Registered Report: Generating Test Suites for GPU Instruction Sets through Mutation and Equivalence Checking

Shoham Shitrit
University of Rochester
sshitrit@u.rochester.edu

Sreepathi Pai
University of Rochester
sree@cs.rochester.edu

***Abstract*—Formal semantics for instruction sets can be used to validate implementations through formal verification. However, testing is often the only feasible method when checking an artifact such as a hardware processor, a simulator, or a compiler. In this work, we construct a pipeline that can be used to automatically generate a test suite for an instruction set from its executable semantics. Our method mutates the formal semantics, expressed as a C program, to introduce bugs in the semantics. Using a bounded model checker, we then check the mutated semantics to the original for equivalence. Since the mutated and original semantics are usually not equivalent, this yields counterexamples which can be used to construct a test suite. By combining a mutation testing engine with a bounded model checker, we obtain a fully automatic method for constructing test suites for a given formal semantics. We intend to instantiate this on a formal semantics of a portion of NVIDIA's PTX instruction set for GPUs that we have developed. We will compare to our existing method of testing that uses stratified random sampling and evaluate effectiveness, cost, and feasibility.**

I. Introduction

Formal semantics for the instruction sets of many processors are now available [1, 2]. These semantics can be used to validate processor implementations through formal verification. In many cases, software interpreters that are correct-by-construction can be automatically extracted from these semantics. However, formal verification cannot be used or is too expensive for many implementations of instruction sets. A user who only has access to hardware processor and no access to the internals has no way of formally verifying it. Tools such as high-performance virtual machines, dynamic binary translators, compilers, etc. all utilize instruction set semantics but are very difficult to verify formally.

Exhaustive testing can be used to validate implementations of instruction sets and, in theory, achieve the same level of assurance as verification. However, the input space of even individual instructions is so large as to render exhaustive testing impractical. Randomized testing can be used instead to achieve a high level of assurance but sampling strategies can significantly affect the quality of the test suite. Moreover, both exhaustive and randomized testing do not make use of the formal semantics, a point that is usually in their favour. In this work, we explore how formal semantics can be used to seed an effective and automatic test generation strategies for instruction sets.

Our work is essentially model-based test generation with the formal semantics specifying the model. The crux of our test generation idea can be traced back to Carrington and Stocks [3] where formal specifications expressed in Z were mutated and tests generated to distinguish the mutant specifications from the original specification. Our work in this paper is based on a similar idea, but our specifications describe instruction sets, are encoded in C, and we use model checking to verify equivalence. This is implemented in a fully automatic test suite generation pipeline. Our evaluation focuses on an empirical analysis of our pipeline and its suitability in generating tests for a real-world instruction set.

We focus on NVIDIA's PTX instruction set [4], a virtual instruction not unlike LLVM IR, but for NVIDIA's GPUs. NVIDIA only provides an informal description of PTX. We have formalized this description in a custom domain-specific language from which we extract the C semantics used in this work. Our method is restricted to instructions whose behaviour is completely defined by their inputs. More complicated instructions whose behaviour depends on internal, implementation-specific, invisible GPU state are not supported by our current framework. These instructions do not consume inputs or produce outputs in the conventional sense. Concretely, this means instructions that enforce behaviour like memory consistency, synchronization, etc. are out of scope for our technique. These instructions are tested using orthogonal strategies like litmus tests [5].

Our contributions are as follows:

- We construct and evaluate a test generation pipeline that uses mutation testing to introduce bugs in an executable specification and equivalence checking to generate inputs to detect those bugs.
- We describe an alternate method for constructing test suites based on stratified random sampling to serve as a point for comparison.
- We find that our pipeline can construct inputs that expose bugs that are missed by stratified sampling-based method and at relatively low cost in CPU time.

International Fuzzing Workshop (FUZZING) 2022
27 February 2022, Virtual
ISBN 1-891562-77-0
https://dx.doi.org/10.14722/fuzzing.2022.23xxx
www.ndss-symposium.org

```
float execute_add_rm_ftz_sat_f32 (float
    src1, float src2) {
        float tmp_dst;
        float dst;
        src1 = FTZ(src1);
        src2 = FTZ(src2);
        tmp_dst = SATURATE(ADD_ROUND(src1,
             src2, FE_DOWNWARD));
        tmp_dst = FTZ(tmp_dst);
        dst = tmp_dst;
        return dst;
}
```

Listing 1. The semantics of the PTX `add.rm.ftz.sat.f32` instruction expressed as a C program.

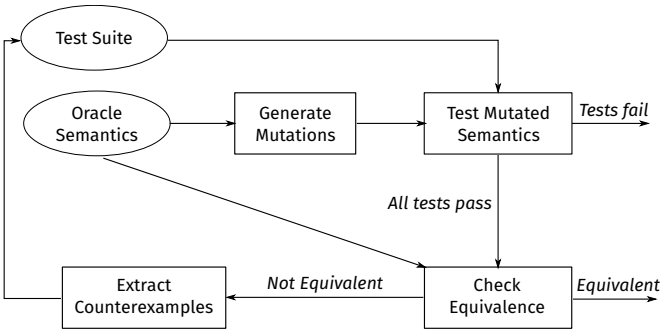

Fig. 1. A pipeline for generating test suites based on mutation generation and equivalence checking.

- We find some bugs in C code can introduce non-determinism which makes testing for them nearly impossible without specialized instrumention.
- We evaluate the possibility of generating a test suite fully using our technique and find it to be time consuming, compared to augmenting an existing test suite

We present preliminary data on 12 PTX instructions which operate on floating-point inputs and that comprise arithmetic, logical and elementary math functions.

II. Motivating Example

To provide an overview of our technique, we demonstrate it on Listing 1 which is a C program semantically equivalent to the PTX `add.rm.ftz.sat.f32` instruction. The instruction adds two 32-bit floating point numbers together rounding towards negative infinity (`.rm`) and saturating the result to lie between $[0.0, 1.0]$, while flushing subnormal inputs and output floating point numbers to zero (`.ftz`).

In the implementation, the `FTZ` function checks if a floating point number is a subnormal and returns a zero with the sign of the original number when it is, or the original number otherwise. The `SATURATE` function forces the output of the rounding add (`ADD_ROUND`) to lie in $[0.0, 1.0]$ by saturating negative values to 0.0 and positive values greater than 1.0 to 1.0.

Consider now a real-life buggy implementation of these semantics in the CUDA compiler where `FTZ` was not applied to `src1` and `src2`. To detect this bug, the values of `src1` and `src2` must obviously be subnormal floats. However, their *sum* must be a normal number to avoid the bug being masked by the `FTZ` on `tmp_dst`. Using similar reasoning, it is easy to show that if a buggy implementation omitted the `FTZ` on the sum (i.e. `FTZ(tmp_dst)` but preserved the `FTZ` applications on the inputs, the bug will only be caught if the inputs are normal floating point values that sum to a subnormal value. Since the inputs and outputs must obey a specific relationship to detect these bugs, existing methods such as random sampling and even branch coverage can fail to generate inputs that will unearth the bug.

Yet, given the buggy implementation and the original semantics, it is possible, in theory, to check the two for equivalence. Since the semantics are expressed as C programs, any tool that can check equivalence of C programs can be used. In our work, we use CBMC [6], a bounded model checker for C programs. The check asserts that for all inputs x and y that are floating point numbers, the two implementations are equal:

```
float x, y;
assert(add_rm_ftz_sat_f32_original(x, y) ==
       add_rm_ftz_sat_f32_buggy(x, y));
```

When the two functions are not equivalent, CBMC will yield counter-examples that cause this assertion fails. These counter-examples can then be added to the test suite.

We note that equivalence checking is undecidable in general. However, several features of instruction set semantics make this less of an issue in practice. First, instruction set semantics are usually simple, performing bitwise manipulations on fixed-size inputs. Second, loops if present, are almost always bounded. Thus, checking the equivalence of instruction set semantics usually boils down to checking the equivalence of small, straight-line programs, a far more tractable task.

Given an equivalence checker and the correct semantics, we can now generate a test suite by creating buggy versions of the semantics and identifying the inputs that distinguish the buggy versions from the original semantics. This test suite can then be used to validate implementations which cannot be formally verified. We use mutation testing to generate the buggy versions of the semantics.

III. Implementation

Our framework for test suite generation for instruction set semantics using mutation testing and equivalence checking is shown in Figure 1.

A. Overview

We are provided with an oracle semantics for the NVIDIA's PTX instruction set for its GPUs as semantically equivalent C programs. Such C programs can be extracted from an existing formal semantics such as those encoded in SAIL [7] but executable models can also be constructed from formal specifications written in K [8]. An existing test suite consisting of input–output pairs for each instruction is also provided. This test suite is not strictly necessary. When provided, however, our technique will only focus on mutations that are not detected

by the existing test suite. Thus, our technique can improve the quality of an existing test suite. In the following discussion, we always assume a test suite is provided since an empty test suite is acceptable as well.

The oracle semantics for each instruction are handed to a mutation generator. The resulting mutants can be regarded as buggy versions of the oracle semantics, where the bugs are the mutations that were introduced. These mutants are tested against the existing test suite. If tests fail for all mutations, the pipeline stops, as the existing test suite is strong enough to detect all introduced mutations.

Each mutant that survives the test suite proceeds to the equivalence checker, a step that does not usually exist in typical mutation testing scenarios. The equivalence checker will consider one mutation at a time and verify equivalence with the oracle semantics. Since a mutation–oracle pair is usually not semantically equivalent, this verification will fail. With an appropriate tool, a counter-example can be extracted showing that the same inputs produce different outputs for the mutant and the oracle. This test case differentiates the oracle and the mutation and can be added to the test suite.

In general, deciding when to stop this pipeline is unclear. There are a finite number of functions that take two n-bit inputs, but this number is very large. We stop when we run out of mutants. Thus, the quality of the pipeline is heavily dependent on the mutation generator.

B. Mutation Generation and Testing

The mutation framework utilized in this pipeline is MUSIC ("MUtation analySIs tool with high Configurability and extensibility") [9]. MUSIC mutates the AST of the oracle C program and outputs a mutated C program. This allows us to CBMC, the bounded model checker for C, which supports reasoning about floating point arithmetic. Other mutation frameworks (e.g. [10]) could be used if coupled with an equivalence checker that supports floating point operations.

MUSIC produces a number of mutant programs with each mutant containing a single mutation chosen from a pre-defined library of mutations. If these mutants fail existing tests, they are discarded. Only mutants that pass all existing tests and survive move on to the equivalence checking stage.

MUSIC generates all the mutants in a single invocation. However, the testing of each of these mutants can be done in parallel since they are a collection of independent tasks. In our Python implementation of the pipeline, we use the facilities of the built-in `multiprocessing` module to test mutants in parallel.

In this stage of the pipeline, we can measure the effectiveness of the existing test suite by computing a mutation score – the number of killed mutations divided by the total number of mutations. The closer the mutation score is to 1.0, the more mutants the test suite has killed, and thus the more resilient the test suite is to detecting bugs in the program.

C. Equivalence Checking and Testing

A non-empty set of mutant survivors indicates that the existing test suite is incapable of differentiating between them and the oracle semantics. Thus, either all the mutants are semantically equivalent to the oracle semantics or the test suite is incomplete. Traditional mutation testing frameworks rely on the user to investigate these survivors to figure out which of these cases a survivor falls into.

```
/* preceding this and not shown is the
    code from Listing 1 */

float mutated_function(float src1, float
    src2)
{
  float tmp_dst;
  float dst;
  src1 = FTZ(src1);
  src2 = FTZ(src2);
  tmp_dst = SATURATE(ADD_ROUND(src1, src2,
      FE_DOWNWARD));
  ;
  dst = tmp_dst;
  return dst;
}

int main() {
  float variable_0;
  float variable_1;
  float result =
      execute_add_rm_ftz_sat_f32(
      variable_0, variable_1);
  float mutated_result = mutated_function(
      variable_0, variable_1);
  assert(result == mutated_result);
  return 0;
}
```

Listing 2. A mutated version of `add.rm.ftz.sat.f32` C semantics that omits the `FTZ` on the output variable `tmp_dst` with the code in `main` that drives CBMC.

However, an equivalence checker can distinguish these two cases. In our technique, we use a model checker to verify equivalence. For our purposes, model checkers are ideal since they will not only verify equivalence, but also yield counterexamples when the programs are not equivalent.

We use the C Bounded Model Checker (CMBC). CBMC converts C programs into logical models expressed in SMT-LIB language. Then, it discharges assertions about the C program using existing SAT/SMT solvers. If the assertions fail, CBMC will produce a *trace* of an execution of the program that causes the assertion to fail. From this trace, we can extract the input values and use the oracle to generate the expected output. The input–output pair can then be added to the existing test suite.

Consider as an example of this procedure, Listing 2, which shows the mutated version of `add.rm.ftz.sat.f32` and the `main` function that sets up the equivalence test. Note that since `variable_0` and `variable_1` are not initialized, their values are non-deterministic, causing CBMC to check the assertion over all possible values.

Predictably, this mutated version fails the equivalence

check, and we obtain the following values from the CBMC trace, shown here in hexadecimal float notation:

```
variable_0      = -0x1.d953fp-124
variable_1      =  0x1.0a34p-123
mutated_result  =  0x1.d8a08p-127
```

Both the input variables are normal numbers indicated by exponents greater than -127. However, their sum – the output of mutated function – is clearly a subnormal number. Lacking the `FTZ` on `tmp_dst`, this subnormal escapes out as a return value. The original function would have returned 0 in this case. The procedure has generated exactly the inputs we were looking for – two normal numbers that sum to a subnormal.

Model checking is sound and exact. It produces no false positives unlike techniques such as abstract interpretation [11]. However, this means that the primary limitation of model checking is state-space explosion. Therefore, like its name implies, CBMC is limited to *bounded* programs. This means that all loops in the program must have a fixed upper bound so that before model checking CBMC can unroll all loops up to that bound.

Nevertheless, CBMC is a very good fit for our needs. The PTX instruction set semantics when expressed as C programs do have loops, but all of them have finite bounds. We note that none of the programs in this preliminary study have loops. They are also relatively small programs that do not make much use of the C standard library. CBMC also supports reasoning about IEEE floating point semantics [12, 13] which is critical to generate tests for the large number of floating point instructions supported by the GPU.

Like testing mutations, each invocation of CBMC on a survived mutant can be executed in parallel. Similar to that stage, we run equivalence checks in parallel using Python's built-in facilities.

IV. Stratified Random Sampling Test Suite Generation

To compare our proposed test generation framework, we use a test suite generated using stratified random sampling. This technique uses the types of the inputs to generate test cases. Each test case is a randomly sampled value from a predefined and fixed set of strata for a type.

As an example, the strata for 32-bit unsigned integers are defined as the set $\{\{0\}, \{1\}, [2, \text{UINT\_MAX}), \{\text{UINT\_MAX}\}\}$ where each element is itself a set and represents a strata. Here, there are four strata, three of which are the singleton sets containing 0, 1, and UINT_MAX and the fourth consists all the other positive numbers. For an instruction consuming a single unsigned 32-bit integer, our procedure would produce four inputs, one from each strata. Random sampling would only be used for the set $[2, \text{UINT\_MAX})$, the test cases would always contain 0, 1, and UINT_MAX.

For a instruction that takes n inputs, we would select the Cartesian product of the n sample sets to yield the test cases. Thus, all instructions consuming two 32-bit unsigned integers would have 16 test cases. Thus, only type and arity decide the inputs used, not the semantics of the instruction *per se*. The maximum number of test cases produced by our combinatorial strategy is 1728, for 3-input instructions whose arguments are floating point numbers.

TABLE I. Sampling strata for each type of parameter to a PTX instruction. b is the bitwidth of the integer, min and max are the minimum and maximum normal floating point numbers, min_ϵ and max_ϵ are their subnormal analogues, nan is the IEEE754 not-a-number.

Type	Strata
Unsigned integers	$\{0\}, \{1\}, [2, 2^b - 1), \{2^b - 1\}$,
Signed Integers	$\{-2^{b-1}\}, \{0\}, \{1\}, (-2^{b-1}, 2^{b-1} - 1), \{2^{b-1} - 1\}$
Floats (Numbers)	$[-\text{max}, -\text{min}], [-\text{max}_\epsilon, -\text{min}_\epsilon]$, $\{-0.0\}, \{+0.0\}, [+\text{min}_\epsilon, +\text{max}_\epsilon], [+\text{min}, +\text{max}]$
Float (Special)	$\{-\text{nan}\}, \{-\infty\}, \{+\infty\}, \{+\text{nan}\}$

The strata definitions are currently *ad hoc* and aim to maximize parameter coverage as well as incorporate all "boundary" values. In particular, we want to elicit exceptional and undefined behaviour such as divide-by-zero, integer overflows, etc. Since we always take at least one sample from each strata, the elements of the singleton sets in Table I will always be included. Stratified sampling performed significantly better than a pure random sampling strategy that considered every n-bit pattern equally likely. In hindsight, this is not surprising, since bit patterns representing a nan or ∞ are fewer in number than those representing the normal numbers, for example.

Unfortunately, an automatic criteria for defining strata is not known. This means that this is not a fully automatic method, unlike the combination of mutation generation and equivalence checking.

V. Evaluation

The goal of our evaluation is to answer the following research questions. The first five questions naturally arise as observations from each stage of the pipeline. The last three questions investigate properties of our proposed method.

RQ1. *How effective is mutation testing at generating mutants that pass our existing test suite generated by stratified random sampling?* The premise of our method is that it will generate buggy versions that evade our current test suite. Effectiveness is measured by the count of mutants surviving our current test suite that is generated by the stratified random sampling method detailed in Section IV.

RQ2. *For mutants that evade the test suite, how many are caught later by the counter-examples generated by the equivalence checker?* This evaluates if the equivalence checker can indeed generate inputs to detect the mutated versions of the semantics and is measured by the increase in the number of mutants killed by the original test suite when augmented with the newly generated inputs.

RQ3. *How many mutants are only caught by the equivalence checker and continue to escape being caught by the test suite even after it has been augmented with counter-examples?* Ideally, the inputs generated by the equivalence checker should result in the mutant being detected during testing, thus this number should be zero. Surprisingly, we found this is not always the case, pointing to a limitation of testing methods that we did not fully appreciate when we started.

RQ4. *Which mutations lead to syntactic differences but are semantically identical?* To be most useful, every syntactically distinct mutant should also be semantically distinct from the original program. Here, we investigate why some mutants remain semantically identical despite being syntactically distinct, pointing to an additional way in which the quality of mutation generators may be evaluated using equivalence checkers.

RQ5. *What are the costs of this pipeline?* Measured primarily as time consumed, we provide data on how long it takes for each stage of the pipeline.

RQ6. *What is the applicability of this pipeline?* Although we provide preliminary results on 12 arithmetic and logical instructions that consume floats, we possess the semantics for 4085 arithmetic and logic instructions that span integral and floating point types of various sizes. Although, in principle, these are supported by our pipeline, we have not yet investigated this empirically. In particular, we are concerned that some instructions, especially bit-level manipulation instructions, might stress our equivalence checker.

RQ7. *How sensitive is the pipeline to the mutation generator and the equivalence checker?* Although we use MUSIC and CBMC, there are similar tools that can be used in the pipeline. Although none of these support exactly what we want – C-level mutation and support for floating point arithmetic – it would still be interesting to evaluate piecemeal if additional mutations are generated that are missed by the test suite created through our pipeline. Likewise, using a symbolic execution like KLEE [14] for equivalence checking would provide a nice contrast to CBMC.

RQ8. *Can our pipeline replace a stratified random sampling approach?* Our pipeline is fully automated, in contrast to stratified sampling which requires manual specification of the strata. Our primary metric here is cost. It is difficult to evaluate coverage since that would involve finding buggy programs that only our stratified sampling-based test suite will catch which implies those bugs cannot be induced by a mutation-based method.

RQ9. *How effective is a state-of-the-art fuzzer compared to equivalence checking?* It is not hard to imagine replacing the equivalence checker with a fuzzer that attempts to discover the inputs that differentiate the mutants. We will use libFuzzer [15] to compare to equivalence checking using CBMC. The fuzzer will be given the same time that CBMC took. (This question was suggested by a reviewer.)

In this work, we provide preliminary observations for a set of 12 instructions, all operating on floating point inputs. MUSIC (commit `891d9ef`), based on LLVM 7, was used to mutate programs. CBMC 5.38 was used to perform the equivalence checks using the built-in MiniSAT solver. The timing results were obtained on a machine with AMD EPYC 7502P 32-core processor and 256GB of RAM running Ubuntu 18.04LTS. Each reported time is the average of 5 runs along with its 95% confidence interval.

Overall, we find that our proposed pipeline is effective at accomplishing our basic goals – mutation testing does generate buggy mutants that escape our existing test suite, equivalence checking is able to detect semantic differences and generate the exact inputs required to distinguish them, and the cost is low when an existing test suite is used. Generating a test suite from scratch (RQ8) is possible but consumes more time. The results for the RQ1–RQ7 are summarized in Table II and in Table III. Results for RQ8 are presented later in Table IV.

TABLE II. THE FLOW OF MUTATIONS THROUGH THE PIPELINE

Instruction	Total	Kill #1	Same	Kill #2	Left	Time (s)
abs.f32	70	65	3	0	2	1.41 ± 0.05
add.rm.ftz.sat.f32	167	143	2	21	1	2.49 ± 0.13
add.rn.f32	24	23	0	0	1	0.72 ± 0.02
add.sat.f32	128	125	1	0	2	1.86 ± 0.08
set.eq.ftz.s32.f32	302	229	66	2	5	4.23 ± 0.22
set.ge.f32.f32	333	233	79	2	19	4.75 ± 0.04
set.gt.s32.f32	212	139	66	2	5	3.27 ± 0.18
set.gt.u32.f32	212	139	66	2	5	3.25 ± 0.15
setp.ge.f32	378	290	76	2	10	5.18 ± 0.16
sqrt.rm.f32	245	127	0	11	107	3.50 ± 0.16
sub.rn.ftz.sat.f32	167	143	2	21	1	2.65 ± 0.10
sub.rz.ftz.sat.f32	167	143	2	21	1	2.56 ± 0.02

TABLE III. EQUIVALENCE CHECKER TEST GENERATION

Instruction	Generated	Unique	Total	Time (s)
abs.f32	2	2	10	7.59 ± 0.09
add.rm.ftz.sat.f32	22	12	76	44.42 ± 0.38
add.rn.f32	1	1	65	4.02 ± 0.04
add.sat.f32	2	2	66	13.81 ± 0.08
set.eq.ftz.s32.f32	7	6	70	46.82 ± 0.43
set.ge.f32.f32	21	11	75	78.10 ± 0.38
set.gt.s32.f32	7	6	70	47.21 ± 0.28
set.gt.u32.f32	7	3	67	46.48 ± 0.42
setp.ge.f32	12	8	72	61.01 ± 0.41
sqrt.rm.f32	118	24	32	217.67 ± 1.72
sub.rn.ftz.sat.f32	22	12	76	43.97 ± 0.50
sub.rz.ftz.sat.f32	22	12	76	44.15 ± 0.40

A. RQ1: Generating Mutants

The *Total* and *Kill #1* columns of Table II show the number of mutants generated for an instruction and the number of those mutants killed in the first round by our existing test suite. This number also includes mutants that failed to compile or did not execute to completion, though the fraction of mutants killed by the test suite is always above 75%, except for `sqrt.rm.f32` where only 58% of mutants were detected by the test suite. Clearly, the number of mutants is related to syntactic complexity of the original program (not evaluated here) – the `abs.f32` instruction is mostly a single call to the `fabsf` function, whereas all the others are more complicated. The stratified sampling-based test suite does a remarkable job of eliminating a majority of mutants despite being based only on input types and arity.

B. RQ2: Equivalence Checker Effectiveness

The *Kill #2* column of Table II shows the number of mutants that escaped the original test suite, but were killed by a second round of testing when the test suite was augmented by the new inputs produced by the equivalence checker. These mutants were detected as being semantically different from the original program and the counter-examples generated should be enough to detect them. While this is indeed the case for 8 of the 11 instructions whose mutants survived, three instructions – `abs.f32`, `add.rn.f32`, and `add.sat.f32` – had mutants that survived the second round of testing as well. Indeed, to our great surprise, there were *always* mutants that were detected by the equivalence checker to be different, but which always evaded detection during testing. We investigate these in more detail in our discussion of RQ3.

Table III shows the results of input generation by the equivalence checker. While the *Generated* column shows the number of new inputs generated, a deduplication pass based on string equality results in far fewer *Unique* inputs being added to the test suite resulting in a final test suite containing *Total* inputs. This suggests that many mutants themselves are semantically identical to each other and this similarity could be exploited to reduce the cost of equivalence checking. For example, equivalence checking could be run only on a random sample of mutants that survived a round of testing. The inputs generated could be used to weed out other mutants in a substantially cheaper "mini-round" of testing to save on the cost of equivalence checking.

C. RQ3: Mutants that Evade Tests

The *Left* column in Table II represents the mutants that survive two rounds of testing, with the second round containing the inputs generated by the equivalence checker. This is logically possible but unexpected.[1] This could be when the equivalence checker flags a mutant as being non-equivalent when it is actually equivalent. This is benign and apparent, but can complicate analysis. But another reason for this happening is the presence of *non-determinism* in the test. However, all our instructions have deterministic behaviour. Nevertheless, we find that the mutation generation process introduces non-determinism in subtle ways by generating code that relies on undefined behaviour.

An excerpt from `abs.f32` shows how a read from an uninitialized variable is introduced into `abs.f32` when an assignment is changed to +=:

```
float tmp_dst;
tmp_dst += fabsf(src);
```

This mutation will go undetected if `tmp_dst` contains the value 0.0 or if the compiler optimizes the undefined read away. The latter is the case on our test system.

Although `sqrt.rm.f32` contains a large number of mutants that evade both rounds of testing, the reason is that it uses the C library `sqrtf` function. CBMC uses its own built-in implementation that is known to produce non-deterministic results when the input is a subnormal because IEEE subnormals have the unusual property that *two* numbers can be squared to obtain them. The current implementation returns one of these numbers non-deterministically resulting in spurious equivalence failures. Although a path to a possible fix is known [16], it has not been implemented as of writing.

The `set.ge.u32.f32` instruction contains a different and more complicated instance, where the comparison:

```
pred2 = (!(isnan(src1) || isnan(src2)))
   && (src1 >= src2);
```

is mutated to:

```
pred2 = (!(isnan(src1) % isnan(src2))) &&
    (src1 >= src2);
```

CBMC recognizes that when both `src1` and `src2` are not not-a-number (NaN), the remainder operation will fail with a divide by zero and it generates two normal numbers as inputs to provoke this behaviour. Unfortunately, when compiling the test at level -O3, the gcc 7.5.0 compiler decides to completely remove the check (which is undefined if `isnan(src2) == 0`) relying instead on the x86 SSE `ucomiss` instruction to detect NaN, so no floating point exception for divide-by-zero is generated and the code executes normally to produce the same result as the oracle. Now, the C99 >= operator should raise a floating-point exception when one of the operands is a NaN, but the `ucomiss` instruction will only do so if it is a *signaling* NaN. Unfortunately, our test suite does not contain a signaling NaN and CBMC is unaware of x86 semantics. The `clang` compiler does not have this "feature" and the test fails as expected.

The compile-time exploitation of undefined behaviour means that some bugs will slip through testing non-deterministically depending on the compile-time environment. Thus, while CBMC can flag undefined behaviour, an implementation may or may not manifest the undefined behaviour. A suggestion from a reviewer, which we plan to adopt, is to instrument the tests using undefined-behaviour sanitizers. This would increase the chances that the compiled version will detect the same undefined behaviours as CBMC.

Equivalence checking could be performed at the assembly-language level, which would detect this program as equivalent to the original except for signaling NaNs and hence produce a signaling NaN as input. Model checkers for x86 exist including those with support for SSE instructions [2], but we do not plan to explore them in this work.

D. RQ4: Syntactic Differences, Semantic Equivalence

Equivalence checking provides insights into the limitations of mutation generators. The *Same* column in Table II shows that, for some instructions, the majority of mutants that survive the first round of tests are those that are equivalent to the original program. Our limited survey of these programs shows this is because the effect of the mutations is masked either statically or dynamically. Some of these reasons are described below, in no particular order.

a) Dead Code: Many mutations like the one introduced below in `abs.f32` are dead code and will never execute.

```
tmp_dst = fabsf(src1);
dst = ((tmp_dst) < 0 ? kill(getpid(), 9)
    : (tmp_dst));
```

In the above code, the assignment is replaced by a ternary operator, but the condition can never be true because `tmp_dst` is the output of `fabsf`.

b) Semantic Equivalence: In contrast, mutations like `fabsf(-src1)` will execute, but the semantics of `fabsf` guarantee that the output will remain unchanged. Although the ternary operator replacement is hard to reason about syntactically, `fabsf` can be detected syntactically and the negation of its operands potentially skipped.

[1]The dual is when a mutant evades the equivalence checker but would be caught by a test. This indicates a soundness bug in the equivalence checker. This can be checked by asking for a proof of equivalence for each entry in the *Same* column of Table II and verifying the proof. This is a process that can be automated but isn't in our work.

c) Near Semantic Equivalence: Some mutations are nearly equivalent to the original code, so there's very little chance they'll change the behaviour of the program. Consider this mutated code from the semantics of `set.eq.u32.f32` described previously:

```
pred2 = (!(isnan(src1) << isnan(src2)))
    && (src1 >= src2);
```

The mutation has the `||` replaced with `<<`. However, for boolean 0–1 inputs and outputs, `<<` agrees with `||` 75% of the time, disagreeing only for `0 << 1`. The original program would store 0 in `pred2` for this case, and that's what happens here as well, masking the effect of the mutation. This suggests that semantic equivalence should be thought of as a continuum rather than a binary value to drive the search for mutations with a higher probability of changing behaviour.

E. RQ5: Pipeline Costs

The *Time* column in Table II contains the time it takes for mutation generation and mutation testing per instruction. The testing of each mutant runs in parallel. The time includes compilation, running, and checking of the outputs. Clearly, mutation generation and testing is not very expensive for these small programs.

Equivalence checking (Table III) is more expensive. Even running in parallel, the throughput is only 1 mutant every few seconds. The overall times are still relatively small, though. The most expensive checks take just a little more than a minute of wallclock time. To scale to larger instruction sets, however, the technique described at the end of Section V-B will probably need to be used.

F. RQ6: Applicability

The instructions chosen for our preliminary tests span arithmetic, logic, and elementary math on floating point inputs. We plan to extend our pipeline to the other 4,085 instructions that span types from 8-bit integers to 64-bit integers and 64-bit floating point numbers as well. Nothing in our pipeline would restrict its applicability, however we suspect reasoning about bit-level instructions might be expensive.

G. RQ7: Evaluating Alternatives

We would like to evaluate the effectiveness of the generated test suite on mutations generated by Mull [10]. This could be coupled to use LLBMC [17], a bounded model checker for LLVM IR to perform equivalence checks on Mull output. LLBMC does not support floating-point instructions and we were unable to obtain it from its website. Alternatively, we can use SeaHorn [18] to check equivalence, though the integration with Mull will need to be worked out.

H. RQ8: Test Suite Generation from Scratch

Table IV shows the time taken to produce a test suite for each program from scratch. In this case, *all* mutants are sent to the equivalence checker for verification. Predictably, the time now taken is much higher, however there is no particular pattern that is evident. In contrast, stratified sampling takes milliseconds to produce inputs that can be reused across programs.

TABLE IV. EQUIVALENCE CHECKER TEST GENERATION FROM SCRATCH

Instruction	Generated	Unique	Time (s)
abs.f32	67	16	123.40 ± 0.31
add.rm.ftz.sat.f32	165	93	294.08 ± 0.86
add.rn.f32	24	14	48.18 ± 0.30
add.sat.f32	127	57	222.40 ± 1.09
set.eq.ftz.s32.f32	218	83	358.23 ± 2.16
set.ge.f32.f32	232	93	394.51 ± 1.31
set.gt.s32.f32	128	61	269.86 ± 0.56
set.gt.u32.f32	128	54	276.26 ± 0.74
setp.ge.f32	260	83	443.78 ± 1.84
sqrt.rm.f32	245	60	425.83 ± 1.64
sub.rn.ftz.sat.f32	165	91	291.80 ± 1.82
sub.rz.ftz.sat.f32	165	92	291.29 ± 1.74

The total number of inputs per instruction is solely the *Unique* number of inputs. As expected, these inputs kill the same number of mutants as the sum of *Kill #1* and *Kill #2* from Table II. However, in nearly every case, the number of inputs is much greater than those listed in Table III. Only `add.rn.f32` requires significantly fewer inputs to kill all mutants than stratified sampling. Other instructions that have fewer inputs are `add.sat.f32`, `set.gt.s32.f32` and `set.gt.u32.f32`.

Although the costs for equivalence checking could be lowered by some form of deduplication before performing the equivalence check, we suspect augmenting a good test suite is more cost effective than generating inputs from scratch.

Apart from cost, this method of test suite generation is sensitive to a particular mutation engine. This sensitivity can be evaluated by using the generated test suite on mutations generated by an alternative mutation engine.

I. RQ9: Comparison to Fuzzing

As of writing, our current infrastructure cannot substitute a fuzzer for an equivalence checker. However, to evaluate feasibility, we instrumented the example in Listing 2 and ran it using libFuzzer (clang 10 and 12). CBMC takes around 2s to detect a counter example. However, libFuzzer is unable to generate inputs that kill the mutant even when given more time. We tried input generation strategies that ranged from 1.1M executions/s (default mutator) to 300K exec/s using a prototype stratified-sampling-based mutator that we continue to refine. Nevertheless, we hope to confer with conference attendees to obtain an experiment setup that will generate a useful comparison for the final revision. As an aside, before this experiment, we did not appreciate the simplicity of the interface offered by equivalence checkers.

VI. RELATED WORK

Hierons et al. [19] surveys a number of techniques that use formal specifications in the service of testing. Using formal instruction set semantics to generate test cases is a form of *model-based test generation*. As semantics for instructions are usually embedded in instruction set simulators, past work on testing these simulators [20, 21, 22, 23, 24] highlight the main approaches used.

Instructions for the x86 instruction set are encoded as a constraint-satisfaction problem to generate test inputs for each path [24]. The work eschews SAT solvers because CSP problems are easier to encode. DeMilli and Offutt [25] explores how mutations can be associated with constraints that can then be solved to generate test inputs.

Martignoni et al. [23] use randomized testing by generating a stream of random bytes and testing the sequence on an actual physical CPU to identify both instructions and inputs randomly. Their method generates inputs randomly but is essentially a form of differential testing.

Specifications for Intel CPUs are mined from their manuals in [22] and used to construct test abstractions. These are instantiated with "boundary" values based on the intuition that boundary values lead to most errors.

Wagstaff et al. [20] instrument the executable specification of the ISA used in the instruction set simulator to obtain path coverage. This information is then used to construct path constraints that are solved using CVC4 to obtain test inputs that exercise rarely-used paths.

Coverage-guided fuzzing is used to construct test inputs in [21] where mutation is used to increase code coverage in an instruction set simulator. In constrast to these works, we mutate a stand-alone semantics which is not embedded in a simulator. We mutate the semantics to deliberately introduce bugs and use equivalence checking to surface inputs that trigger those bugs. Coverage-based techniques would complement our method.

In a manner similar to our goal of using mutation testing to add more tests, SpecTest [26] monitors the executable model of a language extracted from its semantics to obtain coverage of small step semantics that are used by test programs. These test programs are then mutated to increase coverage of language semantics by injecting program constructs that correspond to unexercised semantics. Their mutation technique could be used in our work to develop a richer set of bugs.

Mutation-based testing tools [9, 10] have traditionally checked if a test suite is adequate to detect randomly inserted bugs. It is usually outside the scope of mutation testing tools to generate test inputs. But some methods have generated tests by observing executions for state differences of (Java) objects and then constructing assertions to detect these differences on this state [27]. Values are constructed randomly whereas we rely on counter-example generation using a model checker.

Zhang et al. [28] drive dynamic symbolic execution to trigger mutation-detecting assertions that were introduced at the same time as the mutations, allowing inputs to be generated. In our method, the equivalence checker also operates symbolically but is highly decoupled from the mutation engine, and does not need to introduce additional assertions.

Another method to generate inputs that detect differences is through the use of differential symbolic execution [29]. In this work, Java programs are executed symbolically to generate (abstract) bounded execution summaries that are then checked for equivalence using CVC3. The simplicity of instruction set semantics allows us to avoid the use of summaries.

VII. Conclusion

We have presented an automatic test generation pipeline for instruction set semantics. The pipeline uses mutation testing to introduce bugs and equivalence checking to construct inputs that trigger those bugs. The method can be used to augment both an existing test suite and construct a test suite. However, our preliminary evaluation on 12 PTX instructions finds that the pipeline is cost-effective only for augmentation. We also find mutation testing can introduce undefined behaviour leading to non-determinism that can complicate testing of C implementations. We are currently limited to particular tools to maximize the number of instruction set semantics supported. Good support for floating point would allow us to use other equivalence checkers and mutation testing tools.

Coupling equivalence checking to a mutation testing framework provides a promising method to improve the quality of the mutation generation. As we observe in our evaluation, the ability to detect semantically equivalent mutants also provides an objective metric for mutation tools. Our notion of near semantic equivalence can be used to characterize the quality of individual mutations.

References

[1] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, Jan. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3290384

[2] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A Complete Formal Semantics of x86-64 User-level Instruction Set Architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019, pp. 1133–1148. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314601

[3] D. Carrington and P. Stocks, "A tale of two paradigms: Formal methods and software testing," in *Z User Workshop, Cambridge 1994*, ser. Workshops in Computing, J. P. Bowen and J. A. Hall, Eds. London: Springer, 1994, pp. 51–68.

[4] N. Corporation, *PTX: Parallel Thread Execution ISA*, 7th ed., 2021.

[5] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running Tests against Hardware," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, P. A. Abdulla and K. R. M. Leino, Eds. Springer Berlin Heidelberg, 2011, pp. 41–44.

[6] D. Kroening and M. Tautschnig, "Cbmc – c bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391.

[7] A. Armstrong, T. Bauereiss, B. Campbell, K. E. Gray, R. Norton-Wright, C. Pulte, S. Flur, and P. Sewell, "The Sail instruction-set semantics specification

language," p. 26, 2021. [Online]. Available: https://raw.githubusercontent.com/rems-project/sail/sail2/manual.pdf

[8] T. F. Serbanuta, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Rosu, "The K Primer (version 3.3)," *Electronic Notes in Theoretical Computer Science*, vol. 304, pp. 57–80, Jun. 2014. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1571066114000395

[9] D. L. Phan, Y. Kim, and M. Kim, "Music: Mutation analysis tool with high configurability and extensibility," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018, pp. 40–46.

[10] A. Denisov and S. Pankevich, "Mull it over: Mutation testing based on LLVM," *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr 2018. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2018.00024

[11] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*. Los Angeles, California: ACM Press, 1977, pp. 238–252. [Online]. Available: http://portal.acm.org/citation.cfm?doid=512950.512973

[12] "The CPROVER Manual: Floating point." [Online]. Available: http://www.cprover.org/cprover-manual/modeling/floating-point/

[13] M. Brain, F. Schanda, and Y. Sun, "Building Better Bit-Blasting for Floating-Point Problems," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 79–98.

[14] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *OSDI*, p. 16, 2008.

[15] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing. – LLVM 15.0.0git documentation." [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[16] M. Brain, "sqrtf appears to be non-deterministic and throwing spurious verification failures - Issue #6563 - diffblue/cbmc." [Online]. Available: https://github.com/diffblue/cbmc/issues/6563

[17] F. Merz, S. Falke, and C. Sinz, "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR," in *Verified Software: Theories, Tools, Experiments*, R. Joshi, P. Müller, and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7152, pp. 146–161, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-27705-4_12

[18] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn Verification Framework," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, vol. 9206, pp. 343–361, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-21690-4_20

[19] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Computing Surveys*, vol. 41, no. 2, pp. 1–76, Feb. 2009. [Online]. Available: https://dl.acm.org/doi/10.1145/1459352.1459354

[20] H. Wagstaff, T. Spink, and B. Franke, "Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '14. New York, NY, USA: Association for Computing Machinery, Oct. 2014, pp. 1–10. [Online]. Available: https://doi.org/10.1145/2656106.2656113

[21] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying Instruction Set Simulators using Coverage-guided Fuzzing*," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2019, pp. 360–365.

[22] W. Ma, A. Forin, and J.-C. Liu, "Rapid prototyping and compact testing of CPU emulators," in *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, Jun. 2010, pp. 1–7.

[23] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing CPU emulators," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ser. ISSTA '09. New York, NY, USA: Association for Computing Machinery, Jul. 2009, pp. 261–272. [Online]. Available: https://doi.org/10.1145/1572272.1572303

[24] S. V. Kodakara, D. A. Mathaikutty, A. Dingankar, S. Shukla, and D. Lilja, "Model Based Test Generation for Microprocessor Architecture Validation," in *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, Jan. 2007, pp. 465–472.

[25] R. DeMilli and A. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[26] R. Schumi and J. Sun, "SpecTest: Specification-Based Compiler Testing," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, E. Guerra and M. Stoelinga, Eds. Cham: Springer International Publishing, 2021, pp. 269–291.

[27] G. Fraser and A. Zeller, "Mutation-Driven Generation of Unit Tests and Oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, Mar. 2012.

[28] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, "Test generation via Dynamic Symbolic Execution for mutation testing," in *2010 IEEE International Conference on Software Maintenance*. Timisoara, Romania: IEEE, Sep. 2010, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/5609672/

[29] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: Association for Computing Machinery, Nov. 2008, pp. 226–237. [Online]. Available: https://doi.org/10.1145/1453101.1453131