Lazy Testing of Machine-Learning Models

Anastasia Isychev[1] , **Valentin Wüstholz**[2] , **Maria Christakis**[1]
[1]TU Wien, Austria
[2]Consensys
{anastasia.isychev, maria.christakis}@tuwien.ac.at, valentin.wustholz@consensys.net

Abstract

Checking the reliability of machine-learning models is a crucial, but challenging task. NOMOS is an existing, automated framework for testing general, user-provided functional properties of models, including so-called *hyperproperties* expressed over more than one model execution. NOMOS aims to find model inputs that expose "bugs", that is, property violations. However, performing thousands of model invocations during testing is costly both in terms of time and money (for metered APIs, such as OpenAI's).

We present LAZ (pronounced "lazy"), an extension of NOMOS that automatically minimizes the number of model invocations to boost the test throughput and thereby find bugs more efficiently. During test execution, LAZ automatically identifies *redundant* invocations—invocations where the model output does not affect the final test outcome—and skips them, much like lazy evaluation in certain programming languages. This optimization enables a second one that dynamically reorders model invocations to skip the *more expensive* ones. As a result, LAZ finds the same number of bugs as NOMOS, but does so median 33% and up to 60% faster.

1 Introduction

Machine-learning (ML) models are an integral part of everyday life, for instance by classifying street signs in self-driving cars or natural-language commands in smart homes. As more and more tasks depend on ML models, it is increasingly important to validate their reliability. For this reason, there is a significant amount of work on checking or ensuring robustness and fairness properties of ML models, e.g., [Huang *et al.*, 2017; Gehr *et al.*, 2018; Singh *et al.*, 2019; Albarghouthi *et al.*, 2017; Bastani *et al.*, 2019; Urban *et al.*, 2020; Carlini and Wagner, 2017; Goodfellow *et al.*, 2015; Madry *et al.*, 2018; Galhotra *et al.*, 2017; Udeshi *et al.*, 2018; Tramèr *et al.*, 2017; Athavale *et al.*, 2024].

NOMOS. Recently, Christakis et al. [Christakis *et al.*, 2023] presented NOMOS, an automated framework for expressing and testing functional-correctness properties of models, subsuming more specific properties such as robustness or fairness. In particular, NOMOS provides a specification language for expressing such properties, including so-called *hyperproperties* [Clarkson and Schneider, 2008], expressed over more than one model execution. For example, a hyperproperty for a model that predicts whether a person should get a loan could be: *if the model predicts that a person should get the loan and their salary increases, then the model should still predict that they get the loan*. This is a 2-safety property because checking its correctness requires two model invocations, one before the salary increase and one after.

NOMOS also includes a fully automated framework (based on metamorphic testing [Chen *et al.*, 1998; Segura *et al.*, 2016]) for validating its specifications. On a high level, the framework takes as input the model under test and a set of specified properties that should hold for the model. The properties are then compiled into a test harness, that is, code that tests the model against the properties. Specifically, the harness generates inputs for the model (using metamorphic testing), calls the model with these inputs, and evaluates whether the properties hold. Note that each test targeting a k-safety property needs to invoke the model k times. For example, for the loan predictor above, a single test would have to call the model for a person with salary, say, 50K, and then again, for the same person but with salary, say, 55K. As output, the framework produces the failing tests, i.e., those for which the model violated the specified properties.

LAZ. To effectively test each k-safety property, NOMOS generates thousands of tests, each invoking the model k times. However, such a high number of invocations is costly, both in terms of runtime and money (for metered APIs, such as OpenAI's). In this paper, we alleviate this issue by presenting LAZ (pronounced "lazy"), an extension of NOMOS that uses *static analysis* to automatically minimize the number of model invocations, and consequently, boost the test throughput (tests per second).

More precisely, during test execution, LAZ uses a form of static analysis called *abstract interpretation* [Cousot and Cousot, 1977] to automatically identify and skip *redundant* model invocations, that is, invocations where the model output does not affect the final test outcome. In other words, model invocations are performed lazily depending on whether their outputs may affect the target hyperproperty, much like lazy evaluation in certain programming languages. This optimization enables a second one that dynamically reorders model

invocations such that the more expensive ones are skipped more frequently.

We evaluate the effectiveness of LAZ by using it to test ML models from five distinct domains, namely, tabular data, image classification, speech-command classification, sentiment analysis from natural language, and action policies. Compared to NOMOS, when given the same test budget, LAZ finds the same number of bugs, but it increases the test throughput by median 33% (and up to 60%) and skips median 30% of redundant model invocations (and up to 47%).

Contributions. Overall, this paper makes the following contributions:

- We present a novel static-analysis technique for lazily testing hyperproperties of ML models.
- We implement our technique in the publicly available[1] tool LAZ.
- We demonstrate the effectiveness of LAZ in increasing test throughput across a wide range of domains.

Outline. Our paper is organized as follows. In the following section, we give the necessary background on NOMOS. In Sects. 3 and 4, we describe how LAZ skips and reorders model invocations. Sect. 5 describes implementation aspects of LAZ. We present our experimental evaluation in Sect. 6, discuss related work in Sect. 7, and conclude in Sect. 8.

2 Background on NOMOS

In general, a NOMOS specification adheres to a particular structure that defines:

- a *precondition* using zero or more **`requires`** statements. The precondition expresses the conditions under which the model should be invoked. Each condition is a user-provided relation involving zero or more model inputs.
- the *body*, which may be arbitrary Python code invoking the model under test.
- a *postcondition* using zero or more **`ensures`** statements. The postcondition constitutes the target hyperproperty and expresses the conditions that should hold after executing the body; these conditions may also refer to model outputs, unlike for preconditions.

We show two example NOMOS specifications in Fig. 1.

MNIST. Consider a model trained over the MNIST dataset [LeCun *et al.*, 1999] to classify images of handwritten digits from 0 to 9. An example hyperproperty could be: *if the model correctly classifies a blurred image, then the model should also correctly classify its original (i.e., not blurred) version*. This 2-safety property is shown in Fig. 1a.

For an original input image `x1` (line 1), line 2 creates a blurred version `x2`. Variable `v1` on line 3 is assigned the correct label for `x1`. Lines 4–5 declare two outputs, `d1` and `d2`, which are assigned the model's prediction when calling it with `x1` and `x2`, respectively (see body on lines 7–10). (Line 6 should be ignored for now.) The postcondition on line 11 expresses the above hyperproperty, that if the blurred image

[1] https://github.com/Rigorous-Software-Engineering/LaZ

```
1  input x1;
2  var x2 := blur(x1);
3  var v1 := label(x1);
4  output d1;
5  output d2;
6  returns predict [0,9];
7  {
8    d1 = predict(x1)
9    d2 = predict(x2)
10 }
11 ensures d2==v1 ==> d1==v1;
```

(a) 2-safety property for MNIST.

```
1  input s1;
2  var s2 := relax(s1);
3  output o1;
4  output o2;
5  returns play [0,1];
6  {
7    o1, o2 = 0, 0
8    rs = []
9    for _ in range(10):
10     rs.append(randInt(0, MAX_INT))
11   for i in range(10):
12     o1 += play(s1, rs[i])
13     o2 += play(s2, rs[i])
14 }
15 # 0-lose, 1-win
16 ensures o1 <= o2;
```

(b) 20-safety property for LunarLander.

Figure 1: Example hyperproperties specified in NOMOS.

`x2` is correctly classified, the original image `x1` should also be correctly classified.

LunarLander. Now, consider the LunarLander action policy, a Gym environment [Brockman *et al.*, 2016] where the goal of a spacecraft is to land on an uneven surface. Landing becomes easier by increasing the initial distance between the spacecraft and the surface, therefore giving the spacecraft more time to land. So, an example hyperproperty could be: *if the spacecraft lands successfully, then increasing the distance to the surface should also result in landing successfully*. However, the environment is stochastic, and to account for any randomness, the hyperproperty could also be cumulative: *if the spacecraft lands successfully x times out of n, then increasing the distance to the surface should result in landing successfully at least x times out of n*. The latter property is shown in Fig. 1b.

Line 1 declares an input `s1`, which is an initial state of the landing game. Line 2 "relaxes" `s1` to produce `s2`, which is the initial state of an easier version of the same landing game obtained by increasing the distance to the surface. (Line 5 should be ignored for now.) The body (lines 6–14) initializes outputs `o1` and `o2` and plays the game from `s1` and `s2` in a loop (lines 11–13) while accumulating the number of wins in `o1` and `o2`. To account for stochasticity in the environment, each loop iteration uses a different environment random seed (from array `rs` initialized on lines 8–10). As expected, the

```
1  while budget > 0:
2    # input generation
3    s1 = randState()
4    s2 = relax(s1)
5    # body
6    o1, o2 = 0, 0
7    rs = []
8    for _ in range(10):
9      rs.append(randInt(0, MAX_INT))
10   for i in range(10):
11     o1 += play(s1, rs[i])
12     o2 += play(s2, rs[i])
13   # postcondition
14   if o1 <= o2 :
15     passed += 1
16   else:
17     postcond_violtn += 1
18     process_bug()
19   budget -= 1
```

Figure 2: Snippet of test harness generated by NOMOS from Fig. 1b.

postcondition on line 16 expresses that the number of wins for the easier game should at least match the number of wins for the original game. Note that the property depends on 20 model invocations (2 in each of 10 loop iterations) and is, therefore, a 20-safety property.

NOMOS specifications are automatically compiled into a test harness, i.e., code that tests the model against the properties. Fig. 2 shows a snippet of the test harness generated by NOMOS from the specification of Fig. 1b. NOMOS executes a user-specified number of tests, stored in `budget` (line 1). Each test consists in generating a random input `s1` and relaxing it into `s2` (lines 3–4), running the body of the specification with these inputs (lines 6–12), and checking whether a bug is found (lines 14–18), where a bug is a postcondition violation. Any found bugs are processed for de-duplication.

As output, NOMOS provides the unique failing tests, that is, those for which the postcondition is violated.

3 Skipping Redundant Model Invocations

This section describes how LAZ skips redundant model invocations while running a test harness such as the one in Fig. 2. A model invocation is considered *redundant* if, no matter what the model returns, the final test outcome is not affected. In other words, whether the postcondition holds or fails does not depend on the output of that particular model invocation.

To determine whether an invocation is redundant, LAZ uses a form of static analysis called *abstract interpretation* [Cousot and Cousot, 1977]. Abstract interpretation abstracts the program state into elements of *abstract domains*. We use the Intervals domain [Cousot and Cousot, 1976], which abstracts numerical variables into intervals of their possible values.

In particular, when generating a test harness from a NOMOS specification, LAZ inserts a redundancy check before each model invocation, e.g., before line 11 and 12 of Fig. 2. For each of these checks, LAZ uses an abstract interpreter to statically compute the range of all possible values for the variables that affect the postcondition (e.g., `o1` and `o2` in the case of LunarLander). The analyzer then statically checks the postcondition with these ranges. More specifically, if the abstract interpreter can already *verify* that the postcondition holds (respectively, fails) for all possible values of its variables, then the specific output of each upcoming model invocation cannot influence the test outcome. As a result, these invocations may be soundly skipped at runtime.

LAZ builds on off-the-shelf abstract interpreters by transforming NOMOS specifications into programs supported by such analyzers. As a result, we can directly benefit from decades of static-analysis research and existing high-performance implementations.

LunarLander. Next, we show LAZ in action for the test harness of Fig. 2, which calls the model 20 times.

Before the first model invocation on line 11 of Fig. 2 (in the first of 10 loop iterations), LAZ generates the Python program of Fig. 3a to reflect what is yet to run for this test. This program will be then passed to the abstract interpreter for analysis, and it is generated in the following four steps.

First, all remaining model invocations are replaced by calls to **`nondet`**, which simulate that the model may non-deterministically return any possible value (lines 3–4). In the case of LunarLander, the model may return any value in the range $[0, 1]$, which is encoded as **`nondet`**`(0,1)` in Fig. 3a.

Note that we extend the NOMOS specification language to express the range of possible values returned by a model, e.g., see line 5 of Fig. 1b. Specifically, we add the statement

$$\textbf{returns}\ fn\ [low, hi];$$

where fn is a function name and $[low, hi]$ is the range of all possible integer values that the function returns. This interval syntax is sufficiently expressive for our purposes since integers can always be used as indices in an array containing all possible (non-integer) return values.

Second, after replacing the model invocations, any unused variables are removed, e.g., `s1`, `s2`, and `rs` in the case of LunarLander. *Third, all remaining variables*, in this case `o1` and `o2`, *are assigned the value they have at runtime right before the model invocation that triggered the analysis*. These assignments appear at the beginning of the generated program (line 1 of Fig. 3a). As another example, consider the MNIST specification of Fig. 1a. For this specification and its corresponding test harness, the variables remaining after replacing the model invocations are `d1`, `d2` as well as `v1`, which refers to the result of calling **`label`**`(x1)`.

Finally, the postcondition p is encoded using a non-deterministic if-statement with a branch that asserts p and another that asserts $\neg p$. For instance, lines 6–9 of Fig. 3a encode the postcondition from Fig. 1b: line 7 asserts the postcondition itself, and line 9 asserts its negation.

Note that, in static analysis, assertions are proof obligations, that is, the analyzer is asked to *verify* their validity. The non-deterministic if-statement simply instructs the analyzer to try to verify each assertion *independently* of each other. If either of them is verified (i.e., if the postcondition *always holds* or *always fails*), then all upcoming model invocations are redundant and, thus, skipped. If the assertions remain unverified, it means that the test outcome cannot yet be determined; it still

```
o1, o2 = 0, 0
for _ in range(10):
  o1 = o1 + nondet(0,1)
  o2 = o2 + nondet(0,1)

if nondet(0,1) == 0:
  assert (o1 <= o2)
else:
  assert not(o1 <= o2)
```

(a) Before the first model invocation.

```
1 o1, o2 = 6, 2
2 o2 = o2 + nondet(0,1)
3 for _ in range(2):
4   o1 = o1 + nondet(0,1)
5   o2 = o2 + nondet(0,1)
6
7 if nondet(0,1) == 0:
8   assert (o1 <= o2)
9 else:
10  assert not(o1 <= o2)
```

(b) Before the 16th model invocation.

Figure 3: Example programs passed to the abstract interpreter when testing LunarLander with the harness of Fig. 2.

depends on the concrete values returned by the model, and no model invocations are skipped yet. The latter is exactly the case for the program of Fig. 3a.

Now, assume we execute 15 (out of 20) model invocations without finding any of them to be redundant. Let us also assume that until this point the model has won six games starting from `s1` and two games starting from `s2`. Before the 16th model invocation on line 12 of Fig. 2 (in the eighth of 10 loop iterations), LAZ generates the program of Fig. 3b to reflect what is yet to run for this test. As before, the model invocations are replaced by calls to **`nondet`**. Moreover, the number of outstanding model invocations is updated on lines 2–3, and variables `o1` and `o2` are assigned 6 and 2, respectively, on line 1—these are their values before the 16th model invocation, which triggered the analysis.

In this scenario, `o2` can become at most 5 by the end of the test. Since `o1` has value 6 and does not decrease, the postcondition `o1 <= o2` can never be satisfied. In fact, the analyzer verifies the assertion on line 10 of Fig. 3b by evaluating the intervals for `o1` and `o2` as

$$o1 = [6,6] + 2 * [0,1] = [6,8]$$

$$o2 = [2,2] + 3 * [0,1] = [2,5]$$

and proving that $[6,8] > [2,5]$.

As a result, LAZ now informs NOMOS that the remaining five model invocations of this test may be skipped. By skipping these invocations, the test directly proceeds to the postcondition evaluation, and a bug is reported.

This optimization enables a second one that dynamically reorders the model invocations such that the more expensive ones are skipped more frequently. For instance, in the LunarLander example, we skipped three invocations from relaxed state `s2` and two from harder state `s1`. If the invocations from `s1` are more expensive, reordering lines 11 and 12 of Fig. 2 could have allowed skipping more such invocations (and fewer from `s2`).

4 Reordering Model Invocations

Given a NOMOS specification, it is difficult to statically determine the most effective order of model invocations in terms of test throughput; that is, in terms of skipping more expensive invocations more frequently and thereby increasing the number of executed tests per second. However, during a testing campaign with thousands of tests, the most effective order could be dynamically observed from a small subset of tests and then used for the rest of the campaign.

This is exactly what LAZ does. In particular, the budget of a testing campaign (e.g, line 1 of Fig. 2 in the case of LunarLander) is split into an *exploration* and an *exploitation* phase. The initial exploration phase tries all model-invocation orders for a small fraction of the budget. In practice, trying all invocation orders simply involves a couple of syntactic variations of the original harness. For instance, syntactically speaking, the harness of Fig. 2 contains two model invocations. In this case, the exploration phase would observe the performance for tests where the model invocations occur as they appear in the figure (which is exactly how they appear in the specification of Fig. 1b) as well as for tests where the model invocations occur in reverse order (i.e., by swapping lines 11 and 12 of Fig. 2).

Next, the exploitation phase uses the observed performance characteristics to select the most effective invocation order for the rest of the campaign budget. More specifically, LAZ selects the order with the highest test throughput and resolves any ties by selecting the order with the highest number of skipped invocations. This design can be viewed as an instantiation of ϵ-first policies for multi-armed bandits [Tran-Thanh *et al.*, 2010].

The high-level algorithm for the exploration phase in LAZ is shown in Alg. 1. The algorithm takes four input parameters: (1) the NOMOS specification, (2) the budget for the testing campaign, (3) the percentage of tests to be used for exploration, and (4) the margin (in percent) of what constitutes a meaningful difference in running time across different invocation orders. The last parameter ensures that small fluctuations in running time are not decisive when selecting the best invocation order. The algorithm returns the configuration with the best invocation order.

First, the algorithm determines the number of all possible configurations (lines 1–3). Next, it computes the exploration budget per configuration by splitting the total exploration budget equally across configurations (lines 4–5). For each configuration, the algorithm performs the exploration and records the running time as well as the number of redundant model invocations (lines 6–13). Finally, it selects the most effective configuration, which is then used in the exploitation phase (lines 14–17).

Algorithm 1 The LAZ exploration phase

Input: spec, // NOMOS *specification*
testBudget, *// total number of tests in the campaign*
explorePt, *// percentage of tests to use for exploration*
marginPt *// time margin (in percent) across orders*
Output: bestCfg *// configuration with best invocation order*

```
1: // Compute number of possible invocation orders
2: numInvs = GETMODELINVS(spec)
3: numCfgs = numInvs!
4: // Compute number of tests per order
5: numTests = (testBudget * explorePt) / numCfgs
6: // Explore all invocation orders
7: times = []; skipInvs = []
8: for i = 0 to numCfgs:
9:    ord = GETINVORDER(i)
10:   // Record time and number of skipped invocations
11:   t, s = RUNTESTSWITHCFG(spec, numTests, ord)
12:   times += [t]; skipInvs += [s]
13: end for
14: // Choose the fastest invocation order by
15: // resolving ties with number of skipped invocations
16: bestCfg = GETBESTCFG(times, marginPt, skipInvs)
17: return bestCfg
```

5 Implementation

We implemented LAZ in Python as an extension of the NOMOS framework. For the static analysis, we use the MOPSA abstract interpreter [Journault *et al.*, 2019] with the default Intervals domain and a setting to unroll all loops. Note that MOPSA may be replaced by any other analyzer that can precisely reason about intervals and is sufficiently fast. To minimize the overhead of the abstract interpreter, LAZ caches all analysis queries as well as their results and uses the cache throughout the testing campaign.

The exploration phase of LAZ can be extended to include additional configurations. For instance, if model invocations are very fast, the analysis overhead may be too high to be beneficial. In such cases, LAZ can include a configuration that disables the analysis during the exploration phase; if the analysis overhead indeed turns out to be prohibitive, the analysis is automatically disabled during the exploitation phase. We experiment with such a setup and present our results in the next section.

6 Experimental Evaluation

We evaluate the effectiveness of LAZ by focusing on the following research questions:

RQ1: Does LAZ improve test throughput?

RQ2: Does the exploration phase improve test throughput?

RQ3: How significant is the analysis overhead?

RQ4: What is the effect of LAZ's hyperparameters?

6.1 Benchmarks

We use LAZ to test models from a variety of different domains, involving tabular data (GermanCredit [Hofmann, 1994] and COMPAS [Larson *et al.*, 2016]), images (MNIST [LeCun *et al.*, 1999]), natural language (HotelReview [Liu, 2017]), speech (SpeechCommand [Warden, 2018]), and action policies (LunarLander and BipedalWalker [Brockman *et al.*, 2016]). We use the pre-trained models and the hyperproperties specified for these models from NOMOS [Christakis *et al.*, 2023]—there is a total of 32 specifications.

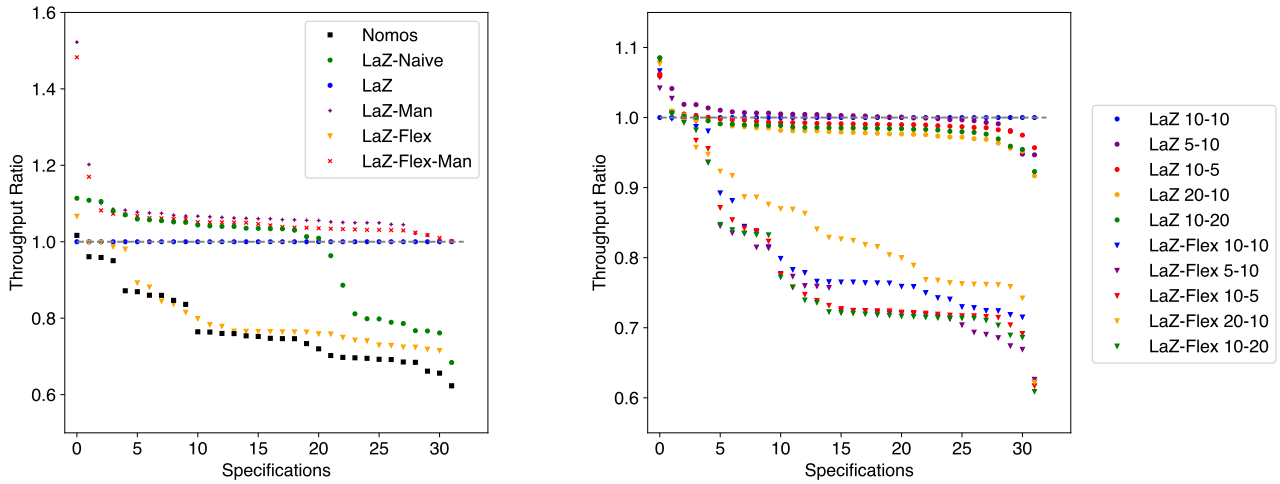
6.2 Setup

We run our experiments using NOMOS and the following variants of LAZ:

1. **NOMOS**: the testing framework by Christakis et al. that eagerly performs all model invocations;
2. **LAZ-NAIVE**: LAZ without the exploration phase; it performs all model invocations in the order they appear in the specification and skips redundant invocations;
3. **LAZ**: default LAZ with the exploration phase; it uses the best model-invocation order during the exploitation phase and skips redundant invocations;
4. **LAZ-MAN**: LAZ where all queries to the static analyzer are replaced with manually derived conditions that identify redundant model invocations with minimal overhead; it provides an upper bound on the test throughput that could only be achieved if the static analysis had practically no performance overhead;
5. **LAZ-FLEX**: LAZ with an additional configuration in the exploration phase that determines whether it pays off to enable the static analysis in the exploitation phase; if it is determined that the analysis should be disabled, no model invocations are skipped in the exploitation phase; if the analysis should be enabled, it also uses the best model-invocation order;
6. **LAZ-FLEX-MAN**: LAZ-FLEX but with the manually derived conditions that identify redundant calls with minimal overhead.

Hyperparameters. For variants 3–6 that include the exploration phase, we compare several hyperparameter settings. The default setting uses 10% of the tests for exploration (parameter $explorePt$ from Alg. 1) and a 10% margin to decide the winning configuration (parameter $marginPt$ from Alg. 1). To evaluate this setting, we independently double and halve each of the two parameters to obtain the following $explorePt$–$marginPt$ settings expressed in percent: 5–10, 10–5, 20–10, 10–20. We compare the default 10–10 setting with these.

Compared to LAZ and LAZ-MAN, LAZ-FLEX and LAZ-FLEX-MAN have one more configuration to explore (the one where the static analysis is disabled). So, given the same $explorePt$ budget, the number of tests per configuration would be smaller for LAZ-FLEX and LAZ-FLEX-MAN than for LAZ and LAZ-MAN. To ensure that we do not favor the exploration phase of LAZ and LAZ-MAN, we use the same number of tests per configuration during the exploration phase across all the above variants. As a result, LAZ and LAZ-MAN scale down the specified exploration budget by a factor of $\frac{numInvs!}{numInvs!+1}$.



(a) NOMOS and LAZ variants with default 10–10 setting.

(b) LAZ and LAZ-FLEX with different $explorePt$–$marginPt$ settings.

Figure 4: Relative test throughput with respect to LAZ 10–10.

Testing campaigns. For each testing campaign, we use a budget of 1000 tests. To account for fluctuations in running time due to randomness in the testing process, we run each experiment with 5 different random seeds.

Hardware. We run all experiments sequentially (no parallelism) on a machine with a Quadro RTX 8000 GPU and an Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz.

6.3 Results

RQ1: Test throughput. We compare the test throughput across NOMOS and all LAZ variants with the default 10–10 setting. Fig. 4a shows their relative throughput with respect to LAZ on the y-axis and the 32 NOMOS specifications against which we tested on the x-axis. For each tool variant, the points in the graph are sorted by throughput; therefore, two points with the same x-coordinate do not necessarily refer to the same specification. All points above the $y = 1$ line exhibit higher throughput than LAZ (which is better), and all points below this line have lower throughput.

We observe that LAZ *outperforms* NOMOS *for all but one specification;* LAZ *is from 4% to 60% faster for 31 specifications.* For the one specification where NOMOS outperforms LAZ, NOMOS is 2% faster. Overall, the median throughput increase of LAZ over NOMOS is 33%. LAZ-NAIVE is from 1% to 10% faster than LAZ for 21 specifications, but from 4% to 46% slower than LAZ for 11 specifications. However, LAZ-NAIVE already consistently outperforms NOMOS. LAZ-FLEX is consistently worse than LAZ, showing that always having the static analysis enabled is more efficient. Similarly, LAZ-FLEX-MAN is almost consistently outperformed by LAZ-MAN, though the difference between them is under 4%. Besides 2 outliers, LAZ's throughput is within 10% of the ideal throughput achieved by LAZ-MAN, which skips the model invocations using precise, manually crafted conditions for each specification.

RQ2: Exploration phase. To evaluate the effectiveness of the exploration phase on the test throughput, we take a closer look at LAZ-NAIVE and LAZ in Fig. 4a. As mentioned in RQ1, the exploration phase either significantly improves throughput (up to 46%), or incurs an overhead (at most 10%).

For 11 out of 32 specifications, LAZ selects the model-invocation order that already appears in each specification for all 5 random seeds (and for 6 additional specifications, LAZ selects this order for some random seeds). Consequently, for these specifications, LAZ-NAIVE happens to always execute the best order of model invocations without the overhead of the exploration phase, which explores all orders. However, if the order that appears in a specification turns out to be worse, LAZ-NAIVE uses the more expensive order for the entire testing campaign, resulting in significantly worse performance.

Overall, LAZ*'s exploration phase can significantly increase throughput for specifications where the original invocation order is suboptimal. On the other hand,* LAZ-NAIVE *avoids the relatively small overhead of the exploration phase in cases where the specification happens to use the most effective order.*

RQ3: Analysis overhead. LAZ calls the static analyzer before every model invocation (until the first redundant one per test); this, of course, incurs runtime overhead. Tab. 1 shows aggregated analysis statistics for each benchmark (i.e., model). The results are averaged over five testing campaigns (with different random seeds) and all specifications of each benchmark. In particular, the second and third columns show the number of skipped model invocations and their percentage over the total number of invocations. Note that the specifications for LunarLander and BipedalWalker are 20-safety properties (the total number of invocations per specification is 20,000 for 1000 tests), whereas all other specifications are 2-safety. The fourth and fifth columns show the number of analysis queries and the percentage of cache hits. The last two columns show the time spent in the static analyzer (in seconds) and the analysis overhead (in percent) over the total testing time.

Overall, a small number of queries to the analyzer suffices to skip many model invocations as shown in the table (cf. columns 2 and 4). Recall from Sect. 5 that LAZ caches

Table 1: Analysis statistics per benchmark (i.e., tested model), averaged over 5 campaigns with different random seeds and all specifications.

Benchmark	Skipped Invocations	Skipped Invocations (in %)	Analysis Queries	Cache Hits (in %)	Analysis Time (in secs)	Analysis Overhead (in %)
GermanCredit	684.0	34.2	4.0	99.6	2.34	4.97
COMPAS	534.1	26.7	5.7	99.4	3.06	5.87
MNIST	936.6	46.8	44.8	95.5	20.89	33.37
HotelReview	378.8	18.9	4.0	99.6	2.49	2.87
SpeechCommand	872.8	43.6	16.0	98.4	7.83	16.17
LunarLander	3285.9	16.4	172.5	97.6	44.91	2.99
BipedalWalker	864.4	4.3	33.5	99.7	5.57	0.06

analysis queries as well as their results and can, therefore, avoid repeatedly posing the same queries to the analyzer. The cache helps save from 95.5% to 99.7% of the analyzer queries, significantly improving LAZ's performance (see column 5).

The number of analysis queries (column 4) depends on two factors, the number of model invocations for a given specification and the diversity of the queries—the more diverse the queries, the slower the cache is saturated. Queries are, for instance, more diverse for models with larger ranges of possible outputs. Consequently, the table shows that more analysis queries are needed for LunarLander and BipedalWalker, which have more model invocations than other benchmarks, as well as for MNIST and SpeechCommand, where the models return more values than other benchmarks.

A single analysis query takes less than 0.6 seconds, which is negligible compared to most model invocations. Note that even when performing a model invocation is faster than performing an analysis query, LAZ-NAIVE still outperforms NOMOS as the query results are cached and pay off in the long run. For such models however, we introduce an alternative exploration phase (LAZ-FLEX) and discuss it in RQ4.

RQ4: Hyperparameter study. In this research question, we first consider four variants of LAZ that double and halve each of the $explorePt$ and $marginPt$ parameters, namely, LAZ 5–10, 10–5, 20–10, and 10–20. Fig. 4b shows their throughput relative to the default 10–10 setting.

Overall, the throughput differences are small, being less than 2.5% for most specifications. Both increasing the exploration phase (20–10) and increasing the time margin (10–20) tend to decrease throughput. When increasing the exploration phase, more time is spent executing worse configurations. When increasing the time margin, small, but not insignificant, throughput differences may not end up being exploited simply because the number of skipped invocations (our tie breaker criterion) is slightly higher for the worse invocation orders.

When reducing the exploration phase (5–10) or the time margin (10–5), the throughput does not change significantly. A shorter exploration phase can improve throughput if the selected order is indeed faster. However, these settings may make the selection of the best invocation order more brittle. Specifically, there is a higher risk of selecting a worse configuration due to normal fluctuations in running time or the random selection of model inputs during the exploration phase.

We conclude that the default configuration strikes a good balance across the vast majority of specifications for different benchmarks. In addition, our approach appears to be robust with respect to different hyperparameter settings.

Next, we also evaluate these different settings for LAZ-FLEX, the variant of LAZ that includes an additional configuration in the exploration phase for disabling the static analysis. As shown in Fig. 4b, LAZ-FLEX performs worse than LAZ for all settings. We observed that LAZ-FLEX often chooses to disable the analysis after the exploration, which suggests that the exploration phase is too short to amortize the cost of running the analysis. In fact, for LAZ-FLEX, the variant with the longest exploration phase (20–10) achieves the highest throughput as it chooses to enable the analysis 1.5x more often than 10–10 and 3x more often than 5–10.

7 Related Work

In the last few years, it has been observed that hyperproperties are useful in specifying a wide range of ML models (e.g., [Seshia *et al.*, 2018; Sharma and Wehrheim, 2020; Christakis *et al.*, 2023; Fluri *et al.*, 2024]). NOMOS [Christakis *et al.*, 2023], in particular, provides a specification language for expressing such properties.

Beyond specifications, there is a large amount of work on testing specific hyperproperties, such as robustness [Tian *et al.*, 2018; Zhang *et al.*, 2018; Zhou and Sun, 2019; He *et al.*, 2020], fairness [Galhotra *et al.*, 2017; Udeshi *et al.*, 2018; Tramèr *et al.*, 2017], and monotonicity [Sharma and Wehrheim, 2020; Deng *et al.*, 2021; Deng *et al.*, 2022]. Recently, more general hyperproperties have been tested in diverse domains, e.g., where ground truth is expensive to obtain or simply beyond human knowledge [Christakis *et al.*, 2023; Fluri *et al.*, 2024].

Static analysis has been used to improve testing in other domains, e.g., for regular software [Christakis *et al.*, 2016; Ferles *et al.*, 2017] or smart contracts [Wüstholz and Christakis, 2020]. Inspired by these, LAZ improves the performance of NOMOS by using static analysis to skip redundant and expensive ML model invocations. In theory, LAZ could complement any hyperproperty-testing technique.

8 Conclusion

We presented LAZ, a novel approach for lazy testing of ML models that automatically identifies redundant and expensive model invocations and skips them. LAZ increases test throughput by median 33% compared to NOMOS, a state-of-the-art testing framework for ML models, and skips up to 47% model invocations with only a few analysis queries.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science) as well as the Austrian Science Fund (FWF) 10.55776/DOC1345324.

References

[Albarghouthi *et al.*, 2017] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. FairSquare: Probabilistic verification of program fairness. *PACMPL*, 1:80:1–80:30, 2017.

[Athavale *et al.*, 2024] Anagha Athavale, Ezio Bartocci, Maria Christakis, Matteo Maffei, Dejan Nickovic, and Georg Weissenbacher. Verifying global two-safety properties in neural networks with confidence. In *CAV*, volume 14682 of *LNCS*, pages 329–351. Springer, 2024.

[Bastani *et al.*, 2019] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. Probabilistic verification of fairness properties via concentration. *PACMPL*, 3:118:1–118:27, 2019.

[Brockman *et al.*, 2016] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

[Carlini and Wagner, 2017] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *S&P*, pages 39–57. IEEE Computer Society, 2017.

[Chen *et al.*, 1998] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST–CS98–01, HKUST, 1998.

[Christakis *et al.*, 2016] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *ICSE*, pages 144–155. ACM, 2016.

[Christakis *et al.*, 2023] Maria Christakis, Hasan Ferit Eniser, Jörg Hoffmann, Adish Singla, and Valentin Wüstholz. Specifying and testing k-safety properties for machine-learning models. In *IJCAI*, pages 4748–4757. ijcai.org, 2023.

[Clarkson and Schneider, 2008] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *CSF*, pages 51–65. IEEE Computer Society, 2008.

[Cousot and Cousot, 1976] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *ISOP*, pages 106–130. Dunod, 1976.

[Cousot and Cousot, 1977] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[Deng *et al.*, 2021] Yao Deng, Guannan Lou, James Xi Zheng, Tianyi Zhang, Miryung Kim, Huai Liu, Chen Wang, and Tsong Yueh Chen. BMT: Behavior driven development-based metamorphic testing for autonomous driving models. In *MET@ICSE*, pages 32–36. IEEE Computer Society, 2021.

[Deng *et al.*, 2022] Yao Deng, Xi Zheng, Tianyi Zhang, Huai Liu, Guannan Lou, Miryung Kim, and Tsong Yueh Chen. A declarative metamorphic testing framework for autonomous driving. *TSE*, pages 1–20, 2022.

[Ferles *et al.*, 2017] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. Failure-directed program trimming. In *ESEC/FSE*, pages 174–185. ACM, 2017.

[Fluri *et al.*, 2024] Lukas Fluri, Daniel Paleka, and Florian Tramèr. Evaluating superhuman models with consistency checks. In *SaTML*, pages 194–232. IEEE Computer Society, 2024.

[Galhotra *et al.*, 2017] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: Testing software for discrimination. In *ESEC/FSE*, pages 498–510. ACM, 2017.

[Gehr *et al.*, 2018] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: Safety and robustness certification of neural networks with abstract interpretation. In *S&P*, pages 3–18. IEEE Computer Society, 2018.

[Goodfellow *et al.*, 2015] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[He *et al.*, 2020] Pinjia He, Clara Meister, and Zhendong Su. Structure-invariant testing for machine translation. In *ICSE*, pages 961–973. ACM, 2020.

[Hofmann, 1994] Hans Hofmann. The German credit dataset, 1994. https://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data).

[Huang *et al.*, 2017] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *CAV*, volume 10426 of *LNCS*, pages 3–29. Springer, 2017.

[Journault *et al.*, 2019] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *VSTTE*, volume 12031 of *LNCS*, pages 1–18. Springer, 2019.

[Larson *et al.*, 2016] Jeff Larson, Surya Mattu, Lauren Kirchner, and Julia Angwin. How we analyzed the COMPAS recidivism algorithm, 2016. https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm.

[LeCun *et al.*, 1999] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits, 1999. http://yann.lecun.com/exdb/mnist.

[Liu, 2017] Jiashen Liu. 515K hotel reviews data in Europe, 2017. https://www.kaggle.com/datasets/jiashenliu/515k-hotel-reviews-data-in-europe.

[Madry *et al.*, 2018] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian

Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*. OpenReview.net, 2018.

[Segura *et al.*, 2016] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. A survey on metamorphic testing. *TSE*, 42:805–824, 2016.

[Seshia *et al.*, 2018] Sanjit A. Seshia, Ankush Desai, Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte, and Xiangyu Yue. Formal specification for deep neural networks. In *ATVA*, volume 11138 of *LNCS*, pages 20–34. Springer, 2018.

[Sharma and Wehrheim, 2020] Arnab Sharma and Heike Wehrheim. Higher income, larger loan? Monotonicity testing of machine learning models. In *ISSTA*, pages 200–210. ACM, 2020.

[Singh *et al.*, 2019] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. An abstract domain for certifying neural networks. *PACMPL*, 3:41:1–41:30, 2019.

[Tian *et al.*, 2018] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *ICSE*, pages 303–314. ACM, 2018.

[Tramèr *et al.*, 2017] Florian Tramèr, Vaggelis Atlidakis, Roxana Geambasu, Daniel J. Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. FairTest: Discovering unwarranted associations in data-driven applications. In *EuroS&P*, pages 401–416. IEEE Computer Society, 2017.

[Tran-Thanh *et al.*, 2010] Long Tran-Thanh, Archie C. Chapman, Enrique Munoz de Cote, Alex Rogers, and Nicholas R. Jennings. Epsilon-first policies for budget-limited multi-armed bandits. In *AAAI*, pages 1211–1216. AAAI, 2010.

[Udeshi *et al.*, 2018] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. Automated directed fairness testing. In *ASE*, pages 98–108. ACM, 2018.

[Urban *et al.*, 2020] Caterina Urban, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Perfectly parallel fairness certification of neural networks. *PACMPL*, 4:185:1–185:30, 2020.

[Warden, 2018] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *CoRR*, abs/1804.03209, 2018.

[Wüstholz and Christakis, 2020] Valentin Wüstholz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *ICSE*, pages 789–800. ACM, 2020.

[Zhang *et al.*, 2018] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *ASE*, pages 132–142. ACM, 2018.

[Zhou and Sun, 2019] Zhi Quan Zhou and Liqun Sun. Metamorphic testing of driverless cars. *CACM*, 62:61–67, 2019.