HyDiff: Hybrid Differential Software Analysis

Yannic Noller
yannic.noller@acm.org
Humboldt-Universität zu Berlin
Germany

Corina S. Păsăreanu
corina.pasareanu@west.cmu.edu
Carnegie Mellon University Silicon Valley, NASA Ames Research Center
USA

Marcel Böhme
marcel.boehme@acm.org
Monash University
Australia

Youcheng Sun
youcheng.sun@qub.ac.uk
Queen's University Belfast
United Kingdom

Hoang Lam Nguyen
nguyenhx@informatik.hu-berlin.de
Humboldt-Universität zu Berlin
Germany

Lars Grunske
grunske@informatik.hu-berlin.de
Humboldt-Universität zu Berlin
Germany

ABSTRACT

Detecting regression bugs in software evolution, analyzing side-channels in programs and evaluating robustness in deep neural networks (DNNs) can all be seen as instances of differential software analysis, where the goal is to generate diverging executions of program paths. Two executions are said to be diverging if the observable program behavior differs, e.g., in terms of program output, execution time, or (DNN) classification. The key challenge of differential software analysis is to simultaneously reason about multiple program paths, often across program variants.

This paper presents HyDiff, the first hybrid approach for differential software analysis. HyDiff integrates and extends two very successful testing techniques: Feedback-directed greybox fuzzing for efficient program testing and shadow symbolic execution for systematic program exploration. HyDiff extends greybox fuzzing with divergence-driven feedback based on novel cost metrics that also take into account the control flow graph of the program. Furthermore HyDiff extends shadow symbolic execution by applying four-way forking in a systematic exploration and still having the ability to incorporate concrete inputs in the analysis. HyDiff applies divergence revealing heuristics based on resource consumption and control-flow information to efficiently guide the symbolic exploration, which allows its efficient usage beyond regression testing applications. We introduce differential metrics such as output, decision and cost difference, as well as patch distance, to assist the fuzzing and symbolic execution components in maximizing the execution divergence.

We implemented our approach on top of the fuzzer AFL and the symbolic execution framework Symbolic PathFinder. We illustrate HyDiff on regression and side-channel analysis for Java bytecode programs, and further show how to use HyDiff for robustness analysis of neural networks.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380363

KEYWORDS

Differential Program Analysis, Symbolic Execution, Fuzzing

ACM Reference Format:
Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: Hybrid Differential Software Analysis. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380363

1 INTRODUCTION

The challenge of *differential software analysis* is to reason about multiple program executions simultaneously. This may include executions of different inputs on the same program or executions of the same input across multiple programs or variants. In this paper, we focus on the problem of generating inputs that trigger maximal behavioral difference across program executions. We say that the considered executions *diverge* if they lead to different observable behavior. We consider various forms of divergence: in terms of control-flow paths (decision difference), observed outputs (output difference) and resource consumption (cost difference), which are required for different target applications.

For instance, automated *regression test generation* aims to generate an input such that its execution on two successive versions diverges [32, 33, 36, 37, 50] in terms of control-flow paths or observed outputs. An input that witnesses an output difference may expose a regression bug, i.e., an error that was introduced by the modifications from one version to the next. Many existing regression test generation techniques focus only on the affected paths in the changed version [6, 37, 46, 48, 50]. However, regression testing inherently requires to reason about both program versions simultaneously to mitigate the problem of false negatives.

Automated *side-channel vulnerability detection* aims to generate two secret inputs such that (given the same public inputs) their execution yields different resource consumption [3, 29]. A difference in observable behavior, e.g., memory consumption or execution time, indicates possible information leakage about the secret inputs. For instance, if the time it takes to check a password of length n turns out to be proportional to n, an attacker can derive the password length by observing the execution time. Such information leaks represent serious vulnerabilities as demonstrated by the recent *Meltdown* [25] and *Spectre* [21] vulnerabilities. Their detection requires reasoning over multiple program executions.

Automated *robustness analysis of deep neural networks* (DNNs) aims to generate two inputs that are only marginally different (to an imperceptible degree), yet the DNN classifies them differently [13, 43]. Such adversarial perturbations are considered a major safety and security concern. For instance, when a DNN classifier is used in an autonomous car to identify street signs, a misclassification undermines the safety of the passengers. Detecting such adversarial behaviors requires reasoning over more than one execution at once.

In this work we present HyDiff, a *hybrid* technique that integrates fuzzing, a fast but shallow testing technique, and symbolic execution, a deep but slow analysis technique, for a differential software analysis that can handle all of the above application scenarios. For fuzzing, we build on the popular greybox fuzzer *American Fuzzy Lop* (AFL) [52] which we extend with a divergence-based feedback channel and new search heuristics that aim to maximize the divergence across multiple program executions. The role of the fuzzer is to generate quickly many inputs, by mutating existing seed inputs, and favoring the ones that are shown to increase execution divergence, according to different divergence metrics. This enables HyDiff to swiftly discover many divergences, particularly in the beginning of the campaign. However, fuzzing alone cannot reach very deep into the code, grappling with complex branch conditions and paths in low-probability domains. We therefore propose to enhance the fuzzing with a symbolic execution component that can tackle these issues by generating inputs via constraint solving based on conditions collected from the code.

To this end we built HyDiff's differential symbolic execution (DSE) component by extending *shadow* symbolic execution (SSE) [33], a differential analysis technique which represents two program versions in one *annotated* program and uses *four-way forking* to explore all four decisions resulting from the combined branching behavior of both versions. We extend SSE from [33] with an *incremental* approach to allow it to periodically check and incorporate the newly available concrete inputs from the fuzzer. These new concrete inputs are executed concolically (to collect symbolic constraints along concrete paths) and they help to drive the symbolic analysis in directions of particular interest, as directed by our heuristics. In addition, HyDiff's DSE component avoids the exploration of uninteresting code areas (e.g., in case of regression testing any unchanged program blocks), by pruning the search space, based on an inter-procedural control flow graph (ICFG) analysis. As a result, HyDiff's DSE implements an efficient four-way forking exploration strategy driven by concrete inputs, which allows it to cut deeper into the program to reveal divergences.

The integration of these two differential analysis approaches allows us to leverage their strengths and overcome their weaknesses. Both components are executed at the same time while they exchange interesting inputs, enhancing each other. We note that hybrid approaches that combine fuzzing and symbolic execution have been proposed in the past [8, 30, 41]. However HyDiff is the first approach that explores the interplay between the two techniques for a *differential* analysis. The problem is significantly harder than a classical program analysis and hence existing hybrid solutions are not applicable. Furthermore, while the majority of previous differential symbolic approaches [33, 37, 50] focus on regression testing, HyDiff is a general approach that is more widely applicable; for instance it can compute divergences that are related not only to different execution paths but also to different *costs* associated with the paths and that can be used in a side-channel analysis.

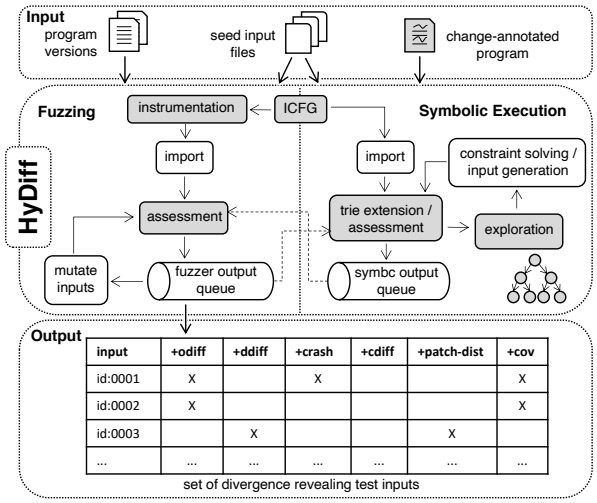

Figure 1: HyDiff overview.

We implemented HyDiff for the analysis of Java bytecode on top of the fuzzer AFL [52] and the symbolic execution framework Symbolic PathFinder (SPF) [34]. We evaluate HyDiff for three applications: (i) regression test generation, (ii) side-channel vulnerability detection, and (iii) robustness analysis of DNNs. For (i) regression test generation, we evaluate HyDiff on the Traffic Anti-Collision Avoidance System (TCAS) [51], several subjects from the Defects4J benchmark [18], and regression errors in Apache CLI [5]. For (ii) side-channel vulnerability detection, we evaluate HyDiff on subjects taken from previous studies on side-channel vulnerabilities [9, 29] as well as an implementation of modular exponentiation, an operation involving non-linear constraints and typically used in cryptography [39]. For (iii) robustness analysis of DNNs, we evaluate HyDiff on neural networks taken from the MNIST dataset [24].

Our results show that HyDiff can identify divergences for all examined applications. Furthermore, it identifies a divergence faster and can reveal more differences than its individual components. In summary, this work makes the following contributions:

- We present HyDiff, the first hybrid differential analysis approach that integrates greybox fuzzing and symbolic execution.
- We extend greybox fuzzing with a divergence-based feedback channel and novel search heuristics.
- We extend shadow symbolic execution with incremental and pruning techniques as well as novel heuristics that drive the search towards finding structural- and cost-related divergences.
- We demonstrate HyDiff in multiple application scenarios, incl. regression test generation, side-channel analysis, and the robustness analysis of neural networks in adversarial settings.

2 OVERVIEW

2.1 HyDiff's Workflow

Figure 1 shows an overview of our approach. HyDiff takes several programs and a set of seed files as input (top) and produces

test inputs classified by the observed divergence (bottom). More specifically, the *inputs* for HyDiff are as follows: (1) For differential fuzzing, HyDiff takes the considered program (versions), a test driver and a target specification. (2) For differential symbolic execution, it takes the change-annotated program, a test driver, and a configuration. (3) For both, it takes seed input files as initial seed corpus. The seed input files are used to perform an initial exploration of the program(s). The test drivers parse the generated inputs and pass them as parameters to the program's entry point(s). The target specification includes a list of changed program locations and is used to guide the fuzzer to these locations. The change-annotated program includes annotations for the differential symbolic execution. These annotations are adapted from [33] and handle changed expressions, added/removed code lines, modified functions etc. The configuration contains technical parameters such as the used constraint solver.

The *output* of HyDiff is a set of generated inputs classified by the observed divergence (see bottom of Figure 1). We distinguish between differences in: output (`+odiff`), control-flow (`+ddiff`), crashing behavior (`+crash`), and execution cost (`+cdiff`). Together, these represent the differential metrics used to guide the search for inputs in our analysis. Furthermore we aim to compute inputs that decrease the *distance* to a divergence-inducing target (e.g., a patch; `+patch-dist`) or increase branch coverage (`+cov`).

The middle layer in Figure 1 presents HyDiff's high-level workflow. The left side shows HyDiff's differential fuzzing component, which makes use of the *inter-procedural control-flow graph* (ICFG) constructed for the program. Marking divergence-inducing program locations (e.g., changes in case of regression testing) in the ICFG allows to compute distance metrics for guiding the exploration in the fuzzer. The *fuzzer output queue* is initialized with the seed input files. Our differential fuzzing uses a lightweight instrumentation-guided genetic algorithm, similar to AFL [52]. The *instrumentation* allows us to compute the patch distance and other differential metrics for an input dynamically, during execution. The fuzzer stores information about the observed inputs such as minimum patch distance or output differences. The inputs that are generated by the fuzzer are *assessed* by checking whether any differential metric gets improved by them. The goal is to keep only these *interesting* inputs in the fuzzer queue, which improve at least one of the metrics. These inputs are further modified using byte-level mutations to generate new inputs that are executed and assessed again by the fuzzer.

On the middle-right of Figure 1, we illustrate HyDiff's differential symbolic execution (DSE) component. The central data structure is a reduced symbolic execution tree, called *trie*, which encodes succinctly the results of the differential symbolic execution. This trie is updated incrementally as the analysis progresses. The nodes in the trie represent the so far covered decision points together with the observed concrete choices for these decisions. This symbolic execution *trie* is initialized with the decision points and choices observed along the concrete executions paths of the provided seed inputs. The trie gets extended when new generated inputs or inputs imported from the fuzzer are executed. This step is called *trie extension* and also includes an *assessment*, namely the gathering of multiple differential metrics about each execution, e.g., patch distance and cost difference, but also information about the branch

```
0  int calculate(int x, int y) {
1    int div;
2    switch (x) {
3    case 0:
4      div = y + 1;
5      break;
6    case 1:
7      div = y + 2;
8      break;
9    ...
10   case 250:
11     div = y + 251;
12     break;
13   default:
14     if (x == 123456) {
15       // CHANGE: expression y + 123455 to y + 123456
16       div = change(y + 123455 , y + 123456);
17     } else
18       div = x + 31;
19   }
20   int result = x / div;
21
22   // CHANGE: added conditional statement
23   if (change(false, result > 0))
24     result = result + 1;
25   return result;
26 }
```

Listing 1: Example program with annotated changes.

coverage. Each node in the trie gets ranked for its ability to show new interesting behavior in terms of the differential analysis. An *exploration* step then picks the most promising node and performs a deeper exploration on alternative paths, starting with that node. This step is performed with a purely (bounded) symbolic execution, as opposed to a concolic execution. We thus use the trie to *guide* the symbolic exploration towards interesting paths that have not been explored before. The resulting, satisfiable path conditions are solved and new inputs are generated. Each new input is *assessed* again for its actual ability to reveal a divergence, since the nodes get picked based on heuristics.

2.2 Illustrating Example

In order to demonstrate the challenges of differential analysis and to illustrate the pertinent concepts of our approach, we introduce a simple example for regression testing. Listing 1 shows a change-annotated program, which represents two successive versions of the `calculate`-program. The changes fix one error, but introduce another—a typical regression bug. Specifically, in line 16 the developer changed the right-hand side expression from $y + 123455$ to $y + 123456$, which fixed a division-by-zero error for $y = -123455$, but introduced another crash for $y = -123456$. In line 23, the developer added a conditional statement $result = result + 1$ if $result > 0$. This changes the output for all positive results. However, it does not directly fix or introduce any crashes.

HyDiff's differential fuzzer component takes the patch as target specification and computes distance values within the ICFG. During fuzzing, these distance values will be used (as patch distance) to guide the differential fuzzer towards the modifications. The fuzzer

```
0  int main(String[] args) {
1    /* Read input. */
2    int x = readInput(args);
3    int y = readInput(args);
4    /* Execute old version. */
5    Measurement.reset();
6    int res1 = calculate_old(x, y);
7    boolean[] dec1 = Measurement.getDecisions();
8    long cost1 = Measurement.getCost();
9    /* Execute new version. */
10   Measurement.reset();
11   int res2 = calculate_new(x, y);
12   boolean[] dec2 = Measurement.getDecisions();
13   long cost2 = Measurement.getCost();
14   /* Report differences. */
15   Result.setDecisionDiff(dec1, dec2);
16   Result.setOutputDiff(res1, res2);
17   Result.setCostDiff(cost1, cost2);
18 }
```

Listing 2: Simplified fuzzing driver for the example

```
0  int main(String[] args) {
1    int x, y;
2    /* Concolic or Symbolic Execution Mode */
3    if (args.length == 1) {
4      /* Read input. */
5      int valueX = readInput(args);
6      int valueY = readInput(args);
7      /* Add symbolic values. */
8      x = addSymbolicValue(valueX, "sym_x");
9      y = addSymbolicValue(valueY, "sym_y");
10   } else {
11     /* Introduce symbolic values. */
12     x = makeSymbolicValue("sym_x");
13     y = makeSymbolicValue("sym_y");
14   }
15   /* Execute change–annotated version. */
16   calculate(x, y);
17 }
```

Listing 3: Simplified symbolic execution driver for the example

component executes each generated input on both successive versions of the `calculate`-program compiled separately. HyDiff's differential symbolic execution (DSE) component takes both versions as change-annotated (integrated) program as shown in Listing 1. The DSE component uses these annotations to infer expressions that indicate a difference between the old and the new version (cf. the expression for `div` in line 16 and the boolean expression in line 23 in Listing 1). Like the fuzzing component, the DSE component requires a small test driver to interface the change-annotated program. The symbolic test driver reads the input, marks symbolic values, and calls the change-annotated program. A simple symbolic test driver for the `calculate`-program is shown in Listing 3. We can see that the symbolic test driver facilitates both a concolic and a symbolic mode. This is required to enable the exploration and assessment phase (Figure 1.middle-right). During the *exploration* phase, the inputs are marked as symbolic (lines 12-13) and the change-annotated program is executed symbolically. During the *trie extension* phase, the given concrete input is marked symbolic (lines 5–9) and the change-annotated program is executed concollicaly, i.e., follows the concrete input. In this example, HyDiff's hybrid approach detects the regression bug more than *nine times faster* than HyDiff's DSE component alone. The differential fuzzing component alone times out after ten minutes without detecting the regression. HyDiff uses the strengths of both techniques, so that it can get into more paths by leveraging symbolic execution and is very fast in finding its first output difference by leveraging fuzzing.

To illustrate the challenges of each individual approach, we present some results first for running both HyDiff components independently and then together on this example. The differential fuzzing component finds its first output difference after 5.07 ($\pm$0.99) sec (where the $\pm$ value denotes the 95% confidence interval). In total it finds 1.37 ($\pm$0.17) output differences and 1.00 ($\pm$0.00) decision differences. The *new* crash is not found within the time bound of 10 min. Therefore, fuzzing is very fast in finding an output difference (less than 6 seconds), but the narrow constraint at the end is difficult to reach for fuzzing: $x = 123456 \wedge y = -123456$. Due to fuzzing's random mutations, it is simply unlikely to hit this exact condition.

In contrast, the differential symbolic execution component finds its first output difference after 135.27 ($\pm$0.66) sec. In total, it finds 35.17 ($\pm$1.10) output differences and 2.00 ($\pm$0.00) decision differences. So it reveals much more output differences than fuzzing within the given time bound. In fact, the DSE component can traverse all paths in 5 minutes. In contrast to fuzzing it also finds the new crash, after 135.80 ($\pm$0.64) sec. Nonetheless, symbolic execution needs relatively long to find its first output difference. The `switch` statement with its large amount of branches is difficult for symbolic execution, simply because it takes its time to explore all of them, especially when the interesting parts are at the end and symbolic execution traverses it in a deterministic order. Note that the branches in the `switch` cannot be pruned because there is a change after the switch, which makes every path via the switch branches a potential interesting path for exploration. In order to find an output difference earlier, symbolic execution would need a hint to direct the exploration.

For the hybrid analysis with HyDiff, the differential fuzzing and symbolic execution components are started with the same seed input. Both run their analysis and exchange inputs that are deemed interesting according to the divergence metrics after a prespecified time bound. The experimental results are as follows: first output difference after 4.73 ($\pm$0.78) sec, in total 35.13 ($\pm$1.04) output differences and 2.00 ($\pm$0.00) decision differences. HyDiff finds the new crash already after 14.43 ($\pm$0.30) sec. The following sections will explain each part and also the hybrid approach in more detail.

3 DIFFERENTIAL ANALYSIS

3.1 Differential Fuzzing

HyDiff's differential fuzzing (DF) component is a heuristic-driven greybox fuzzer with a divergence-based feedback channel. By slightly modifiying existing inputs in the seed corpus, new test inputs are generated. Generated test inputs that increase coverage or divergence (as measured by our differential metrics) are added to the seed

corpus for further fuzzing. More specifically, the fuzzing process works as follows: we use the provided program, the target specification (for regression testing we define the locations of the program changes), initial seed inputs, and the fuzzing driver as input. By using some instrumentation we can measure various metrics during the program execution, to drive the differential analysis. As the first step, the fuzzer imports the initial seed files and starts with mutating these inputs. By executing resulting mutated inputs with the instrumented program, the fuzzer can assess the inputs based on the collected metrics, and hence decide whether to keep any of them and use them in further mutation steps. These interesting inputs will be stored in the fuzzing queue. This process continues in a loop until a defined time bound is reached or the user aborts the execution.

Our core contribution in differential fuzzing is the input assessment, i.e., the selection of the mutated inputs that will be used for further mutations according to the divergence heuristics that we introduce in this work (cf. the gray areas in the upper part of Figure 1). Specifically we use the following differential metrics: the *output difference*, the *decision history difference*, the *cost difference*, and the *patch distance*.

The *output difference* has a binary value and is determined by comparing the results (depending on their type) in the fuzzing driver. Output differences also cover observed crashes, as long as the two program versions behave differently. Observing an output difference is a clear sign for a divergence revealing input. We encode the output difference and store it to remember it in further comparisons to avoid duplicates.

The *decision history* encodes a sequence of boolean values to represent each branching decision. The difference between two decision histories is used as binary value, determined by comparing each boolean value pair. If a different value pair is found, then it means that different decisions were made during program execution. Such a difference does not necessarily mean a semantic divergence, e.g., it could also just represent some refactoring, but it is an indication. Similar to the output difference, we encode and store this difference. We call this metric also just *decision difference*.

The *execution cost* represents the resource consumption during program execution: this can be time, which we approximate by counting the executed instructions or some other user-specified cost. Observing a *cost difference* is an indicator for a semantic divergence.

The *shortest distance* to the predefined targets is calculated by leveraging the information from an ICFG of the analyzed program, which is generated in advance and stored within the instrumentation. Such a *target distance* (in regression testing also called *patch distance* [28]) is calculated for the new version only because we are interested in reaching the changes in the new program version. Hitting the changed areas is necessary to find a difference in behavior, but it does not provide any guarantee for a semantic difference. The idea is to keep inputs during the mutation process, which may not improve any other differential metric, but come *closer* to the patched code regions. In case of multiple targets, we calculate distance values for all of them, and keep an input as soon as its distance value for one of the targets improves.

In summary, our input assessment keeps an input as soon as one of these metrics reveals a new behavior. In order to do so, the fuzzer stores which differences (with which values) were already observed. Additionally, the fuzzer also keeps inputs that increase the *branch coverage* in general, to be able to make progress even if no difference is currently detected. All metric values, besides the output differences, are measured via the program instrumentation.

3.2 Differential Symbolic Execution

HyDiff's differential symbolic execution (DSE) component runs in two modes: *concolically*, which can incorporate inputs from the fuzzer, but also purely *symbolically*, being able to make progress on its own, even if no inputs from the fuzzer are present. This has the benefit that DSE can operate when for example fuzzing cannot generate interesting inputs because it is stuck at some point, but can also benefit from concrete inputs to guide its own execution. The workflow of the DSE component is similar to the one in Badger [30], which however is not differential. It consists of three main phases: (i) trie extension, (ii) exploration, and (iii) input generation (cf. the middle-right of Figure 1). These steps are performed in a loop until the search space was explored exhaustively or the user aborts the analysis.

In the beginning the trie gets extended / initialized with the decision points and concrete choices, which occur along the paths of the seed inputs. Therefore, the inputs are being executed with dynamic symbolic execution, which uses the notion of the four-way forking strategy to enable the focused search for divergences, similar to [33]. As illustrated in Listing 1, HyDiff's DSE expects *change-annotations* in the program or in the driver. These change-annotations specify divergence-inducing statements in the program (e.g., modifications for regression testing). As a novel application, we also use them to infer changes in the program input (e.g., for the side-channel analysis see Section 4.3). Every executed annotation introduces a so-called *differential expression*, which consists of four parts: the old symbolic value, the old concrete value, the new symbolic value, and the new concrete value. For example in Line 16 in Listing 1, assume the concrete value for `y` is -123456 while the symbolic value of `y` is denoted by β. The statement `div=change(y+123455,y+123456)` introduces the following differential expression $\{\text{old}_{\text{sym}} = \beta + 123455,\ \text{old}_{\text{con}} = -1,\ \text{new}_{\text{sym}} = \beta + 123456,\ \text{new}_{\text{con}} = 0\}$.

The *differential expressions* are the key to handling two program executions at once; furthermore, our *exploration strategy* is driven by these differential expressions to find paths where the control-flow diverges across executions. Such paths are called: *diff paths*. As soon as the DSE hits such a *diff path* (i.e. the execution exercises a control-flow divergence), it switches to the execution of the *new* version only. Therefore, the subsequent path explorations will only use the second parameter of the change-annotations. This also means that subsequent control-flow divergences will not be detected because they are covered by the already identified divergence. Nevertheless, deeper control-flow divergences might be reachable by paths, which do not trigger the prior control-flow divergence.

During the *extension* of the trie, its nodes get assessed and ranked according to their ability to reveal divergences. The exploration step picks the most promising trie node for further exploration. Therefore, we calculate for each node the following differential metrics: the *cost difference*, and the *patch distance*. Additionally, we determine whether the node is on a *diff path*, and whether the path

condition at this node contains a *diff expression*. The information about the *diff expression* in a path condition is a good indication for a potential future divergence because divergences will be only possible if there is a *diff expression* present. Therefore, we also use this information to rank the nodes.

Mirroring the differential fuzzing, we also compute the *execution cost* which is calculated by counting the number of executed bytecode statements or it is user provided. However, in DSE we use the symbolic execution framework, which interprets every bytecode instruction, to count every statement when visited, instead of instrumenting the bytecode. In contrast to differential fuzzing, the DSE component cannot use the *output difference* as a search metric, since the execution of *diff paths* is limited to the new version, and hence, the full information about the output is not always available. However, the intrinsic goal of DSE is to push the exploration to diff paths, i.e., to identify *decision differences*. For DSE the *patch distance* is computed as the distance to the change() statements in the program. Based on the ICFG these distances and also the reachability information are pre-calculated and stored in memory. Additionally, these information are also used in DSE to prune all paths that cannot reach any change() statement.

After selecting the currently most promising node, DSE will perform a *trie-guided* symbolic execution from the beginning of the program to the selected trie node. *Trie-guided* means that it uses the choices stored in the trie to select the branch at a conditional statement, which makes it very efficient because no constraint solving is invoked. After reaching the selected node, DSE starts a bounded symbolic execution to gather new path conditions and generate new inputs (cf. step (iii) input generation in Figure 1). All the inputs generated by HyDiff's DSE component are concretely executed again. The reason is that the models used by symbolic execution as well as the change annotations may not be precise enough to ensure that every diff path discovered during symbolic execution also triggers a diff path in the real program execution. Therefore, we replay the inputs, which were generated with symbolic execution, with our fuzzing driver and reassess them for the induced difference, in order to report the correct results.

In our experiments we use the following *heuristics* to rank the nodes for exploration: (1) Prioritize nodes that contain a differential expression, but are not yet on a diff path. (2) Prioritize a node without differential expression before a node which is already on a diff path. (Note: here we only have nodes that can reach the changes). (3) Prioritize new branch coverage. (4) If two nodes have not yet touched any change, then prioritize the node with smaller distance. (5) Prioritize nodes that already have higher cost differences between the two versions. (6) Prioritize higher trie nodes.

The highest priority is to find *decision differences*, i.e., divergence of control-flow. Therefore, HyDiff's DSE component favors such potential nodes (points 1 and 2). It is the most valuable divergence metric, also because output difference cannot be encoded. It can be simply detected by checking whether we are currently in a *diff path*. The next priority is to support the fuzzer during exploration, for which it is necessary to solve constraints corresponding to conditions that are difficult for the fuzzer (point 3). As further indications for a difference we use the information about the *patch distance* and the *cost difference* (point 4 and 5). As last search parameter we favor higher nodes in the trie, which leads to a broader exploration of the search space, which also supports fuzzing. These heuristics represent the default configuration setup for our differential analysis and that they can be easily modified. Why do we not just pick nodes on a diff path with highest priority? When a node is on a diff path, then the analysis has already identified an input to trigger the divergences that causes the diff path. All unexplored branches at this node will be on a diff path as well, all of them caused by the same divergence. Therefore, our strategy focuses first on other nodes, which are not yet on a diff path, but show the potential to reach one and potentially trigger a different divergence.

3.3 Hybrid Analysis

HyDiff implements a hybrid approach, which combines the above described differential fuzzing and the differential symbolic execution components. The components communicate with each other to exchange interesting inputs discovered with either technique. Both components get started with the same seed input(s). After a pre-defined time property, each part checks the other output queue for interesting inputs. Therefore, differential fuzzing has to execute the inputs from differential symbolic execution to see whether they improve any of the differential metrics. Differential symbolic execution has to replay the inputs from differential fuzzing to extend the trie, check whether the path was already explored, and whether a new code area was hit, which could introduce differential expressions and update the ranking of the trie nodes. The default setup for HyDiff is to start both components at the same time, however, this can be changed via its configuration. For example one component can be started later than the other and use the already generated inputs by the other component as additional seed inputs.

3.4 Implementation Details

We implemented our approach for the differential analysis of Java bytecode. The fuzzing part is built on top of AFL, similar to Kelinci [19], where AFL is used as the underlying fuzzing engine. A Java wrapper is used to relay the program execution triggered by AFL to the actual Java program. We instrument the Java bytecode with the ASM bytecode manipulation framework [16] to measure the differential metrics. The ICFG is constructed by using Apache Commons BCEL [11]. The symbolic execution part is build on top of Symbolic PathFinder (SPF), a symbolic execution tool for Java bytecode [34], and its extension for shadow symbolic execution [31]. For the four-way forking we had to modify each bytecode instruction interpretation to be able to handle differential expressions and fork the execution accordingly.

4 APPLICATIONS AND EVALUATION

In this section we describe three applications of our HyDiff approach: regression testing, side-channel analysis and differential analysis of neural networks. The broad scope of these case studies illustrates the generality of our hybrid differential analysis. HyDiff is not limited to one type of behavioral difference, such as output differences for regression testing. It also allows to detect other types of differences, such as in execution cost differences for side-channel vulnerability discovery. The elegant way of representing changes not only in the programs (as in regression testing) but also in the input facilitates the differential analysis even of

Table 1: Results for Regression Testing (t=600sec=10min, average over 30 runs). The bold values represent significant differences to the closest other subject verified with the Wilcoxon ranked sum test ($\alpha = 0.05$).

Subject	Differential Fuzzing (DF)				Differential Symbolic Execution (DSE)				HyDiff			
(# changes)	$\overline{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\overline{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\overline{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$
TCAS-1 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	**22.47 (±0.39)**	21	1.00 (±0.00)	3.00 (±0.00)	49.87 (±5.48)	29	1.00 (±0.00)	**4.67 (±0.40)**
TCAS-2 (1)	441.83 (±57.70)	120	0.70 (±0.23)	2.13 (±0.73)	182.37 (±1.96)	177	1.00 (±0.00)	9.00 (±0.00)	186.87 (±12.30)	92	1.23 (±0.18)	**13.83 (±0.37)**
TCAS-3 (1)	588.43 (±15.18)	392	0.10 (±0.11)	38.63 (±1.96)	**239.07 (±2.57)**	232	2.00 (±0.00)	19.00 (±0.00)	263.20 (±3.61)	236	2.00 (±0.00)	**57.43 (±1.54)**
TCAS-4 (1)	28.47 (±10.42)	2	1.00 (±0.00)	18.27 (±1.06)	-	-	0.00 (±0.00)	3.00 (±0.00)	43.70 (±14.01)	3	1.00 (±0.00)	**22.53 (±1.01)**
TCAS-5 (1)	184.93 (±46.66)	24	2.00 (±0.00)	31.97 (±1.06)	185.40 (±1.95)	180	2.00 (±0.00)	24.00 (±0.00)	94.60 (±30.72)	1	2.00 (±0.00)	**49.83 (±1.27)**
TCAS-6 (1)	233.63 (±54.48)	4	0.97 (±0.06)	4.13 (±0.83)	**5.30 (±0.23)**	4	1.00 (±0.00)	6.00 (±0.00)	7.57 (±0.26)	6	1.00 (±0.00)	**10.37 (±0.70)**
TCAS-7 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	**56.97 (±0.76)**	54	2.00 (±0.00)	6.00 (±0.00)	71.70 (±1.71)	62	2.00 (±0.00)	**8.93 (±0.39)**
TCAS-8 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	**51.70 (±0.16)**	51	2.00 (±0.00)	6.00 (±0.00)	65.33 (±0.75)	61	2.00 (±0.00)	**8.77 (±0.49)**
TCAS-9 (1)	221.73 (±48.83)	10	1.00 (±0.00)	6.13 (±0.85)	184.20 (±0.57)	181	1.00 (±0.00)	15.00 (±0.00)	185.53 (±18.42)	39	1.00 (±0.00)	**22.37 (±0.89)**
TCAS-10 (2)	173.47 (±46.27)	1	1.93 (±0.09)	12.27 (±1.69)	**5.23 (±0.15)**	5	2.00 (±0.00)	12.00 (±0.00)	7.63 (±0.22)	7	2.00 (±0.00)	**21.30 (±0.82)**
Math-10 (1)	221.13 (±56.26)	10	**64.50 (±15.98)**	15.50 (±2.35)	**2.87 (±0.15)**	2	7.00 (±0.00)	10.00 (±0.00)	3.87 (±0.20)	3	44.33 (±5.47)	**32.00 (±1.39)**
Math-46 (1)	377.87 (±63.43)	77	0.80 (±0.14)	36.33 (±1.07)	**117.80 (±0.55)**	113	1.00 (±0.00)	5.80 (±0.14)	122.00 (±8.34)	49	1.00 (±0.00)	38.17 (±0.82)
Math-60 (7)	6.93 (±0.63)	4	219.17 (±5.26)	92.90 (±1.64)	**3.60 (±0.18)**	3	2.00 (±0.00)	3.00 (±0.00)	4.77 (±0.15)	4	**234.23 (±5.63)**	94.20 (±2.67)
Time-1 (14)	5.17 (±1.20)	2	123.30 (±5.86)	170.63 (±3.43)	5.33 (±0.17)	5	33.00 (±0.00)	32.00 (±0.00)	3.80 (±0.69)	1	**189.73 (±11.94)**	**225.33 (±5.62)**
CLI1-2 (13)	-	-	0.00 (±0.00)	159.53 (±4.05)	-	-	0.00 (±0.00)	4.00 (±0.00)	-	-	0.00 (±0.00)	**169.40 (±4.07)**
CLI2-3 (13)	10.83 (±3.33)	2	82.30 (±3.98)	176.83 (±3.62)	-	-	0.00 (±0.00)	35.00 (±0.00)	13.27 (±3.62)	2	84.63 (±4.24)	**242.70 (±3.80)**
CLI3-4 (8)	7.43 (±1.60)	1	96.73 (±4.54)	279.13 (±4.51)	-	-	0.00 (±0.00)	7.00 (±0.00)	8.93 (±2.13)	2	**113.33 (±4.80)**	**471.50 (±8.93)**
CLI4-5 (13)	589.57 (±16.05)	358	0.07 (±0.09)	219.30 (±3.74)	-	-	0.00 (±0.00)	2.00 (±0.00)	551.97 (±45.65)	125	0.13 (±0.12)	**235.17 (±5.73)**
CLI5-6 (21)	**4.13 (±1.04)**	1	143.87 (±4.99)	182.00 (±5.54)	-	-	0.00 (±0.00)	5.00 (±0.00)	6.17 (±1.31)	2	**177.80 (±4.39)**	**214.47 (±6.38)**

non-traditional software, such as the robustness analysis of neural networks. For all three applications we conducted experiments for differential fuzzing, differential symbolic execution, and their combination. Our implementation's source code as well as all evaluation artifacts (incl. the subjects and all drivers) are published here: https://doi.org/10.5281/zenodo.3627893 [44].

4.1 Experiment Infrastructure

For our experiments we used a virtual machine with Ubuntu 18.04.1 LTS featuring 2x Intel(R) Xeon(R) CPU X5365 @ 3.00GHz with 8GB of memory, OpenJDK 1.8.0_191 and GCC 7.3.0. Due to the randomness in fuzzing, we repeated each experiment 30 times and reported the averaged results together with the 95% confidence intervals and the max/min values. Although symbolic execution is a deterministic process, we observed small variations between experiments, mostly in the time until the first observed difference. The variations can be caused by the constraint solver and other activities on the machine. Therefore, we decided to average the results for it as well over 30 repetitions. The experiments for regression testing use a timeout of 10 min (=600 sec), the experiments for side-channel analysis a timeout of 30 min (=1800 sec) to match the experiments from DiffFuzz [29], and the experiments with the DNN are executed for 1 hour (=3600 sec) because of the long running program executions. As seed input we used a randomly generated file, but we ensured that this initial input does not crash the application, which is a precondition for AFL. The highlighted values in the Tables 1, 2 and 3 represent significant differences to the closest other subject verified with the Wilcoxon Test (with 5% significance level).

4.2 Regression Testing

In regression testing we have multiple versions of the same program and search for regression bugs, i.e., changes in the program that lead to semantic failures. Our motivational example in Section 2.2 already shows how the fuzzing and symbolic execution driver would look like. The fuzzing driver executes both versions with the same input and measures the differences (i.e., decision differences, output differences, cost differences and target distances). The symbolic execution driver executes only a single program version, which contains change annotations. In the case of regression testing, our approach aims to find all divergence-revealing inputs. Whether an input represents a regression or only a progression, i.e., whether an observable change in the output is intended or a bug, is out of scope for this work.

4.2.1 Subjects. For the evaluation we searched for Java applications with multiple available versions. We started with the Traffic collision avoidance system (TCAS) originally taken from the SIR repository [40], which was used before in other regression testing work related to SPF [51]. It has 143 LOC and contains injected mutations as changes, in our case 1-2 changes per version. We used the first ten versions of TCAS for a preliminary assessment of our approach. We further analyzed real-world applications from the Defects4J benchmark [18], which contains a large set of Java bugs, but not necessarily regression bugs, and hence, requires some manual investigation. We searched for regression bugs in the projects Math (85 KLOC) and Time (28 KLOC) and identified four subjects: *Math-10*, *Math-46*, *Math-60* and *Time-1*. They contain between 1 and 14 *changes* per version, where one *change* means a difference between the program versions that can be represented by one *change-annotation*, and hence, also can span multiple lines. Additionally, we investigated five versions from Apache CLI [12] (4966 LOC), which was also used before in other regression testing work [5] and contains between 8 and 21 changes per version.

4.2.2 Results. We collected four metrics: $\overline{t}$ *+odiff* denotes the average time (sec) until the first output difference (incl. a crash in the new version) has been observed, t_{min} denotes the minimum time over all runs for an output difference, $\overline{\#odiff}$ denotes the average number of found output differences, and $\overline{\#ddiff}$ denotes the average number of found decision differences.

As shown in Table 1, HyDiff is able to classify the subjects correctly in the given timeout of 10 minutes: for all regression subjects, except for the *CLI1-2*, HyDiff identifies at least one input that triggers an output difference (+odiff). For *CLI1-2* there is no

Table 2: Results for Side-Channel Analysis (t=1800sec=30min, average over 30 runs). The bold values represent significant differences to the closest other subject verified with the Wilcoxon ranked sum test ($\alpha = 0.05$).

Benchmark		Differential Fuzzing (DF)			Differential Symbolic Execution (DSE)			HyDiff		
Subject	Version	$\overline{\delta}$	δ_{max}	$\overline{t} : \delta > 0$	$\overline{\delta}$	δ_{max}	$\overline{t} : \delta > 0$	$\overline{\delta}$	δ_{max}	$\overline{t} : \delta > 0$
Blazer_login	Safe	0.00 (±0.00)	0	-	0.00 (±0.00)	0	-	0.00 (±0.00)	0	-
Blazer_login	Unsafe	132.87 (±14.87)	238	5.07 (±1.18)	254.00 (±0.00)	254	34.20 (±0.19)	254.00 (±0.00)	254	**3.47 (±0.74)**
Themis_Jetty	Safe	11.77 (±0.60)	15	3.77 (±0.72)	**21.00 (±0.00)**	21	57.70 (±1.03)	13.80 (±1.02)	23	4.20 (±0.96)
Themis_Jetty	Unsafe	70.87 (±6.12)	105	6.83 (±1.62)	98.00 (±0.00)	98	58.50 (±0.51)	**100.53 (±1.37)**	111	5.90 (±1.12)
STAC_ibasys	Unsafe	129.40 (±19.52)	280	41.60 (±3.17)	280.00 (±0.00)	280	70.30 (±3.36)	280.00 (±0.00)	280	45.63 (±3.11)
RSA	1717	107.27 (±1.22)	112	2.20 (±0.23)	108.00 (±0.60)	112	3.47 (±0.18)	108.17 (±1.10)	116	2.33 (±0.38)
RSA	834443	186.77 (±1.75)	196	1.87 (±0.18)	183.63 (±0.89)	190	3.57 (±0.18)	184.50 (±1.24)	191	1.73 (±0.23)
RSA	1964903306	272.77 (±2.47)	286	1.90 (±0.28)	252.00 (±0.00)	252	3.30 (±0.16)	275.93 (±3.17)	307	1.93 (±0.26)
RSA (30s)	1717	85.00 (±6.14)	104	2.20 (±0.23)	**102.00 (±0.00)**	102	3.47 (±0.18)	97.80 (±0.74)	104	2.33 (±0.38)
RSA (30s)	834443	152.93 (±6.58)	187	1.87 (±0.18)	173.23 (±1.04)	183	3.57 (±0.18)	172.80 (±0.80)	181	1.73 (±0.23)
RSA (30s)	1964903306	226.67 (±7.94)	262	1.90 (±0.28)	252.00 (±0.00)	252	3.30 (±0.16)	**254.27 (±1.75)**	269	1.93 (±0.26)

output difference expected [5]. DF and DSE in contrast, cannot classify all subjects correctly.

In terms of time to find the first output difference, DF is not significantly faster than HyDiff, except for the subject *CLI5-6*, although it does only mean 2 seconds performance benefit in absolute values. In fact in most cases HyDiff is faster than DF.

In comparison to DSE, HyDiff is most of the times comparable, but DSE is also significantly faster for some subjects (e.g., *TCAS-1* or *TCAS-7*). In such cases the differential symbolic execution part of HyDiff is kept busy with importing inputs from the fuzzer, which holds off its own analysis progress. This is a problem, which could be tackled with a more fine-tuned configuration of HyDiff. Note that for most of the cases the absolute difference between HyDiff and DSE is just a couple of seconds.

In terms of actually finding indicators for a regression, namely output and decision differences, HyDiff shows its benefits. For the most of the subjects HyDiff finds a comparable or larger number of output differences (+odiff) than the single parts. For example for *Time-1* and *CLI5-6* HyDiff can identify way more output differences. In addition, for all cases, except *Math-46* and *Math-60*, HyDiff can identify significantly more decision differences (+ddiff). In general, the inputs imported by symbolic execution are useful to push the exploration, but in this special case symbolic execution does not perform very good in terms of the number of generated paths.

Summarized, our results show that HyDiff clearly outperforms the single techniques in terms of identifying regressions, whereas at the same time, HyDiff only loses some seconds in contrast to DSE to identify the first output difference.

4.3 Side-Channel Analysis

Side-channels are dangerous because they allow an adversary to uncover secret program data from observations made over the non-functional behavior of a program with respect to a resource consumption, such as execution time, consumed memory or response size. Traditional techniques for detecting side-channels involve the analysis of two program copies via self-composition [3] in an attempt to find two secret-dependent paths that lead to a noticeable difference in resource consumption. If no such difference is found, this corresponds to the classic notion of *non-interference* meaning no vulnerability was found. If, on the other hand, a difference is found, this corresponds to a vulnerability that needs to be fixed. To perform side-channel analysis with HyDiff, we need to fuzz three values: one public value, and two secret values (the approach naturally extends to tuples of values). The fuzzing driver calls the program twice, each with the same public value, but with another secret value. We measure the cost difference between both executions, but also the decision difference and the output difference. We use all metrics to drive the fuzzing process, but at the end the most interesting is the cost difference, which is a measure for the severity the side-channel vulnerability.

For the symbolic execution part we leverage the change-annotations to model the change in the secret part of the input: $secret = change(secret_1, secret_2)$. This assignment directly occurs in the driver, and hence, the program itself does not contain any change annotations. Therefore, the patch distance is not relevant in this setting. The differential expression gets introduced straight in the beginning, so we do not use any control flow information to prune any node (every node in the trie has already touched the change). This also means that we can use a simpler symbolic exploration strategy for side-channel analysis. In our case we developed the following strategy: (1) Prioritize new branch coverage. (2) Prioritize higher cost difference. (3) Prioritize higher nodes in the trie. The primary goal of symbolic execution in the hybrid analysis framework is the support of the fuzzing component by solving complex branching conditions, which are infeasible for fuzzing. Hence, we prioritize the increase in branch coverage during symbolic exploration (point 1). The second goal is to find inputs that increase the cost difference, since they signal side-channel vulnerabilities (point 2). Finally, we prefer nodes higher in the trie (closer to the root node) because this likely leads to a broader exploration of the search space (point 3). Note that it is easy to change the strategies so that different analysis types can be incorporated. In case of the presented strategy for the side-channel analysis, we also experimented with different orders in the prioritization, but we did not notice an improvement. Side-channel analysis is concerned with finding differences with regard to non-functional characteristics of the program, which are affected by the input size. In order to be able to handle multiple input sizes, we allow to define a maximum input size, e.g., the maximum length of an input array. The fuzzing driver will read up to the maximum number of values from the input file. For the symbolic execution, the driver will introduce a decision straight in the beginning (before actually calling the application), which determines the input size. This decision will reflect a node straight after the root node in the trie. We note that such an extension of the driver's functionality would be necessary also when incorporating multiple input sizes in the regression analysis.

Table 3: Results for DNN Analysis (t=3600sec=60min, average over 30 runs). The bold values represent significant differences to the closest other subject verified with the Wilcoxon ranked sum test ($\alpha = 0.05$).

%	Differential Fuzzing (DF)				Differential Symbolic Execution (DSE)				HyDiff (SymExe first 10min)			
	$\overline{t}$ +odiff	t_{min}	$\#odiff$	$\#ddiff$	$\overline{t}$ +odiff	t_{min}	$\#odiff$	$\#ddiff$	$\overline{t}$ +odiff	t_{min}	$\#odiff$	$\#ddiff$
1	2725.40 (±341.09)	1074	0.57 (±0.20)	**7.73 (±0.18)**	298.07 (±1.26)	291	1.00 (±0.00)	1.00 (±0.00)	297.10 (±2.38)	267	**1.20 (±0.14)**	6.10 (±0.11)
2	2581.47 (±326.21)	1032	0.93 (±0.28)	**7.93 (±0.13)**	296.33 (±1.16)	291	1.00 (±0.00)	1.00 (±0.00)	297.93 (±1.29)	292	**1.53 (±0.20)**	6.93 (±0.13)
5	2402.97 (±329.59)	1189	1.23 (±0.37)	6.47 (±0.18)	308.83 (±2.66)	301	1.00 (±0.00)	1.00 (±0.00)	**301.83 (±1.16)**	296	**2.03 (±0.29)**	**6.90 (±0.17)**
10	2155.40 (±343.76)	996	1.57 (±0.34)	**8.10 (±0.17)**	327.13 (±4.36)	306	1.00 (±0.00)	1.00 (±0.00)	**311.07 (±1.01)**	306	**2.37 (±0.31)**	7.00 (±0.13)
20	1695.83 (±228.18)	953	2.70 (±0.37)	**9.13 (±0.12)**	344.47 (±1.62)	337	1.00 (±0.00)	1.00 (±0.00)	**341.83 (±1.27)**	336	3.13 (±0.34)	7.20 (±0.14)
50	1830.83 (±259.79)	1220	2.43 (±0.42)	6.33 (±0.21)	**449.33 (±1.25)**	442	1.00 (±0.00)	1.00 (±0.00)	452.63 (±2.06)	434	**3.77 (±0.34)**	**7.27 (±0.16)**
100	1479.17 (±231.25)	960	2.47 (±0.37)	**9.37 (±0.20)**	581.77 (±2.51)	570	1.00 (±0.00)	1.00 (±0.00)	**575.13 (±2.65)**	564	**3.10 (±0.35)**	7.60 (±0.18)

Yet, our regression subjects only considered simple input types, for which the input size was not relevant.

4.3.1 Subjects. For the evaluation we selected subjects from [29], previously analyzed in [1, 9], which represent state-of-the art in side-channel analysis: Blazer_login (25 LOC) and Themis_Jetty (17 LOC), and a sophisticated authentication procedure STAC_ibasys (707 LOC) from [29] which handles complex image manipulations. They come in *unsafe* and *safe* variants, where the *safe* variant usually means that it does not leak any information. The subject *Themis_Jetty Safe* is known to still leak information (but the difference in cost is small). Additionally we analyzed an implementation of modular exponentiation, RSA_modpow (30 LOC), from [39] which has a timing side channel that is due to an optimized step in the exponentiation procedure. In [22] it was shown how a similar vulnerability was exploited to break RSA encryption/decryption. In our experiments we used three different values for modulo.

4.3.2 Results. We collected three metrics: $\overline{\delta}$ denotes the average cost difference (= δ) until timeout, δ_{max} denotes the maximum δ over all runs and $\overline{t} : \delta > 0$ denotes the average time (sec) until the first δ greater than zero has been observed.

As shown in Table 2, HyDiff can detect all side-channels, i.e., it finds cost differences (δ) greater than zero for all unsafe examples. DSE alone needs quite a long time to actually find its first $\delta > 0$, but then it performs well in maximizing it. DF quickly discovers an input with $\delta > 0$, but needs longer to actually find a large value for δ. HyDiff represents the perfect combination of both single techniques: the speed of fuzzing and the reasoning strength of symbolic execution. HyDiff finds very large values for δ comparable to DSE, and finds its first $\delta > 0$ in a comparable time as DF. Both are important factors for the identification and the assessment of side-channel vulnerabilities.

The RSA subjects do not clearly show the same results: DF performs quite similar as DSE, and hence, HyDiff does not show any clear benefit compared to just DF. In order to understand this subject better, we also reported the values after 30 sec of the experiments. These values show that DSE is able to generate a high δ value right in the beginning, whereas fuzzing needs more time. For HyDiff this means the following: the impact of symbolic execution on HyDiff for this subject would be only visible in the first seconds of the experiment. But after 30 min, all techniques show similar results.

4.4 Analysis of Deep Neural Networks

We also propose here a non-standard application of differential analysis to adversarial generation in neural networks with piecewise linear activation functions. The analysis of neural networks is notoriously hard, due to the huge number of paths, allowing us to evaluate HyDiff in domains of high complexity.

Given a DNN model, we first re-write it into the Java program form. While the Java translation preserves the prediction ability of the original DNN, the advantage is that program analysis techniques and tools can now be applied for analyzing the DNN model. Specifically, we used HyDiff to find adversarial inputs for an image classification network. These inputs should differ only slightly, but should lead to different classifications. This problem seems to naturally fit into the differential analysis context. Our idea is to change up to x% of the pixels in the image, and check if there can be any difference in the output of the network, i.e., the final classification.

For the fuzzing driver we therefore fuzz values for the complete image, and we fuzz values and positions of the pixels to change. Hence the fuzzing driver will have two images that differ in only up to x% of the pixels. Then it executes the DNN model with both inputs and measures the metrics similar to regression testing. Of course we are particularly interested in the output difference.

For the symbolic execution we introduce changes, similar as for the side-channel analysis, directly in the driver, so that we execute the DNN model only once, but with a change-annotated input. The DNN analysis specifically searches for differences in the classification, i.e. output differences similar to regression, and not cost differences like the side-channel analysis. Therefore, we use the same heuristics as for regression testing,

4.4.1 Subjects. Particularly, in this experiment, we trained a DNN model for handwritten digit recognition using the MNIST dataset [24]. The data set comes with a training set of 60,000 examples and a test set of 10,000 examples. The trained model has an accuracy of 97.95% on the test set. It consists of 11 layers including convolutional/max-pooling/flatten/dense layers with Rectified Linear Unit (ReLU) activations, contains 10,000 neurons, and uses the max function in the final classification layer.

4.4.2 Results. For DNN analysis we collected the same four metrics as for regression analysis (cf. Section 4.2.2). As shown in Table 3, HyDiff can be used for the analysis of DNNs as it finds output differences in the classification of an image when changing only up to 1% of its pixels.

For the DNN subjects we can make the following observations: differential fuzzing and differential symbolic execution also find the output differences but perform very differently. Differential fuzzing becomes faster in finding an output difference with more pixels allowed to change, which seems reasonable as it is easier to find differences in the classification of two images, when they differ largely. But even for a 100% change differential fuzzing still

needs very long to find an output difference, which is caused by the very long runtime of each program execution. Differential fuzzing executes the same program execution several times to assess the input, which makes this step even more expensive. In contrast, differential symbolic execution becomes slower with more pixels to change. First of all, it does not need to execute the whole program to generate an input because it might be enough to just look at the first two branches to extract a path condition, which leads to another classification. Secondly, with more pixels to change, symbolic execution will introduce more symbolic variables, which will eventually make the path constraints quite complex, and hence, it needs more time for them to be solved. Our results indicate that due to the complexity of the DNN, DSE can generate only one *interesting* input within the given time bound of 60min.

With HyDiff in its default setup (i.e., differential fuzzing and differential symbolic execution start at the same time), there does not happen any synchronization between both components because the expensive program execution. HyDiff's differential fuzzing does not use any inputs from differential symbolic execution and vice versa because they are busy with their own analysis. For this reason HyDiff does simply produce the combined result of both components running separately. Therefore, we decided to alter the setup for the DNN analysis: we start differential symbolic execution first, and with a 10min delay we start differential fuzzing, so that fuzzing can use the already generated inputs from symbolic execution as additional seed inputs. The result for this setup (cf. Table 3) show that HyDiff combines both techniques very well: HyDiff shows comparable times to the first output difference as differential symbolic execution (which is significantly faster than fuzzing), but shows significantly more output differences than fuzzing. Another observation is that the confidence intervals for HyDiff are much smaller than for differential fuzzing, which shows that the results are much more robust. So HyDiff does not only combine the results of both components, but the components can benefit from each others inputs to further improve the outcome.

4.5 Result Summary

HyDiff does not only combine the outcomes of fuzzing and symbolic execution. It does perform a continuous synchronization across output queues, which helps each part to find even more divergences. For example in *Math-60*, *Time-1*, *CLI3-4*, and *CLI5-6* (cf. Table 1), HyDiff finds significantly more output differences than the single components. In particular for *CLI3-4* and *CLI5-6* DSE itself does not find any of them, but the inputs for decision differences identified by symbolic execution push HyDiff's differential fuzzer in the right direction. For the side-channel analysis HyDiff shows a good trade-off between fuzzing and symbolic execution, and in particular for *Themis_Jetty Unsafe* the hybrid technique also outperforms the single components in terms of the generated δ values (cf. Table 2). In the DNN analysis HyDiff finds significantly more output differences, and for the majority of the subjects HyDiff is also faster in finding the first output difference (cf. Table 3).

4.6 Limitations

In addition to the benefits, it is important to discuss potential limitations of our approach and evaluation. Firstly, there is the required manual effort to prepare a program for differential analysis by HyDiff. As mentioned in our motivating example (Sec. 2.2), our approach needs drivers to parse the input and call the program-under-test. Additionally, our approach expects information about the syntactic changes in the program (e.g., for regression testing). For fuzzing the change locations need to be specified, and for symbolic execution the program needs to be annotated with the change-annotations. In order to ensure reproducibility of our evaluation, we make the test drivers and annotations publicly available [44]. We believe that these steps can be automated to a large extend.

Secondly, the purpose of our evaluation was to demonstrate the versatility of HyDiff in a few case studies from very different domains. We cannot claim, the results for each case study will generalize for other kinds of programs (or classifiers) written in other languages or from other domains. In order to mitigate the impact of randomness on the results, we repeated each experiment 30 times and report 95%-confidence intervals.

Lastly, HyDiff is inherently parallel while its components are not. The differential fuzzing (DF) and symbolic execution (DSE) components are run in parallel to boost their advantages and mitigate their weaknesses. These components communicate via a shared queue. In our evaluation, we compare one instance of HyDiff with one instance for each of its constituent techniques. Technically, this gives more computational resources to HyDiff. While DSE does *not* support a parallel mode with a shared queue, to validate our results, we conducted experiments running two instances of DF with a shared queue (Parallel DF). For the regression analysis, HyDiff still outperformed Parallel DF in terms of time to discovering an output difference ($\overline{t}$*+odiff*) for about the same number of subjects as DF. Parallel DF was not able to identify output differences for subjects, for which DF also did not find any output difference, i.e. the parallel variant was not able to solve the actual limitations of DF. However, Parallel DF produced more test cases that reveal a decision or output difference ($\overline{\#ddiff}$ and $\overline{\#odiff}$) — which is expected as Parallel DF also generates about twice as many test cases. We note that multiple test cases may reveal the same output difference. To classify output differences, we would need to conduct a post-processing to further classify the outputs, e.g. in expected and unexpected behavior, similar to [33]. For the side-channel subjects, Parallel DF showed a slightly better performance compared to DF ($\overline{t} : \delta > 0$), but without outperforming HyDiff. For the DNN subjects, Parallel DF also improved the number of identified test inputs for decision and output differences, but without any improvement on the time to discovering an output difference ($\overline{t}$*+odiff*). We further conducted experiments with giving DSE twice the time budget of HyDiff, but for which DSE did not show any significant improvement.

5 RELATED WORK

Existing differential testing techniques include symbolic execution-based [6, 33, 37, 50] and fuzzing-based [14, 17, 29, 32, 38] techniques. With regard to *symbolic execution-based techniques*, HyDiff leverages four-way forking in an incremental manner and introduces novel heuristics to maximize divergence. With regard to *fuzzing-based techniques*, HyDiff contributes a novel divergence-based feedback-channel to greybox fuzzing and leverages several fitness

functions to evaluate and maximize the divergence across multiple program executions.

Differential symbolic execution. Most related to HyDiff is shadow symbolic execution (SSE) [33], which we already discussed throughout the paper. SSE uses four-way forking along the path(s) of a concrete input that reaches a change. HyDiff is more general as it also follows branches which are not already affected by a change, but still can reach a change annotation. This allows HyDiff to detect more divergences (improving upon effectiveness). Novel search-heuristics and the integration with fuzzing allow HyDiff to detect divergences faster (improving upon efficiency). Other related work includes directed incremental symbolic execution (DiSE) [37, 50] which leverages program slicing and symbolic execution along these slices to cover changed statements in the source code. Such techniques could not be readily integrated with a fuzzer, as do not consider any analysis along concrete inputs. Xu et al. [47–49] present several techniques to maintain coverage-adequacy of a test suite after the program was changed. However, unlike HyDiff, this stream of works only considers a single version at a time. Furthermore they are only applicable to regression generation without any consideration about costs.

Differential fuzzing. Bert [17, 32] is a blackbox regression testing technique which generates random input values to expose behavioral differences. In contrast, the greybox fuzzing component of HyDiff is guided by divergence-feedback. NEZHA [38] implements both a blackbox and a greybox fuzzing approach, showing better performance with the greybox component. DifFuzz [29] exposes side-channel vulnerabilities in programs using a resource-based feedback-channel for greybox fuzzing as well as resource-related search heuristics; in contrast to DifFuzz our fuzzing component incorporates more sophisticated differential metrics (as DifFuzz only looks at cost differences). DLFuzz [14] uses greybox fuzzing for differential testing of DNNs. In contrast to all these greybox fuzzing approaches, HyDiff leverages a novel divergence-based feedback channel and a combination of differential metrics that includes output difference, decision difference, and patch distance. Moreover, HyDiff works with increased effectiveness due to the integration with a differential symbolic execution engine.

Hybrid techniques. The integration of efficient fuzzing and effective symbolic execution-based techniques is an active area of research [7]. There exist several works that attempt to integrate both approaches [8, 30, 41]. Like HyDiff, most of these hybrid approaches demonstrate significant benefits of combining the random, collateral path exploration of fuzzing with the systematic path enumeration of symbolic execution. To the best of our knowledge, there does not exist an effective integration of both approaches in the context of differential analysis as it is described here. Badger [30] is a hybrid analysis framework that combines fuzzing and symbolic execution for the worst-case analysis of Java byte code. Similar to HyDiff Badger runs greybox fuzzing and symbolic execution together which synchronize via their queues. However, Badger performs an analysis that does not reason about multiple program paths simultaneously and therefore does not include the advanced features that we presented here. The fuzzing part of Badger only needs to handle a cost metric and the branch coverage measurement, in particular it does not need to further guide the fuzzing process to areas that show differences between versions. In contrast, HyDiff's fuzzing part needs to be guided to, e.g., changed code blocks, and needs to be able to handle metrics like output and decision difference. The symbolic execution part of Badger simply performs a concolic execution, where nodes get selected for exploration based on their cost value. In contrast, HyDiff's symbolic execution needs to handle multiple program behaviors, and hence, uses a modified form of shadow symbolic execution which was extended as we described in this paper.

Adversarial testing of DNNs. Recently, several quantitative coverage metrics [20, 26, 35] have been proposed to guide the testing of DNNs. Different from the adversarial testing works [27, 42, 45] for DNNs so far, in this paper, we adopt a hybrid analysis approach. Moreover, when applying HyDiff to DNNs, we benefit from reusing existing software testing tools such as AFL and SPF.

Differential Verification. Demonstrating the functional equivalence of two programs for *all* inputs is known as differential or regression verification. For example, Lahiri et al. [23] present the tool SymDiff, which searches for output differences by leveraging the modular verifier Boogie [2] to generate the verification condition and the SMT solver Z3 [10] to solve it. HyDiff's analysis is not limited to detect output differences, but also incorporates metrics to detect, e.g., cost differences. Hawblitzel et al. [15] propose an automated full-system verification including proving non-interference of the information flow. Compared to such differential verification approaches, while it would be interesting to explore the provision of statistical guarantees [4], HyDiff cannot provide formal guarantees about the absence of a behavioral differences. However, the lack of formal guarantees is a tradeoff for the scalability necessary for the analysis of real-world applications.

6 CONCLUSION

We proposed a hybrid differential program analysis approach, HyDiff, combining fuzzing and symbolic execution, which are specially tailored to differential analysis and which support each other to amplify the exploration. The evaluation presents three applications: regression testing, side-channel analysis and adversarial generation for deep neural networks. We showed that even when the single techniques failed to find any divergences, HyDiff succeeded. Additionally, our hybrid analysis outperforms the single techniques in terms of the time to the first divergence and the total number of identified divergences.

In the future we plan to extend our case studies and experiment with additional heuristics, to continue the investigation of the differential analysis for deep neural networks, and to explore more types of change annotations to strengthen the applicability of our differential symbolic execution.

ACKNOWLEDGMENTS

This research was partially funded by the Australian Government through an Australian Research Council Discovery Early Career Researcher Award (DE190100046), by the NSF Grant CCF 1901136, and by the German Research Foundation (GR 3634/4-1 EMPRESS).

REFERENCES

[1] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving

the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 362–375. https://doi.org/10.1145/3062341.3062378

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387.

[3] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* 100–114. https://doi.org/10.1109/CSFW.2004.1310735

[4] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. *ACM Transactions on Software Engineering and Methodology* 27, 2, Article 7 (June 2018), 52 pages. https://doi.org/10.1145/3210309

[5] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. 2013. Partition-based Regression Verification. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 302–311. http://dl.acm.org/citation.cfm?id=2486788.2486829

[6] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. 2013. Regression Tests to Expose Change Interaction Errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 334–344. https://doi.org/10.1145/2491411.2491430

[7] Marcel Böhme and Soumya Paul. 2016. A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Transactions on Software Engineering* 42, 4 (April 2016), 345–360. https://doi.org/10.1109/TSE.2015.2487274

[8] Sang K. Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*. 380–394. https://doi.org/10.1109/SP.2012.31

[9] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 875–890. https://doi.org/10.1145/3133956.3134058

[10] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[11] Apache Software Foundation. 2019. Commons BCEL. https://commons.apache.org/proper/commons-bcel/. Accessed: 2020-01-24.

[12] Apache Software Foundation. 2019. Commons CLI. https://commons.apache.org/proper/commons-cli/. Accessed: 2020-01-24.

[13] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[14] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Păsăreanu (Eds.). ACM, 739–743. https://doi.org/10.1145/3236024.3264835

[15] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 165–181.

[16] INRIA. 2019. ASM: a very small and fast Java bytecode manipulation framework. https://asm.ow2.io. Accessed: 2020-01-24.

[17] Wei Jin, Alessandro Orso, and Tao Xie. 2010. Automated Behavioral Regression Testing. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society, 137–146. https://doi.org/10.1109/ICST.2010.64

[18] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[19] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2511–2513. https://doi.org/10.1145/3133956.3138820

[20] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1039–1049. https://doi.org/10.1109/ICSE.2019.00108

[21] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[22] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '96)*. Springer-Verlag, London, UK, UK, 104–113. http://dl.acm.org/citation.cfm?id=646761.706156

[23] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 712–717.

[24] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. 2013. MNIST database. http://yann.lecun.com/exdb/mnist/. Accessed: 2020-01-24.

[25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990.

[26] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 120–131. https://doi.org/10.1145/3238147.3238202

[27] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 100–111. https://doi.org/10.1109/ISSRE.2018.00021

[28] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage Testing of Software Patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/2491411.2491438

[29] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. 2019. DiffFuzz: Differential Fuzzing for Side-channel Analysis. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 176–187. https://doi.org/10.1109/ICSE.2019.00034

[30] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. 2018. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 322–332. https://doi.org/10.1145/3213846.3213868

[31] Yannic Noller, Hoang Lam Nguyen, Minxing Tang, and Timo Kehrer. 2018. Shadow Symbolic Execution with Java PathFinder. *SIGSOFT Softw. Eng. Notes* 42, 4 (Jan. 2018), 1–5. https://doi.org/10.1145/3149485.3149492

[32] Alessandro Orso and Tao Xie. 2008. BERT: BEhavioral Regression Testing. In *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008) (WODA '08)*. ACM, New York, NY, USA, 36–42. https://doi.org/10.1145/1401827.1401835

[33] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a Doubt: Testing for Divergences between Software Versions. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 1181–1192. https://doi.org/10.1145/2884781.2884845

[34] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20, 3 (2013), 391–425. https://doi.org/10.1007/s10515-013-0122-2

[35] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785

[36] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential Symbolic Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 226–237. https://doi.org/10.1145/1453101.1453131

[37] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 504–515. https://doi.org/10.1145/1993498.1993558

[38] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 615–632. https://doi.org/10.1109/SP.2017.27

[39] Corina S. Păsăreanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 387–400. https://doi.org/10.1109/CSF.2016.34

[40] SIR. 2019. Software-artifact Infrastructure Repository. http://sir.unl.edu. Accessed: 2020-01-24.

[41] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. https://doi.org/10.14722/ndss.2016.23368

[42] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 109–119. https://doi.org/10.1145/3238147.3238172

[43] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[44] HyDiff Replication Package v1.0.0. 2020. https://doi.org/10.5281/zenodo.3627893.

[45] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. 2019. Adversarial Sample Detection for Deep Neural Network Through Model Mutation Testing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1245–1256. https://doi.org/10.1109/ICSE.2019.00126

[46] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 359–368. https://doi.org/10.1109/DSN.2009.5270315

[47] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. 2013. Continuous Test Suite Augmentation in Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC '13)*. ACM, New York, NY, USA, 52–61. https://doi.org/10.1145/2491627.2491650

[48] Zhihong Xu, Yunho Kim, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. 2015. Directed test suite augmentation: an empirical investigation. *Software Testing, Verification and Reliability* 25, 2 (2015), 77–114. https://doi.org/10.1002/stvr.1562

[49] Zhihong Xu, Yunho Kim, Moonzoo Kim, and Gregg Rothermel. 2011. A Hybrid Directed Test Suite Augmentation Technique. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. 150–159. https://doi.org/10.1109/ISSRE.2011.21

[50] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed Incremental Symbolic Execution. *ACM Trans. Softw. Eng. Methodol.* 24, 1 (2014), 3:1–3:42. https://doi.org/10.1145/2629536

[51] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 144–154. https://doi.org/10.1145/2338965.2336771

[52] Michal Zalewski. 2014. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/. Accessed: 2020-01-24.