representation of a CT candidate example.

e.g., design2vec [56], or that aims to learn deep static analyses, e.g., ProGraML [10].

A CT consists of a pair of STIs and scheduling hints. Each STI is a sequence of system calls that will be invoked from one application process, and the scheduling hint tells the executor how to schedule the two kernel threads (e.g., "switch to thread B when thread A executes the i-th instruction"). Figure 4 shows an overview of the graph representation in SNOWCAT. The CT graph is made of vertices corresponding to kernel basic blocks (i.e., sequences of assembly instructions uninterrupted by control-flow entry or exit).

There are two types of vertices: *sequentially-covered blocks* (*SCBs*), i.e., blocks that were covered during the sequential execution of the two constituent STIs; and *uncovered reachable blocks* (*URBs*), i.e., blocks that are *statically* reachable from the *sequentially-covered blocks*, within a small number of control-flow hops, but that were not reached during the sequential execution. We use the whole-kernel CFG to identify *URBs*. We set the limit to only identify 1-hop *URBs* to avoid path explosion and maintain a reasonable number of nodes per CT graph (§5.1.1). Each vertex holds two features: the vertex type (*URB* or *SCB*) and the code in the basic block (assembly instructions as text).

The vertices are connected by five types of edges. *SCB* control-flow edges represent the control flow taken during the sequential execution of the constituent STIs. *URB* control-flow edges represent static control-flow edges connecting the *SCBs* to *URBs*. A third edge type that demonstrates intra-thread data flow during the sequential execution connects blocks in the same thread. A fourth edge type connects blocks from different threads in this case that have a *potential data flow*; potential data flow occurs between two instructions in different threads, of which one is a write and the other a read, and that address overlapping memory ranges [19, 36]. Finally, a fifth edge type represents the candidate schedule as a pair of *scheduling hints*: proposed yield points from a block in one thread to a block in another thread (see next few paragraphs). A summary of the parts of a CT graph can be found in the appendix (Table 7).

Scheduling-Hint Edges. In our CT graphs, the scheduling hints are enforced via virtualization tools such as SKI, which implements a uni-processor scheduler (i.e., only one thread runs at a time). Given threads *A* and *B*, we consider

two scheduling hints marked by *thread-switch-point* instructions $A.x$ and $B.y$. The concurrent-test execution system will (try to) enforce these two hints by starting with the first instruction of A, executing A up to instruction x, and then yielding to thread B. Thread B will then execute from its first instruction up until it reaches instruction y, at which point it will yield back to A. A will then continue from the instruction following x. It is meaningful to have more than two scheduling hints, but we configure SNOWCAT to set two scheduling hints per CT because they are sufficient for discovering most concurrency bugs [42]. A similar setting is also used in related work [15, 17, 25].

We call these scheduling instructions "scheduling hints" because the actual interleaving exercised might be different from what was hinted. For example, SKI [17] will skip a thread switch if the *thread-switch-point* is not encountered, or will invoke additional switches if it detects deadlocks (e.g., a thread switch happens in the critical section).

Our graph representation captures the scheduling hints in the above example, by connecting a scheduling hint edge from the block containing instruction $A.x$ to the first block of B, and a second edge from the block containing $B.y$ back to the block containing $A.x$; note that this is a simplification of the case where the successor instruction of $A.x$ lies in a different block from $A.x$, and it essentially tells the model to "finish the block it was executing before the yield".

3.2 *PIC* Model Architecture

The goal of the per-interleaving coverage (*PIC*) predictor is to predict which, if any, of the *URBs* and *SCBs* of the two threads are covered. The model is trained using actual dynamic tests and their observed coverage upon completion. Specifically, training examples consist of input/target pairs $< x_i, y_i >$, where x_i is the CT graph (§3.1), and y_i is an assignment of `COVERED`/`UNCOVERED` to the vertices of x_i.

More precisely, x_i is a graph (V_i, E_i), where the vertices are $V_i = C_i \cup U_i$ (the *SCBs* and *URBs*, respectively), and the edges are $E_i = S_i \cup P_i \cup D_i \cup I_i \cup A_i$ (the sequential control flow edges, the intra-thread data flow edges, the possible control flow edges to uncovered blocks, the inter-thread possible data flow edges, and the scheduling hint edges, respectively). Similarly, $\forall v \in V_i, y_i[v] \in \{\texttt{COVERED}, \texttt{UNCOVERED}\}$ (covered/uncovered under the concurrent execution).

SNOWCAT uses a model architecture that consists of two major modules. First, a sequence model (BERT) [11] that is responsible for generating embeddings of code blocks based on their assembly code. Second, a graph neural network (GNN) [49] that takes the graph as input, learns relationships between embeddings of code blocks and performs a binary classification on every node (code block) in the graph.

Since the graph neural network architectures we use are standard, we just outline the GNN "interface" here. It can be seen as a parametric function that predicts targets $\hat{y}$ from input graphs x, $\text{GNN}(x; \text{Emb}(.); \theta_{\text{GNN}}) = \hat{y}$, where θ_{GNN} are the learnable parameters of the model, and Emb is an *input embedding function* of the graph features into vectors of floating-point numbers, so that they can be used readily by the GNN. Recall that our graph features (besides the graph structure itself) are the vertex and edge types, and the text representation of a vertex (block) as assembly.

To embed vertex and edge types, we use a simple learnable embedding matrix that maps types to learnable parameters θ_{Emb}, one per type (2 types of vertices, 5 types of edges).

To embed the assembly text *asm*, we use a standard BERT-like encoder (an instance of a Transformer [57]), pre-trained on all assembly code in the Linux kernel. We treat all assembly as text, but elide any numerical tokens, such as register offsets, since they do not provide much useful signal to the model, and their semantics (e.g., memory accesses) are captured by other features in our graphs already. We then pre-train BERT with this preprocessed assembly text, to learn a $\text{BERT}_{\text{asm}}(asm; \theta_{\text{BERT}})$ function in the standard way [11] (i.e., training on a masked language model objective). We use this BERT-on-assembly encoder as the embedding function for the assembly feature of every vertex in the graph.

In summary, the learnable parameters of our model are θ_{GNN} for the GNN itself, θ_{BERT} for the Assembly encoder, and θ_{Emb} for the 2 vertex and 5 edge types. Note that θ_{BERT} is pre-trained once, since what looks like "natural" assembly code does not change much from kernel version to version. However, we do fine-tune these parameters during the training of the GNN whenever a new *PIC* model is trained on a new kernel version.

We train the GNN by minimizing the binary cross-entropy loss between the predicted coverage $\hat{y_i}$ and the ground truth y_i of all blocks. We compute the binary cross entropy between target and prediction in each graph example first, and the model minimizes it across the examples of the training population.

3.3 Predicted-Coverage-Guided Concurrency Testing

Once the *PIC* model is trained, SNOWCAT can use it to predict the block coverage of new CT candidates that consist of new CTIs and schedules. This section introduces how SNOWCAT selects interesting schedules and CTIs for dynamic executions based on the predicted block coverage.

SNOWCAT can use an external interleaving exploration tool to propose new schedules (scheduling hints) and then SNOWCAT generates the graph CT of these new candidates to get the predicted block coverage from the *PIC* model. Finally, SNOWCAT applies a prioritization strategy on the predicted coverage and only executes the CT if it is interesting under the strategy.

SNOWCAT uses one of three strategies to select interesting CT candidates based on the predicted *SCB* and *URB* coverage. Their effectiveness in finding effective schedules and CTIs is evaluated in §5.3 and §5.6.2.

S1: New set of positive blocks. Under this strategy, a CT is interesting if it can trigger a new predicted coverage bitmap (sequence of block-coverage Booleans) that has not been observed before. The intuition is that new coverage roughly determines a controlflow change, even if it does not necessarily cover any new individual blocks. To avoid future CTs that produce the same coverage, SNOWCAT remembers the predicted block coverage of each previously chosen CT.

S2: New positive blocks. Under this strategy, a CT is interesting and selected if the predicted *URB* and *SCB* coverage contains at least one code block that has not been observed before. Similar to S1, SNOWCAT remembers predicted-to-be-covered code blocks of every CT it selects, so future CTs can be evaluated.

S3: Positive blocks with limited trials. This strategy limits the number of executions that each positive code block can be attempted. On the one hand, a trial limit higher than 1 can encourage a code block to be attempted several times (e.g., in different calling stacks). On the other hand, the trial limit will prevent SNOWCAT from trying too many CTs on blocks that might be false positives produced by the model.

4 Implementation

Concurrent test candidate representation. SNOWCAT uses Syzkaller to generate and execute sequential test inputs (STIs). During the STI execution, SNOWCAT collects necessary information such as the *SCB* control flow. SNOWCAT uses Angr [52] to build the kernel CFG, which is necessary for *URBs* identification. In total, we wrote ~2.5K LOC in Python for converting a concurrent test candidate to an input graph to the model.

Graph dataset collection. To label graphs for training and evaluation, SNOWCAT modifies SKI—a customized QEMU emulator that applies PCT [6] interleaving exploration on the guest kernel—to dynamically execute and profile the concurrent test candidates, so the block coverage can be collected and used for labeling all nodes in the graph. ~0.5K LOC in C is added to SKI to instrument the guest kernel executions for trace collection. Around 1K Python code and 0.2K LOC Bash scripts are implemented for automating and distributing data collection.

Model training and evaluation. The assembly code embedding module is a RoBERTa model trained using the framework fairseq [45]. The GNN module is a GCN [32] implementation from the Pytorch Geometric framework [14]. In total, about 1K LOC in Python is implemented for training the *PIC* model; 5K LOC Python code and 0.5K LOC Bash scripts for the evaluation.

Kernel concurrency testing. The evaluation of SNOWCAT uses existing kernel concurrency testing tools including SKI, Razzer and Snowboard. About 1K LOC Python code and 500 LOC C code are implemented to integrate the coverage predictor and perform concurrent test candidates selection.

5 Evaluation

The section evaluates the effectiveness and efficiency of SNOWCAT in kernel concurrency testing with respect to existing testing tools. Specifically, it seeks to answer four questions:

RQ1: Can the *PIC* model accurately predict the coverage of *URBs* in concurrent kernel executions? (§5.2)

RQ2: Can SNOWCAT identify more effective test candidates given a budget by using the coverage predictor? (§5.3)

RQ3: Can the cost of SNOWCAT amortize well as kernels evolve? (§5.4)

RQ4: Is the *PIC* model beneficial to existing testing workflows? (§5.6)

Setup overview. We evaluated SNOWCAT on Linux kernels 5.12, 5.13, and 6.1. First, we focus on Linux kernel 5.12 for the initial proof of concept. We train, tune, and evaluate *PIC* models on Linux 5.12 data. Second, Linux kernels 6.1 and 5.13 are used to study the generalization ability of *PIC*, in which different retraining trade-offs are studied. The experimental platform details are described in §A.1.

5.1 *PIC* Model Training

We now describe our training methodology, given the *PIC* architecture (§3.2).

5.1.1 Dataset Construction. Although it is important to produce datasets to evaluate RQ1 (a typical ML microbenchmark evaluation), we are also interested in how SNOWCAT can be used in "practical" settings, as per the remaining RQs. This means that the "test" period for the model is significantly longer than the training and validation period. We have therefore constructed training/validation/evaluation[3] datasets that deviate from the typical 90%/5%/5% example mix in ML research.

Specifically, we collected 44,686 concurrent test inputs (CTIs) (i.e., random pairs of sequential test inputs (STIs)) from SKI, on Linux kernel 5.12, and we split them into 21,621 training CTIs, 2,702 validation CTIs, and 20,363 evaluation CTIs. We then produced 64 interleavings for the training and validation CTIs, and 1000 for the evaluation CTIs; the much higher number of interleavings for evaluation CTIs was meant to facilitate experiments where we want to give SNOWCAT the ability to search for good schedules for a long time, beyond what a typical tool like SKI might do. When projected to our block-oriented graphs, this resulted in, on average, 64 unique interleavings per CTI for training and validation, and 953 unique interleavings for evaluation, for

[3]Note that we use the term *evaluation* for what is typically called the *test* split in ML research, because all our examples are "tests" and it would be confusing to use the same term for two concepts.

a total of 1.37M, 0.17M, and 19.05M graphs across the three dataset splits. A CT graph contains, on average, 9.7K vertices (2.4K *URBs* and 7.3K *SCBs*), and 14.1K edges (8.4K *SCB* control flow, 4.2K *URB* control flow, 1K intra-thread data flow, 0.6K inter-thread data flow, and always 2 scheduling hint edges).

We also augmented our graphs with *shortcut edges*—edges that connect vertices that are k sequential control-flow edges apart—which is a common "densification" technique that improves model performance on code GNNs [56].

5.1.2 Model Tuning. To train *PIC*, we explored 80 different sets of hyperparameters (see §A.2), trained for 5 epochs. We then chose the model training checkpoint with the highest Average Precision (AP) [63]; AP computes the mean precision (true positive predictions divided by all positive predictions) over all recall values.[4] This gives a metric of the "goodness" of a model across all tuning points. The model with the highest AP is called *PIC*-5. To favor positive predictions on "surprising" blocks, we computed AP over *URBs* only when selecting hyperparameters.

One interesting observation from this hyperparameter exploration is that *PIC* models that have deeper GNN modules can achieve higher performance; the number of layers of a GNN is roughly equivalent to from "how far" in the graph information is gathered before making a decision about a vertex. In our case, this observation indicates that analyzing concurrent executions requires considering broader control and data flows.

PIC-5 was then tuned to choose a threshold for the predicted classification probability. We chose the threshold with the highest mean F2 score [64] on graph *URBs* over the validation dataset. We chose F2 because it favors a higher recall over a higher precision.

5.2 *PIC* Model Performance

5.2.1 Model accuracy. The performance of *PIC*-5, under the tuned threshold, is evaluated using several binary classification metrics [64, 65] on the evaluation dataset (§5.1.1). Due to the lack of advanced analysis approaches that are comparable to the *PIC* model, several baseline approaches are proposed and used for comparison:

- **All blocks as positive predictor (All pos)** predicts every node in the graph as positive. This predictor represents a simple static analysis approach.
- **Random binary predictor (Fair coin)** predicts every node in the graph as positive with a probability of 50%.
- **Biased random binary predictor (Biased coin)** predicts nodes in the graph as positive with a probability of 1.1%. This is the average frequency of positive *URBs* in our training graphs.

[4] A classifier typically predicts a probability of positive result. A tunable threshold determines when a prediction is reported as positive. The threshold can be tuned to output fewer but higher-confidence positive predictions, trading off precision and recall.

Predictor	F1	Precision	Recall	Accuracy	BA
PIC-5	55.13%	48.34%	69.18%	99.01%	84.47%
All pos	2.17%	1.11%	99.55%	1.11%	49.77%
Fair coin	2.14%	1.10%	49.76%	49.99%	50.00%
Biased coin	1.02%	1.11%	1.17%	97.74%	50.22%

Table 1. *URBs* predictor performance. Average metrics across all graphs. BA stands for balanced accuracy.

Table 1 presents results on *URBs* in each graph. *All pos* has extremely low accuracy while *Fair coin* and *Biased coin* have much trouble with precision. The root cause of their bad performance is that positive/negative labels are extremely skewed for *URBs*. In other words, most *URBs* are actually not covered during the concurrent executions, so naive baselines cannot predict accurately.

PIC-5 achieves much better performance across metrics. First, its accuracy is satisfying. Considering the accuracy is now dominated by the true negative rate due to the skewed label distribution, the high accuracy indicates that *PIC*-5 has a high true negative rate—it can accurately identify *URBs* that are actually not-covered during concurrent executions. Second, *PIC*-5 outperforms the baselines by two-digit margins on precision and recall. It is expected that the precision and recall are a bit lower than accuracy because the former two metrics reflect how well the predictor can correctly identify the actually-covered *URBs*, which is more challenging than identifying the actually-not-covered ones.

We show results here for just *URBs*, because they are a harder target subpopulation, but we also show results on the full set of blocks in §A.3, and they look similar.

5.2.2 Inference cost. The *PIC* model can make predictions fast once trained and deployed for inference. On our inference machines, it takes on average 0.015 seconds to predict the coverage for one CT candidate. In contrast, one dynamic execution of a candidate takes 2.8 seconds because of the heavy instrumentation for thread serialization and bug detection. In other words, Snowcat is able to predict coverage for 190 test candidates in the same time it takes to run one dynamic execution. This favorable performance asymmetry, balanced with reasonable precision and recall of *PIC*, explains our positive end-to-end efficiency results in the rest of this section.

5.3 Selecting Interesting Schedules with *PIC*

Snowcat integrates the *PIC* model into SKI, which uses the PCT algorithm to explore interleavings. By combining PCT with *PIC* to select promising interleavings according to our selection strategies (§3.3), we build the *MLPCT* exploration algorithm. The effectiveness of Snowcat hinges both on the predictive power of *PIC*, and on the choice of the selection strategy. This section studies the impact of *MLPCT* towards achieving high coverage, compared to SKI using PCT. Two metrics are proposed:

- *Data-race-coverage* measures the number of unique possible data races found by a data race detector (an implementation of DataCollider [13]) in explored interleavings.
- *Schedule-dependent block coverage* measures the number of unique code blocks under concurrent executions excluding all *SCBs* of the concurrent test. Higher block coverage implies both that more kernel behaviors are explored, but also that concurrency-dependent behaviors are being explored, which is the goal of this work.

We study both kinds of exploration goodness under two scenarios: (a) given a CTI, what is the maximum coverage metric we can get (§5.3.1), and (b) given a stream of CTIs, what is the maximum cumulative coverage we can get (§5.3.2). Both experiments focus on testing Linux kernel 5.12 and *PIC*-5 is used by *MLPCT*.

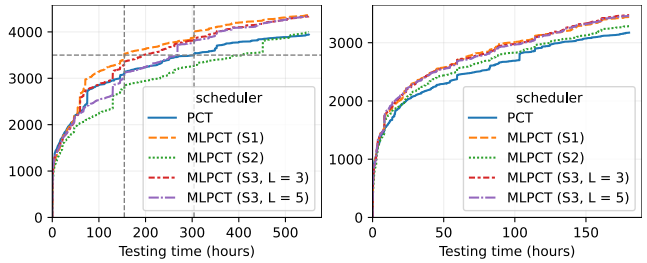
5.3.1 Coverage Improvement Per CTI. For this experiment, we choose a random CTI, and then explore it using SKI as well as our *MLPCT*/SKI variants. We use a budget of 50 dynamic executions for both, but also cap the number of *PIC* inferences to 1,600. We do this for 1.3K CTIs, and we report coverage increase averaged over all inputs.

Most *MLPCT* strategies perform better than PCT (10% to 20% more data races and 6.5% to 25.8% more covered *schedule-dependent* code blocks), showing that *MLPCT* can identify more fruitful interleavings for actual dynamic executions.

We have also studied how increasing this budget all the way to 200 affects the *MLPCT* benefit. As the original PCT now executes more dynamic tests, it gets closer to a saturation point, so *MLPCT* has less headroom to shine. This is consistent with observations in the original SKI work [17] about the number of useful unique schedules for a single CTI. Appendix A.4 shows more detail.

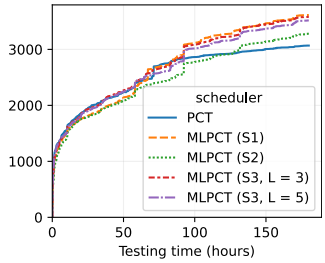
5.3.2 Cumulative Coverage Improvement. In this experiment, we seek to achieve the highest coverage by running SKI and our ML-enabled variants on a stream of PCT-generated CTIs, each receiving a budget of 50 dynamic test executions. Unlike the experiment in §5.3.1, earlier CTIs have a larger "unexplored" coverage map, but as more CTIs and their interleavings are tested, that coverage saturates.

As shown in Figure 5a, most *MLPCT* strategies reach higher coverage in terms of unique data races sooner than SKI (up to 10%). As an illustrative example, SKI took 304 hours to reach 3,500 unique possible data races, whereas the best Snowcat strategy S1 took only 155 hours. Strategy S2 seems to be overly conservative: it only selects schedules that are predicted to execute at least one uncovered code block (§3.3), but because we cap inferences at 1,600, it runs out before it reaches all 50 dynamic executions. This is understandable considering the skewed distribution of

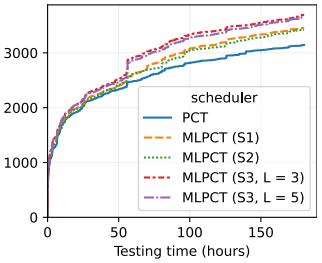


(a) Testing Linux kernel 5.12 using *PIC*-5

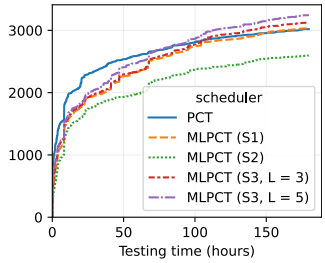
(b) Testing Linux kernel 6.1 using *PIC*-5



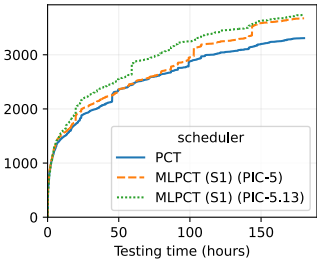
(c) Testing Linux kernel 6.1 using *PIC*-6.ft.sml



(d) Testing Linux kernel 6.1 using *PIC*-6.ft.med



(e) Testing Linux kernel 6.1 using *PIC*-6.fs.med



(f) Testing Linux kernel 5.13 using *PIC*-5 and *PIC*-5.13.ft.sml

Figure 5. *Data-race-coverage* history comparison between PCT and *MLPCT*. Varied total numbers of data races between figures are due to the non-deterministic random CTI and schedule generation.

positive *URBs*. Other strategies explore new, unexplored coverage maps (i.e., combinations of covered *URBs* and *SCBs*), achieving higher coverage faster.

Generally, SKI/PCT requires 100–200 more hours to reach the same *Data-race-coverage* size as *MLPCT*. While this result is very encouraging, it comes with a high start-up cost: *PIC*-5 took 240 hours of data collection and training to achieve its performance. We next turn to understanding how this start-up cost can be amortized as the kernel evolves.

5.4 Adapting to Newer Kernels

If every time a new Linux kernel comes out, we have to spend 240 hours training to save 100 hours from data-race discovery, the cost/benefit balance would not be favorable. In this section we seek to understand how little (re)training we can get away with, as we move from kernel version to kernel version, hoping to achieve an amortization point.

Name	Retrain method	Data Collection (h)	Training (h)	Total Adaptation (h)
PIC-6.ft.sml	fine tune	15.4	12.7	28.1
PIC-6.ft.med	fine tune	61.7	50.3	112.0
PIC-6.fs.sml	from scratch	15.4	12.5	27.9
PIC-6.fs.med	from scratch	61.7	49.3	111.0
PIC-5	None	0.0	0.0	0.0
PIC-5.13.ft.sml	fine tune	15.1	12.5	27.6

Table 2. Retraining time cost (hours) of *PIC* models that are used to test Linux kernel 6.1 and 5.13.

We conduct an experiment that tests a new kernel—Linux kernel 6.1 (released about 18 months after version 5.12 with numerous changes)—using 4 new *PIC* models. These 4 models are trained on newly collected training data. We collect such data by generating new CTIs for Linux kernel 6.1, selecting a number of those, exploring their interleavings, and running dynamic executions, as we did to get the original training dataset. However, the new datasets are collected in a smaller scale than the dataset for Linux kernel 5.12 since full-sized data collection and training are not favorable in incremental training settings. We use the same hyperparameters as for *PIC*-5. The validation performance of the retrained models is analyzed in §A.5. Specifically, we select the variants detailed in the top part of Table 2 (we show *PIC*-5 for comparison). Those include two *fine-tuned* variants, but also two *from-scratch* variants, where *PIC*-5 is discarded and a fresh model for Linux 6.1 is trained.

Several interesting observations are made. First, Figure 5c and Figure 5d show that fine-tuning *PIC*-5 with modest new 6.1 data and training time is a feasible and efficient approach to increase the testing effectiveness on the new kernel. In fact, considering that *MLPCT* is faster than PCT by 50–100 hours, SNOWCAT does not only find more data races in 6.1 than PCT—17% more races after a week, but also incurs a similar (with *PIC*-6.ft.med) or even lower (with *PIC*-6.ft.sml) end-to-end time cost. What's more, this amortization scales with further kernel versions and fine-tuning.

Second, the *from-scratch* variants (Figure 5e and Figure 10 in §A.5) do not perform well, since they do not have the knowledge already gleaned from many hours of training on Linux kernel 5.12, which do instruct the model usefully about the structure and semantics of kernel code, no matter what the version. In fact, *PIC*-5 performs *better* without the benefit of Linux 6.1 data (Figure 5b) than the *from-scratch* 6.1 models (Figure 5e), finding 300 more possible data races; this is a reminder that dataset size trumps all other scaling factors with large deep models [26].

Motivated by the promising results achieved by *PIC*-5 on Linux 6.1, we later conduct another experiment to test Linux kernel 5.13 (released about 2 months after 5.12) to verify the effectiveness of *PIC*-5 on a third kernel using two models. One is *PIC*-5 and another one is *PIC*-5.13.ft.sml (Table 2), which is trained by fine-tuning *PIC*-5 with a small amount of new data collected on Linux 5.13, under the same data collection and training settings as *PIC*-6.ft.sml. We configure *MLPCT* to use the S1 strategy, which shows the best overall results in the previous exploration, and run *MLPCT* under the guidance of *PIC*-5 and *PIC*-5.13.ft.sml separately but on the same CTI stream.

Figure 5f compares the *Data-race-coverage* history between *MLPCT*, using the two models, and PCT. First, both models enable *MLPCT* to outperform PCT, strengthening the advantage of the model-guided approach. Second, *PIC*-5.13.ft.sml helps *MLPCT* find possible data races faster than *PIC*-5 by up to 40 hours. However, they achieve a similar coverage level in the end. This result reveals that, when testing a new kernel that has fewer changes since Linux kernel 5.12, *PIC*-5 remains highly effective—it reaches a similar level of overall *Data-race-coverage* as the fine-tuned model while fine-tuning *PIC*-5 is more useful in terms of increasing the data race discovery speed.

5.5 Finding New Concurrency Bugs

To see if *MLPCT* helps discover new kernel concurrency bugs, we analyzed all data races found by *MLPCT* in Linux kernel 6.1. We manually pruned benign data races [42] that are annotated [28] or commented as tolerable by developers in the source code or commit messages, data races caused by synchronization primitives (e.g., locks), and data races involving only kernel variables that are not sensitive to correctness (e.g., timers). We spent about 100 person-hours total on manual inspection and reproduction.

We arrived at 14 new data races that are likely to be concurrency bugs and reported them to developers. Of those, 9 are confirmed to be bugs (3 patched), 3 are considered to be harmless and 2 are pending confirmation, as shown in Table 3a. These new bugs reside in different subsystems of the kernel and can cause data loss, wrong system-call return values, inconsistent kernel state, etc. All 9 confirmed new concurrency bugs are only found by *MLPCT*—PCT alone could not expose any of them in the time allotted to it, which shows that testing random schedules is not effective in finding new kernel concurrency bugs. A possible reason is that mainstream kernels, such as Linux, are already extensively tested under random schedules by existing fuzzing tools such as Syzkaller, which can perform basic concurrency testing by invoking system calls simultaneously in different threads.

Moreover, the effectiveness of *MLPCT* implies that prioritizing the testing of interleavings that trigger new *URB* control flows is helpful in finding new bugs. Taking bug #7 as an example (shown in Figure 6), this bug only manifests when two kernel threads run functions `vivid_fop_release()` and `vivid_ratio_rx_read()` concurrently under very complex interleavings. First, `vivid_fop_release()` must acquire and release a shared mutex lock before `vivid_ratio_rx_read()` grabs this lock (❶ → ❷). If the lock is acquired in the opposite order (❷ → ❶), the lock-protected code in `vivid_ratio_`

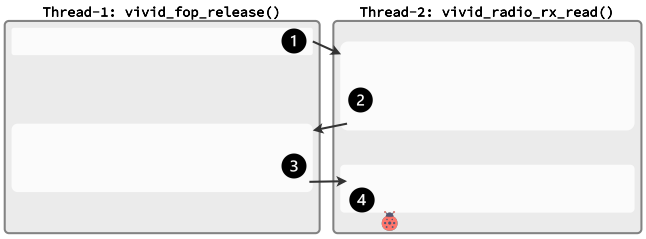


Figure 6. A concurrency bug found by SNOWCAT, which involves four shared variables and only exposes when at least three ordering constraints are satisfied. #7 in Table 3a.

`rx_read()` will execute strictly before ❶, therefore masking the bug. Second, `vivid_ratio_rx_read()` needs to update `dev->rds_owner` before `vivid_fop_release()` checks its value in the `if` statement (❷ → ❸), so that the true branch will be taken. Third, `vivid_fop_release()` should change `dev->rds_owner` to `NULL` before `vivid_ratio_rx_read()` reads it (❸ → ❹), so that `vivid_ratio_rx_read()` will be induced to call `vivid_ratio_rds_init()` again, which is a double initialization and makes `dev`'s state inconsistent.

Our analysis indicates that bug #7 has existed for over 9 years, even though testing tools (e.g., Syzkaller) have exercised the associated code extensively. The challenge in finding this bug is discovering schedules that satisfy all order constraints: ❶ → ❷, ❷ → ❸, ❸ → ❹. With testing tools that only perform random interleaving exploration (e.g., SKI), the chance of discovering this bug is extremely low. However, SNOWCAT managed to find this bug because it rejected many schedules that do not trigger new *URBs* and finally selected one that can exercise *URBs* in the code region ❸. The particular schedule consists of two scheduling hints that enforce the thread switches ❶ → ❷ and ❷ → ❸. Once the first two order constraints are satisfied, the third one can be satisfied easily. Because the thread scheduler, unless given extra scheduling hints, is unlikely to call a thread switch after the read of `dev->rds_owner` but before the write to `dev->rds_last_block` in code region ❸, code region ❹ can almost always see the `dev->rds_last_block` updated by ❸, which triggers the bug.

5.6 *PIC* Integration Case Studies

This section presents the exploration of using the *PIC* model to improve existing kernel concurrency testing tools, Razzer and Snowboard, by selecting more effective concurrent test inputs (CTIs) for execution.

5.6.1 Find race-inducing CTIs with Razzer. Razzer [25] is a kernel concurrency testing tool that uses static analysis to identify possible data races and then tries to generate CTIs to trigger the data race. Given a possible data race, Razzer will generate many sequential test inputs (STIs) through fuzzing, and then inspect their execution code coverage to find pairs of STIs that can separately execute the two data-racing instructions. Such pairs of STIs will be executed concurrently,

ID	Summary	Subsystem	Status	Found by
1	AV: do_alloc_pages() & do_alloc_pages()	sound/	**Fixed**	MLPCT
2	DR: sctp_copy_laddrs() & sctp_inet6addr_event()	net/	Benign	MLPCT
3	AV: console_callback() & kbd_event(0	drivers/	**Harmful**	MLPCT
4	DR: __inet6_bind() & udp6_seq_show()	net/	Benign	MLPCT, PCT
5	AV: snd_vmidi_output_work() & snd_virmidi_unuse()	sound/	**Harmful**	MLPCT
6	AV: proc_do_sync_threshold() & proc_do_sync_threshold()	net/	**Fixed**	MLPCT
7	AV: vivid_fop_release() & vivid_radio_rx_read()	drivers/	**Fixed**	MLPCT
8	AV: vt_do_kdskled() & vt_do_kdskled()	drivers/	Pending	MLPCT, PCT
9	AV: n_tty_lookahead_flow_ctrl() & reset_buffer_flags()	drivers/	**Harmful**	MLPCT
10	DR: sg_open() & open_wait()	drivers/	Benign	MLPCT, PCT
11	AV: vc_deallocate() & vc_cons_allocated()	drivers/	**Harmful**	MLPCT
12	DR: __d_clear_type_and_inode() & step_into()	fs/	Pending	MLPCT
13	AV: do_fb_ioctl() & fb_set_var()	drivers/	**Harmful**	MLPCT
14	AV: tracefs_apply_options() & acl_permission_check()	fs/	**Harmful**	MLPCT

(a) Testing results by *MLPCT* and PCT

15	OV: p9_conn_cancel() & p9_fd_request()	net/	**Fixed**	SB-PIC
16	DR: clone_mnt() & mnt_hold_writers()	fs/	Pending	SB-PIC
17	AV: ext4_ext_truncate() & ext4_fill_raw_inode()	fs/	**Fixed**	SB-PIC
18	AV: snd_seq_oss_midi_filemode() & snd_seq_oss_midi_open()	sound/	**Fixed**	SB-PIC
19	AV: snd_seq_oss_midi_close() & snd_seq_oss_midi_open()	sound/	**Harmful**	SB-PIC
20	DR: p9_fd_write() & p9_fd_read()	net/	Benign	SB-PIC
21	AV: ext4_fill_raw_inode() & ext4_ext_truncate()	fs/	Pending	SB-PIC

(b) Testing results by SB-PIC

Table 3. Overall testing results by SNOWCAT, which includes 17 concurrency bugs and 4 benign data races. DR denotes "data race", OV denotes "order violation [42]" and AV denotes "atomicity violation [42, 43]". Bugs confirmed as harmful are in bold type.

hoping that the two instructions will race over the shared memory.

However, if a data-racing instruction resides in a *URB* of the STI, Razzer will not even attempt to trigger the data race. Extending Razzer to allow pairs of STIs that contain the data-racing instructions in either *SCBs* or *URBs* is a feasible solution but might encourage too many test candidates. Naturally, the *PIC* model is a good option to prune this search space. We evaluate this idea using 6 known harmful data races in the Linux kernel 5.12.

We extend the approach of Razzer to Razzer-Relax which reports STI pairs as CTI candidates if the data-racing instructions reside in *SCBs* or *URBs*. Razzer-Relax is able to find more potentially useful CTIs, therefore increasing the chance of finding bugs. In addition, Razzer-PIC is designed based on Razzer-Relax. Razzer-PIC uses *PIC*-5 to evaluate CTIs identified by Razzer-Relax and only keeps CTIs that are predicted to cover the two corresponding code blocks under some random schedules. For each of the evaluated data races, we let these three variants of Razzer propose CTIs and then dynamically execute the selected CTIs with 5K random schedules using SKI to verify if the data race can be reproduced.

Table 4 presents the results of this experiment. First, it is shown that Razzer cannot reproduce 5 data races with the most conservative CTI search algorithm. This finding motivates the use of Razzer-Relax and Razzer-PIC. However, while Razzer-Relax can successfully reproduce all data races, it incurs a significant time cost. For instance, it takes over

ID	Razzer			Razzer-Relax			Razzer-PIC		
	# CTIs	# TP CTIs	Hours to reproduce (avg / worst)	# CTIs	# TP CTIs	Hours to reproduce (avg / worst)	# CTIs	# TP CTIs	Hours to reproduce (avg / worst)
A	0	0	Na / Na	73	1	17.8 / 35.8	4	1	1.2 / 2.0
B	2	0	Na / Na	76	43	0.9 / 16.7	21	21	0.5 / 0.5
C	0	0	Na / Na	139	1	34.2 / 68.1	7	1	2.0 / 3.4
D	0	0	Na / Na	195	36	2.7 / 78.4	96	22	2.1 / 36.8
E	10	0	Na / Na	322	2	52.4 / 157.4	22	1	5.7 / 10.8
F	430	1	103.8 / 210.8	1119	3	136.1 / 547.5	435	2	72.1 / 212.7

Table 4. Data race reproducing results when using Razzer, Razzer-Relax and Razzer-PIC. "# CTIs " shows the number of CTIs selected by each approach. "# TP CTIs " shows the number of true positive inputs. Worst case to identify a true positive happens if it is at the end of the schedule queue. Average time to reproduce is computed by shuffling the CTI execution queue 1000 times and averaging the time taken to reach the true positive.

547 hours to reproduce data race #F in the worst case. In contrast, Razzer-PIC can reproduce all races as Razzer-Relax but incurs a much lower time cost. In the worse case, Razzer-PIC can find all 6 bugs 15x faster than Razzer-Relax on average. On the most challenging races #C, #D, #E, and #F, Razzer-PIC can reduce the time by 22%–94%, saving hundreds of hours in total. Such results show the potential of the *PIC* model in finding error-inducing CTIs and Razzer-PIC would assist developers with bug reproduction, where the latency is crucial [9, 27, 42].

Additionally, we observe that many CTIs selected by Razzer-PIC actually trigger the two data race instructions to run in dynamic executions, which means *PIC*-5 does correctly predict the execution of their corresponding blocks. However, the target data race is not reproduced by these inputs because the two instructions triggered by them do not access the same memory, which is another requirement for the two concurrent memory accesses being a data race. This observation suggests the opportunity of training *PIC* to predict the inter-thread data flows between code blocks (§6). *PIC* trained on this task can further reduce the time for concurrency bug reproduction and possibly assist points-to-analysis on the Linux kernel, which is of limited use in practice due to the high false positive rate.

5.6.2 Better Clustering of Similar CTIs in Snowboard.

Snowboard [19] is a kernel concurrency testing framework that builds on SKI by clustering CTIs that trigger "similar" kernel behaviors using various heuristics, and then only sampling a fixed number (1 as published) of exemplar CTIs from each cluster for dynamic executions, assuming the remaining CTIs trigger similar kernel executions and therefore are unnecessary to test. Here we explore if choosing exemplars from a cluster can be improved using *PIC*. In contrast to §5.6.1, we seek to use *PIC* to only select CTIs that trigger different kernel executions, rather than CTIs that trigger a specific data race.

We first study if the amount of CTIs sampled per cluster would affect the effectiveness of Snowboard. We run Snowboard twice to test Linux kernel 6.1 with different CTI sampling sizes but the same INS-PAIR clustering strategy, which clusters CTIs by whether the two constituent STIs separately trigger a kernel instruction to respectively read from and write to a shared kernel memory region in their single-thread executions. In the first run, we use the default CTI sampling size in Snowboard—1 CTI per cluster—and find 1 new bug in Linux kernel 6.1 after testing 322,570 unique clusters. In the second run, we disable CTI sampling so that Snowboard will execute all CTIs in each cluster and we find 6 new bugs (Table 3b) after testing the same number of clusters, demonstrating that the choice of cluster exemplars might determine whether exploration will bear fruit in a fertile cluster. We call the 6 INS-PAIR clusters where the exhaustive application of Snowboard finds bugs the *6 buggy clusters*.

We consider an application of test candidate selection strategies (§3.3) on choosing exemplar CTIs from a CTI cluster, by relaxing Snowboard's one-exemplar-per-cluster policy to allow multiple samples. Specifically, whereas Snowboard chooses exemplars from the cluster at random, we invoke *PIC* for each CTI in the cluster, with a single scheduling hint that enforces the write instruction from the instruction pair to yield to the read instruction of the pair. By passing the CTI with a synthetic scheduling hint to *PIC*, we predict the coverage, and select CTIs that, cumulatively, exercise unique block coverage (S1) or increase total block coverage (S2). The selected exemplars from the two sampling approaches are then tested by the regular interleaving exploration mechanism of Snowboard. We call these sampling approaches *SB-PIC (S1)* and *SB-PIC (S2)*, and use *PIC*-6.ft.med (Table 2) for them. We compare *SB-PIC* to a relaxed Snowboard sampling approach we call *SB-RND*, which samples a fixed percentage of CTIs from the cluster.

We seek to compare how likely the two sampling approaches are to find bugs, and the amount of CTIs they need to execute using the 6 buggy INS-PAIR clusters we found above. On each buggy cluster, we run *SB-PIC* and *SB-RND* separately configured with 25%, 50% and 75% sampling percentages. If the CTIs sampled by each sampler lead Snowboard to uncover the bug, we call it a *bug-finding run*. Since this experiment is non-deterministic, we run 1000 trials for each buggy cluster and each sampling approach. We report

Bug ID	Cluster Size	Bug finding probability					# executed CTIs/ (sampling rate)				
		SB-PIC (S1)	SB-PIC (S2)	SB-RND (25%)	SB-RND (50%)	SB-RND (75%)	SB-PIC (S1)	SB-PIC (S2)	SB-RND (25%)	SB-RND (50%)	SB-RND (75%)
15	50	100%	95%	27%	53%	76%	45 (90%)	29 (58%)	12 (25%)	25 (50%)	37 (75%)
16	40	100%	66%	26%	50%	77%	40 (100%)	20 (50%)	10 (25%)	20 (50%)	30 (75%)
17	207	100%	55%	24%	48%	75%	207 (100%)	86 (41%)	51 (25%)	104 (50%)	155 (75%)
18	100	100%	100%	48%	77%	93%	100 (100%)	24 (24%)	25 (25%)	50 (50%)	75 (75%)
19	50	100%	100%	25%	50%	76%	50 (100%)	38 (76%)	12 (25%)	25 (50%)	37 (75%)
21	754	100%	50%	27%	50%	74%	743 (98%)	154 (20%)	188 (25%)	377 (50%)	565 (75%)

Table 5. Results of finding bugs using different sampling methods in Snowboard. Each Snowboard instance is repeated for 1000 times on every buggy INS-PAIR cluster. "Bug finding probability" is the number of runs in which the bug was found divided by 1000, "# executed CTIs" is the average amount of sampled/executed CTIs, and "sampling rate" is the percentage of CTIs sampled from the cluster. "Bug ID" refers to the bugs listed in Table 3b.

the percentage of *bug-finding runs* out of 1000 trials as *bug finding probability*. Additionally, we report the number of executed CTIs per cluster and the percentage of executed CTIs as sampling rate.

As reported in Table 5, *SB-PIC (S1)* shows the perfect bug finding probability. However, it is not a useful sampling approach because it often executes all CTIs in the cluster, which will incur significant testing costs. Fortunately, *SB-PIC (S2)* produces promising results. On average, it finds each of the 6 bugs with a probability of 77.6% but only needs to execute 44.8% CTIs per cluster. In contrast, *SB-RND (25%)* and *SB-RND (50%)* sample 25% and 50% CTIs per cluster but only achieve bug finding probabilities of 29.5% and 54.6% on average—*SB-PIC (S2)* is respectively 2.6x and 1.4× better.

Furthermore, when compared with *SB-RND (75%)* that has an average bug finding probability of 78.5%, *SB-PIC (S2)* requires fewer dynamic executions—only 44.8% CTIs per cluster—to achieve a similar bug finding capability (77.6%). This high efficiency is valuable as a low sampling rate can save significant testing resources in a real-world testing campaign where abundant clusters need to be tested to uncover a few buggy ones. For instance, testing 322,570 clusters using *SB-RND (50%)* and *SB-RND (75%)* would take about 5,662 and 8,443 hours, respectively, on a 30-vCPU VM. Therefore, sampling CTIs using *SB-PIC (S2)* can significantly improve the effectiveness and efficiency of Snowboard.

6 Discussion

Useful prediction tasks for concurrency testing. SNOWCAT chooses to predict the coverage of 1-hop *URBs* and *SCBs* for the concurrent test candidate. However, there might be other prediction tasks that can improve concurrency testing, such as predicting the inter-thread data flows and interrupt handler coverage. In particular, predicting the coverage of multi-hop *URBs* (e.g., 5-hop *URBs*) may provide SNOWCAT more details about the concurrent test execution. However, it is unlikely that this extension would yield significant improvements. First, 1-hop *URB* coverage is already sufficient to identify test candidates that trigger diverging kernel executions—any control flow changes during the concurrent execution will trigger 1-hop *URBs*. Second, adding multi-hop *URBs* to the concurrent test graph will greatly increase the graph size and consequently decrease the efficiency of the coverage predictor (e.g., higher training and inference cost). Thus, extending SNOWCAT to support new prediction tasks should be motivated by a study that compares the concurrency testing effectiveness of different coverage metrics.

CT graph enhancements. Adding more concurrency-related information to test graphs could help SNOWCAT train more accurate *PIC* models. For instance, information about data races that might happen when the concurrent test is executed and special code blocks that belong to kernel synchronization primitives can be encoded in the graph by adding edges of new types and adding new node types.

Guide test input and schedule generation using *PIC*. Neuzz and Design2vec use the trained model to perform input mutation. A similar algorithm can be applied on the *PIC* model to identify promising test candidates that trigger new *URBs*. For instance, the *PIC* model can suggest that certain *SCB* control flows are needed to trigger a specific *URB*. However, it is still challenging to generate sequential test inputs that can trigger arbitrary *SCB* control flows.

Predict concurrent executions on weak memory models. The *PIC* model is trained using kernel concurrent execution traces collected under the sequential memory model. While it is possible to train new *PIC* models using traces under weak memory models, it is unclear how the hardware optimization (e.g., out-of-order execution) can be represented in the concurrent test graph.

7 Related work

Kernel concurrency testing. SKI [17] takes a CTI as input and executes CTs that explore various interleavings of the CTI using the PCT algorithm [6]. Snowboard [19] generates effective CTIs by predicting the inter-thread data flows that could happen when running two STIs concurrently and prioritizes the testing of CTIs that trigger less-tested data flows. Then Snowboard exercises different interleavings of the predicted data flows to test their impact on the kernel.

Razzer [25] uses static analysis [54] to identify possible kernel data races. Razzer filters out the false positive data races using dynamic executions. It uses a fuzzing tool to find CTIs that may trigger the possible data race and executes those CTIs to check if the data race can actually happen. Krace [68] proposes a new coverage metric called alias coverage for filesystem concurrency fuzzing, which measures the pairs of instructions that touch the shared memory during the execution. It executes every randomly-generated CTI under random interleavings and then measures the alias coverage. If a CTI increases the overall alias coverage, Krace will further mutate it to generate new CTIs.

SNOWCAT differs from the aforementioned tools in that it introduces a new workflow for concurrency testing. Given the CT candidates, SNOWCAT efficiently evaluates their potential in exercising new kernel behaviors using the coverage predictor and only selects the more promising CTs for dynamic executions. As shown in §5.3 and §5.6, the new workflow helps SNOWCAT outperform SKI, Snowboard and Razzer with higher testing effectiveness and efficiency.

Kernel testing. Feedback-based fuzzing has been shown to be effective in generating STIs and finding kernel bugs. Syzkaller [20] tries to maximize the code/edge coverage of the kernel using a feedback-based mechanism—it keeps mutating STIs that can increase the coverage. Moonshine [46] improves Syzkaller by extracting system call sequences from real-world applications. HFL [30] uses symbolic executions to guide the mutation of STIs. SNOWCAT benefits from the development of such tools in that more effective CTIs can be generated from better STIs.

Machine learning for software testing. NEUZZ [50] trains a neural network model to predict the application edge coverage given a test input. Then NEUZZ uses the trained model to guide input mutation—it computes the model gradients to find out which part of the test input needs to be mutated to increase the coverage. FuzzGuard [72] explores the use of ML for directed fuzzing, in which only a specific set of code blocks (target blocks) are interested rather than all blocks. FuzzGuard trains a model to learn the reachability of target blocks given a test input and then uses the model to predict and skip inputs that cannot hit target blocks.

Design2vec [56] uses a GCN model to predict the coverage of the hardware. In addition to the test input, Design2vec inputs the whole control data flow graph of the hardware in RTL. Along a similar vein, ProGraML [10] uses a graph representation of LLVM IR, at a finer granularity (individual instructions) including data- and control-flow edges, towards predicting static properties of code. SNOWCAT uses a similar model architecture as Design2vec, and a little coarser-granularity than ProGraML (basic blocks). However, SNOWCAT can take the scheduling information of the concurrent test into consideration and predict the coverage of the test input when it is executed under a specific interleaving.

Machine learning for kernel testing. SyzVegas [62] uses reinforcement learning to schedule different kernel fuzzing tasks (e.g., test generation/mutation/selection), which otherwise would be scheduled under fixed manually-written policies. It proposes a reward assessment model to learn the costs and benefits of different fuzzing tasks over time and then makes better arrangements of tasks in the following runs. SNOWCAT is in general orthogonal to SyzVegas as it predicts the concurrent kernel executions and improves the test selection using the predicted coverage.

Healer [55] proposes an algorithm to learn the system call influence relations—the influence of a system call A on the execution path of another system call B. It finds such relations by running STIs in which system call B is called right after A or without A and comparing the coverage of these STIs. The system call A is concluded to have influence on the call B if A helps B trigger new coverage. Healer keeps learning influence relations and generating new STIs that encourage learned influence relations to test more kernel execution paths. Compared to Healer, SNOWCAT learns the influence between interleaved instructions triggered by concurrently-running system calls and predicts their coverage, which is more challenging and requires more efficient and accurate approaches such as deep learning.

8 Conclusion

This work introduces SNOWCAT, a kernel concurrency testing framework that uses a kernel code block coverage predictor for identifying and prioritizing interesting concurrent test candidates. The coverage predictor is achieved via a GNN model named *PIC* that takes the concurrent test input and scheduling hints and predicts if certain concurrency-sensitive blocks would be executed or not. The coverage predictor enables SNOWCAT to use a new testing workflow in which new concurrency test candidates are evaluated based on the predicted coverage and only executed dynamically if they are interesting (e.g., cover a new set of code blocks). The evaluation of SNOWCAT shows that this workflow is both effective and efficient. SNOWCAT can find more potential data races, reproduce known bugs quickly and find new bugs with high probabilities. More importantly, the coverage predictor can generalize across different kernel versions, showing SNOWCAT can scale well to rapidly-evolving kernels.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Shan Lu, for their insightful feedback. We also thank Stefan Bucur, David Culler, Franjo Ivancic, Phil Levis, Jeff Mogul, and the Reliable and Secure System Lab members for their detailed and helpful comments on this work and earlier drafts. This work was funded in part by the National Science Foundation (NSF) under grants CNS-2140305 and CNS-2145888, Google, and Intel.

References

[1] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2435–2452. https://www.usenix.org/conference/usenixsecurity21/presentation/abubakar

[2] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 2021. Kard: Lightweight Data Race Detection with per-Thread Memory Protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 647–660. https://doi.org/10.1145/3445814.3446727

[3] Darrell Anderson. 2002. *Fstress: A Flexible Network File Service Benchmark*. Technical Report.

[4] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2020. Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 723, 12 pages.

[5] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks? arXiv:2105.14491 [cs.LG]

[6] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 167–178. https://doi.org/10.1145/1736020.1736040

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[8] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3133956.3134069

[9] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 17–32. https://www.usenix.org/conference/osdi18/presentation/weidong

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*.

[12] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[13] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 151–162.

[14] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[15] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding Complex Concurrency Bugs in Large Multi-Threaded Applications. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/1966445.1966465

[16] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. 2010. A study of the internal and external effects of concurrency bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 221–230. https://doi.org/10.1109/DSN.2010.5544315

[17] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 415–431.

[18] Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2023. Probabilistic Concurrency Testing for Weak Memory Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 603–616. https://doi.org/10.1145/3575693.3575729

[19] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-Thread Communication Analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 66–83. https://doi.org/10.1145/3477132.3483549

[20] Google. 2015. Syzkaller-kernel fuzzer. https://github.com/google/syzkaller

[21] Google. online. Kernel Thread Sanitizer (KTSAN). https://github.com/google/ktsan/wiki Accessed: 7 May 2021.

[22] Sun Hao, Shen Yuheng, Liu Jianzhong, Xu Yiru, and Jiang Yu. 2022. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 351–366. https://www.usenix.org/conference/atc22/presentation/sun

[23] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. 2022. Demystifying the Dependency Challenge in Kernel Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 659–671. https://doi.org/10.1145/3510003.3510126

[24] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. 2020. Strategies for Pre-training Graph Neural Networks. arXiv:1905.12265 [cs.LG]

[25] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP.2019.00017

[26] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[27] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China)

(SOSP '17). Association for Computing Machinery, New York, NY, USA, 582–598. https://doi.org/10.1145/3132747.3132767

[28] Linux Kernel. online. The Kernel Concurrency Sanitizer (KCSAN). https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html Accessed: 7 May 2021.

[29] Michael Kerrisk. online. syscalls(2) — Linux manual page. https://man7.org/linux/man-pages/man2/syscalls.2.html Accessed: 7 May 2021.

[30] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[31] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[32] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*. ICLR. https://openreview.net/forum?id=SJU4ayYgl

[33] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (dec 2017), 32 pages. https://doi.org/10.1145/3158105

[34] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 162–180. https://doi.org/10.1145/3341301.3359638

[35] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/osdi-04/cp-miner-tool-finding-copy-paste-and-related-bugs-operating-system-code

[36] Congyu Liu, Sishuai Gong, and Pedro Fonseca. 2023. KIT: Testing OS-Level Virtualization for Functional Interference Bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 427–441. https://doi.org/10.1145/3575693.3575731

[37] Jian Liu, Lin Yi, Weiteng Chen, Chengyu Song, Zhiyun Qian, and Qiuping Yi. 2022. LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 125–142. https://www.usenix.org/conference/usenixsecurity22/presentation/liu-jian

[38] Ilya Loshchilov and Frank Hutter. 2016. Stochastic Gradient Descent with Warm Restarts (SGDR): Communication Efficient Distributed Optimization using Quantization. *arXiv preprint arXiv:1608.03983* (2016).

[39] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. A Study of Interleaving Coverage Criteria. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers* (Dubrovnik, Croatia) *(ESEC-FSE companion '07)*. Association for Computing Machinery, New York, NY, USA, 533–536. https://doi.org/10.1145/1295014.1295034

[40] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. A Study of Interleaving Coverage Criteria. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 533–536. https://doi.org/10.1145/1287624.1287703

[41] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 103–116. https://doi.org/10.1145/1294261.1294272

[42] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[43] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1168857.1168864

[44] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. Checker: A Soundy Analysis for Linux Kernel Drivers. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 1007–1024.

[45] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.

[46] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[47] Benedek Rozemberczki, Peter Englert, Amol Kapoor, Martin Blais, and Bryan Perozzi. 2021. Pathfinder Discovery Networks for Neural Message Passing. arXiv:2010.12878 [cs.LG]

[48] Nicola Ruaro, Kyle Zeng, Lukas Dresel, Mario Polino, Tiffany Bao, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. 2021. SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID '21)*. Association for Computing Machinery, New York, NY, USA, 456–468. https://doi.org/10.1145/3471621.3471865

[49] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[50] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 803–817. https://doi.org/10.1109/SP.2019.00052

[51] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. 2021. Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification. arXiv:2009.03509 [cs.LG]

[52] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17

[53] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2541–2557. https://www.usenix.org/conference/usenixsecurity20/presentation/song

[54] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

[55] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3477132.3483547

[56] Shobha Vasudevan, Wenjie (Joe) Jiang, David Bieber, Rishabh Singh, hamid shojaei, C. Richard Ho, and Charles Sutton. 2021. Learning Semantic Representations to Verify Hardware Designs. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 23491–23504. https://proceedings.neurips.cc/paper/2021/file/c5aa65949d20f6b20e1a922c13d974e7-Paper.pdf

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[58] Martin Vechev, Eran Yahav, and Greta Yorsh. 2009. Experience with Model Checking Linearizability. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software* (Grenoble, France). Springer-Verlag, Berlin, Heidelberg, 261–278. https://doi.org/10.1007/978-3-642-02652-2_21

[59] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML]

[60] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 205–214. https://doi.org/10.1145/1287624.1287654

[61] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2741–2758. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

[62] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2741–2758. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

[63] Wikipedia contributors. 2020. Average Precision — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Information_retrieval&oldid=793358396#Average_precision.

[64] Wikipedia contributors. 2020. F-score — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/F-score.

[65] Wikipedia contributors. 2020. Precision and recall — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Precision_and_recall.

[66] Wikipedia contributors. 2022. Register-transfer level — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Register-transfer_level.

[67] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. 2007. Model Checking Concurrent Linux Device Drivers. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 501–504. https://doi.org/10.1145/1321631.1321719

[68] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

[69] Peiqi Yin, Xiao Yan, Jinjing Zhou, Qiang Fu, Zhenkun Cai, James Cheng, Bo Tang, and Minjie Wang. 2022. DGI: Easy and Efficient Inference for GNNs. arXiv:2211.15082 [cs.LG]

[70] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. 2020. UBITect: A Precise and Scalable Method to Detect Use-before-Initialization Bugs in Linux Kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 221–232. https://doi.org/10.1145/3368089.3409686

[71] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 540–555. https://doi.org/10.1145/3447786.3456258

[72] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong