Almost Correct Invariants: Synthesizing Inductive Invariants by Fuzzing Proofs

Sumit Lahiri
sumitl@cse.iitk.ac.in
Indian Institute Of Technology Kanpur
India

Subhajit Roy
subhajit@cse.iitk.ac.in
Indian Institute Of Technology Kanpur
India

ABSTRACT

Real-life programs contain multiple operations whose semantics are unavailable to verification engines, like third-party library calls, inline assembly and SIMD instructions, special compiler-provided primitives, and queries to uninterpretable machine learning models. Even with the exceptional success story of program verification, synthesis of inductive invariants for such "open" programs has remained a challenge. Currently, this problem is handled by manually "closing" the program—by providing hand-written *stubs* that attempt to capture the behavior of the unmodelled operations; writing stubs is not only difficult and tedious, but the stubs are often incorrect—raising serious questions on the whole endeavor.

In this work, we propose *Almost Correct Invariants* as an automated strategy for synthesizing inductive invariants for such "open" programs. We adopt an active learning strategy where a data-driven learner proposes candidate invariants. In deviation from prior work that attempt to *verify* invariants, we attempt to *falsify* the invariants: we reduce the falsification problem to a set of reachability checks on non-deterministic programs; we ride on the success of modern fuzzers to answer these reachability queries. Our tool, Achar, automatically synthesizes inductive invariants that are sufficient to prove the correctness of the target programs. We compare Achar with a state-of-the-art invariant synthesis tool that employs theorem proving on formulae built over the program source. Though Achar is without strong soundness guarantees, our experiments show that even when we provide almost no access to the program source, Achar outperforms the state-of-the-art invariant generator that has complete access to the source. We also evaluate Achar on programs that current invariant synthesis engines cannot handle—programs that invoke external library calls, inline assembly, and queries to convolution neural networks; Achar successfully infers the necessary inductive invariants within a reasonable time.

CCS CONCEPTS

• **Software and its engineering** → **Software verification**; **Software testing and debugging**; • **Theory of computation** → *Logic and verification*; **Invariants**; • **Security and privacy** → *Logic and verification*.

KEYWORDS

inductive invariant synthesis, fuzzing, verification, testing

ACM Reference Format:
Sumit Lahiri and Subhajit Roy. 2022. Almost Correct Invariants: Synthesizing Inductive Invariants by Fuzzing Proofs. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534381

1 INTRODUCTION

Program verification has come a long way, with a plethora of algorithms, improved encodings, and sophisticated heuristics that have allowed us to assert correctness on complicated and large software systems. A popular strategy to verification is via synthesis of *inductive invariants* that allows for an inductive proof of correctness (proving that the program meets its specification on all inputs). However, synthesis of inductive invariants requires the program to be *closed*—that is, the source code of all procedures is available so that one is able to translate the semantics of the program into a mathematical representation, popularly in some fragment of first-order logic, for formal reasoning techniques to be applied. This presents a difficulty to the verification of *real* (open) programs—those containing external function calls, inline assembly, system calls, vector instructions, and consult complex machine learning models. Currently, handling such programs requires the programmer to manually provide *stubs*—simplified implementations that approximate the behavior of the constructs required to "close" the program. Writing such stubs is challenging: (1) it requires manual effort, is laborious and error-prone, (2) being simplified models of the behavior of the real operation, they often are neither sound nor complete, (3) sometimes, especially for external libraries, the verification engineer is not conversant of the complete semantics of the operation being modeled, the stub then being a reflection of the engineer's understanding rather than the true semantics[1].

In this work, we propose *Almost Correct Invariants*—a new methodology that performs a best-effort attempt at inductive invariant synthesis for such "open" programs. We assume the presence of such real-world artifacts like external function calls, inline assembly code, and system calls as *opaque* operations. Access to the semantics of such *opaque* operations is *only* possible by executing the operation as an input-output oracle, i.e. one can only capture the output by *executing* the operation on the desired input(s).

Our primary point of deviation from prior techniques is the following: instead of attempting to *verify* the inductive invariants, we attempt to *falsify* candidate invariants; any invariant that cannot be

[1]Private correspondence of the author with technical managers in a leading organization focused on static analysis and verification.

```
Precondition: (b0 >= 0)
int multiply (s⃗: [a0, b0, supported]) {
  // loop-head
  int a = a0, b = b0, r = 0; unsigned shift = 0;
  while (b != 0) { // loop guard
    if (supported) {  // loop-body
      shift = __builtin_ctz(b) ;
    } else {
      shift = 0; }
    if (shift) {
      r += a << shift;
      b -= 1 << shift;
    } else {
      r += a; b -= 1;
    }}
  return r; // loop tail
}
Postcondition: (r == a0 * b0)
```

Listing 1: Program to *multiply* two numbers.

falsified is assumed *almost correct*. We use a fuzzing engine to drive our falsification attempts. Though such invariants are not guaranteed to hold on all program paths, they are still representative of the program behavior as they hold on *almost all* (if not all) program paths. Such "almost correct" artifacts have been been found useful in many domains like reactive synthesis of controllers [24, 25], almost correct specifications [9] etc.

There has also been interest in identifying likely invariants from dynamic program traces [27, 84]; however, such techniques are incapable of inferring inductive invariants against a specification as they lack a verification or falsification mechanism (they generally infer invariants over a user-provided set of tests). *Ours is the first attempt at applying falsification to infer almost correct inductive invariants against given program specifications on program using opaque operations.*

We motivate our use-case with a program (Listing 1) that computes the *product* of two numbers by repeated addition. Such "software emulations" are common in small devices that may lack multiplication hardware. Some compilers provide special primitives that can be used by a program to build optimized implementations. This program uses the primitive `__builtin_ctz()` that is provided by GCC; this primitive returns the count of trailing zeros in the binary representation of the number. Our program supports two modes:

Slow version. If a compiler is used that does not provide the primitive (indicated by the flag `supported = false`), we repeatedly add the multiplicand `a` to the result `r`, a total of `b` times.

Fast version. If we use a compiler that makes this primitive available (indicated by `supported = true`), the computation is optimized: it exploits the fact that if `b` is a power of 2, then the result of `a*b` can be found by left-shifting `a` by $log(b)$; note that $log(b)$ is nothing but the count of trailing zeros in `b` (which is provided by `__builtin_ctz(b)`. If `b` is not a power of 2, we may use its binary expansion as a sum of powers of two. For example, $35 = 2^4 + 2^2 + 2^0$; so the optimization can be applied to each of the summands and the results added.

E.g., with `a = 11596` and `b = 26357`, while the slow version takes **26357** iterations, the *optimized* version takes only **10** iterations.

```
((b0 >= 0) && (a0 == a) && (r == (b0 - b) * a))
```

Listing 2: Invariant for Listing 1 generated by Achar.

The specification asserts that if `b0 >= 0` (Precondition), the program (if it terminates) should return `a0 * b0` in the variable `r` (Postcondition). As the semantics of `__builtin_ctz()` may not be available to the verifier, the correctness of this program cannot be established by current verifiers. Our algorithm is able to establish the correctness of the above program (with only closed-box access to `__builtin_ctz()`) by emitting an inductive invariant (shown in Listing 2) as a witness to the proof.

We build our synthesis engine for inductive invariants using active learning: while a *learner* proposes candidate invariants, a *teacher*, employing the above scheme, searches for counterexamples to the validity of the proposed invariant. The teacher encodes the axiomatic semantics of the program into a non-deterministic program, and uses a reachability oracle (implemented using a fuzzer) to search for violations to the proof of correctness. Failure to uncover a violation provides a *best-effort proof of correctness*.

We build our ideas into our tool, Achar, and use it to synthesize best-effort inductive invariants for programs employing *opaque* operations. On a set of **133** programs, Achar exhibits better performance than a state-of-the-art tool (Code2Inv [73, 74])—successfully verifying **8** additional programs—even when the teacher was provided only closed-box access to the program. The generated invariants often reveal interesting characteristics of the participating *opaque* operations. We also evaluate Achar on a set of programs that contain *opaque* operations (inline assembly, external library calls, and can invoke neural networks) and hence, are beyond the capabilities of current invariant generators.

Inductive invariants not only prove the correctness of a program, they also expose interesting properties of the program, including that of the *opaque* operations. For example, the invariant in Listing 2 states that this program maintains `r` as a multiple of `a0`; when `b` reduces to `0`, `r` reduces to nothing but `a0*b0`, thereby proving the correctness of this program. This makes *Almost Correct Invariants* also an interesting tool for program understanding when the complete source code is not available. Such applications have been also been suggested by prior literature [32, 33, 54]. Motivated by this goal, some invariant generation engines, like LoopInvGen [57], attempt to synthesize invariants on sampled data without any access to the source code; however, they are incapable of driving their active learning loop through *opaque* operations due to the use of a theorem prover for invariant validation. The use of such learners within our *Almost Correct Invariants* framework provides a synergistic combination where both the learner and the teacher can operate over *opaque* components. *To the best of our knowledge, Achar is the first end-to-end closed box synthesizer for inductive invariants.*

The following are the primary contributions of our work:

- We propose *Almost Correct Invariants*, a methodology to synthesize inductive invariants for "open" programs that contains *opaque* operations (§2, §3).
- We design an algorithm for *falsifying* invariants that contain *opaque* operations (§3). Including *opaque* operations within the

$$
\begin{aligned}
\langle P\rangle &::= \langle S\rangle;\ \textbf{while}\ \langle C\rangle\ \textbf{do}\ \langle S\rangle;\\
\langle S\rangle &::= \textbf{skip} \mid \langle var\rangle := \langle a\rangle \mid \langle S\rangle;\langle S\rangle \mid \textbf{if}\ \langle C\rangle\ \textbf{then}\ \langle S\rangle\ \textbf{else}\ \langle S\rangle\\
&\quad\mid\ \langle var\rangle ::= \textbf{opaque}(\langle var\rangle,\ldots,\langle var\rangle)\\
\langle a\rangle &::= \langle var\rangle \mid \langle num\rangle \mid \langle a_1\rangle\langle op_{\mathrm{a}}\rangle\langle a_2\rangle\\
\langle op_{\mathrm{a}}\rangle &::= + \mid - \mid * \mid /\\
\langle C\rangle &::= \textbf{true} \mid \textbf{false} \mid \textbf{not}\ \langle C\rangle \mid \langle C\rangle\langle op_{\mathrm{b}}\rangle\langle C\rangle \mid \langle a\rangle\langle op_{\mathrm{r}}\rangle\langle a\rangle \mid \star\\
\langle op_{\mathrm{b}}\rangle &::= \texttt{and} \mid \texttt{or}\\
\langle op_{\mathrm{r}}\rangle &::= < \mid \geq \mid == \mid > \mid \leq
\end{aligned}
$$

Figure 1: Language Syntax

invariant allows us to synthesize "simple" invariants, hiding the complexity within the participating *opaque* operations.

- We demonstrate a methodology for hybrid verifiers that allows the use of SMT solvers and fuzzers in synergy for verification of challenging software (§3).
- We build an implementation, ACHAR, capable of synthesizing (best-effort) inductive invariants on open programs (§4).
- We defend our claims through an elaborate set of experiments, both on benchmarks used in prior literature and new examples adapted from open-source repositories (§5).

Though our technique uses a fuzzer, the reader should note the deviation of our work from the typical use of fuzzers: while fuzzers are generally employed as a testing strategy on programs to search for bugs, we employ fuzzers as a *falsification* engine for inductive proofs. We provide a detailed comparison of *Almost Correct Invariants* against both testing and verification algorithms in the final section.

2 OVERVIEW

For the purpose of illustration, we use the simplified programming language syntax shown in Figure 1. Our subject programs $\mathcal{P}$ typically contain a single `while` loop; we refer to the statements preceding the loop as *loop head* ($\mathcal{P}_{head}$), the statements constituting the loop as the *loop body* ($\mathcal{P}_{body}$) and the statements following the loop as the *loop tail* ($\mathcal{P}_{tail}$). Further, the condition guarding the exit from the loop is referred to as the *loop guard* ($\mathcal{P}_{guard}$). The statements comprising the loop head, body and tail can be assignments to expressions, skip statements or branching statements over conditions (C), including non-determinism ($\star$). Additionally, we also allow *opaque* operations (eg. external library calls) that only allow one to observe their input-output interactions. For ease of exposition, we limit our discussions to programs with a solitary *opaque* operation returning a scalar value; ACHAR allows multiple *opaque* operations that can return a response vector (assigned to a set of variables).

Figure 2 provides the axiomatic semantics of the language: all rules (except *opaque*) are quite straightforward and are based on Hoare Logic [39]. As an *opaque* operation only allow inspection of its input-output interaction, we define its behavior by its collecting semantics over all possible responses over the input state. We notate the (partial) correctness specification as a Hoare triple [39], $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$ —given that we invoke $\mathcal{P}$ in a state that satisfies φ_{pre}, if the program terminates, it can only exit in a state that satisfies φ_{post}. The assertions φ_{pre} and φ_{post} are expressed in some

$$\frac{}{\{\varphi\}\ \textbf{skip}\ \{\varphi\}}\ \textit{skip} \qquad \frac{}{\{\varphi[\mathrm{a/x}]\}\ x := a\ \{\varphi\}}\ \textit{asgn}$$

$$\frac{\{\varphi_1\}\ \mathrm{S}_1\ \{\varphi_2\} \qquad \{\varphi_2\}\ \mathrm{S}_2\ \{\varphi_3\}}{\{\varphi_1\}\ \mathrm{S}_1\ ;\ \mathrm{S}_2\ \{\varphi_3\}}\ \textit{seq}$$

$$\frac{\{\varphi_1 \wedge \mathrm{b}\}\ \mathrm{S}_1\ \{\varphi_2\} \qquad \{\varphi_1 \wedge \neg\mathrm{b}\}\ \mathrm{S}_2\ \{\varphi_2\}}{\{\varphi_1\}\ \textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2\ \{\varphi_2\}}\ \textit{branch}$$

$$\frac{\{\varphi \wedge \mathrm{b}\}\ \mathrm{S}\ \{\varphi\}}{\{\varphi\}\ \textbf{while}\ \mathrm{b}\ \textbf{do}\ \mathrm{S}\ \{\varphi \wedge \neg b\}}\ \textit{loop}$$

$$\boxed{\frac{\varphi_2 \equiv \{\vec{s'} \mid t = [\![\textbf{opaque}(\vec{s}[\vec{x}])]\!], \varphi_1 \models \vec{s}, \vec{s} \models \vec{x}, \vec{s'} = \vec{s}[t/y]\}}{\{\varphi_1\}\ \vec{y} = \textbf{opaque}(\vec{x})\ \{\varphi_2\}}\ \textit{opaque}}$$

$$\frac{\models (\varphi_1 \rightarrow \varphi_2) \qquad \{\varphi_2\}\ \mathrm{S}\ \{\varphi_3\} \qquad \models (\varphi_3 \rightarrow \varphi_4)}{\{\varphi_1\}\ \mathrm{S}\ \{\varphi_4\}}\ \textit{conseq}$$

Figure 2: Axiomatic Semantics

base logic, popularly in a fragment of the first-order logic. The loop rule in Hoare logic is of special interest: it requires inference of an *inductive loop invariant*, φ, that satisfies the *loop* rule to establish the correctness of the `while` loop w.r.t the provided specification.

Problem Statement. Given a specification $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$, where the program $\mathcal{P}$ may contain *opaque* operations, we attempt to synthesize an *inductive loop invariant* $\mathcal{I}$ that establishes the correctness proof $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$. The validity of $\mathcal{I}$ is contingent on the validity of the following assertions:

$$\text{(precondition check)}\quad \{\varphi_{pre}\}\ \mathcal{P}_{head}\ \{\mathcal{I}\} \tag{1}$$

$$\text{(loop-condition check)}\quad \{\mathcal{I} \wedge \mathcal{P}_{guard}\}\ \mathcal{P}_{body}\ \{\mathcal{I}\} \tag{2}$$

$$\text{(postcondition check)}\quad \{\mathcal{I} \wedge \neg\mathcal{P}_{guard}\}\ \mathcal{P}_{tail}\ \{\varphi_{post}\} \tag{3}$$

Our Methodology. We set up the invariant synthesis problem as an active learning problem: the learner (L) proposes potentially interesting expressions, from a user-provided grammar $\mathcal{G}$, as candidate invariants. The teacher verifies a candidate invariant against the correctness specification $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$:

- if the teacher is able to establish the validity of the proposed invariant (w.r.t. the three conditions shown above), it declares success, returning the invariant;
- if the teacher is able to find a violation to any of the three conditions specified above, it forwards a failure message back to the learner, with the *reason* of failure, in the form of *counterexamples* to one or more of the above conditions. In this case, the learner is supposed to learn from the failure to propose a "better" invariant in the next round.

Different algorithms has been used for learners: using formal methods [3, 30], linear algebraic techniques [16, 67] and machine learning [28, 32, 33, 51, 57, 73, 74]. However, the design of the teacher has remained quite the same across different learners: typically, the set of validity checks is encoded in some fragment of first-order logic and posed as a satisfiability check to a theorem prover. The result from the theorem prover, including the generated counterexample, is communicated back to the learner.

Algorithm 1: GenInv($\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$, $\mathcal{G}$)

```
1  $E \leftarrow \emptyset$;
2  while true do
3      $\mathcal{I} \leftarrow$ Learner($\mathcal{G}, E$);
4      $\vec{cex} \leftarrow$ Teacher( $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$ , $\mathcal{I}$);
5      if $\vec{cex} == \bot$ then
6          return $\mathcal{I}$;
7      else
8          $E \leftarrow E \cup \{\vec{cex}\}$
```

However, in the presence of *opaque* operations—as is the case with most real-life programs—the validity checks on the proposed invariants cannot be decided by a theorem prover. The popular way to sidestep this challenge is to manually construct approximate models (called *stubs*) for these *opaque* operations. As discussed (see §1), this is not a satisfying solution to the problem. Verifying programs with such *opaque* operations, often provided by third-party vendors, with sketchy or non-existent documentation, has remained a pain point for many organizations.

We tackle this problem by constructing a non-deterministic program $\widehat{\mathcal{P}}$ such that the validity checks on the Hoare triples translate to reachability queries to certain *goal* program points in $\widehat{\mathcal{P}}$; that is, a goal point can be reached only if at least one of the Hoare triples for the invariant conditions is not valid. Unreachability of all the goal states is, hence, a certificate of the validity of the candidate invariant. We leverage the recent advancement in fuzzing technology to build a reachability oracle to answer these queries.

In contrast to prior approaches that require engineering the program semantics into "efficient" encodings conducive to theorem provers, our translation is quite straightforward—involving a syntactic extraction of relevant regions of the program $\mathcal{P}$ in a (non-deterministic) template. However, due to use of testing technology, our implementation does not guarantee soundness of the validity checks, and thus, of the synthesized inductive invariant. Hence, it is a best-effort attempt to synthesize *Almost Correct Invariants*.

3 ALGORITHM

3.1 Primary Algorithm

Our invariant synthesis strategy (Algorithm 1) invokes the learner (at Line 3) with a user-specified context-free grammar ($\mathcal{G}$) and the current set of examples (E) to generate a candidate invariant ($\mathcal{I}$). The grammar ($\mathcal{G}$) controls the expressiveness of the invariant; the learner constructs the invariant as a string from $\mathcal{G}$. The teacher (at Line 4) verifies the proposal $\mathcal{I}$ against the provided specification $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$: if the teacher is able to produce a counterexample that witnesses the violation of any of the conditions (Eqns: 1,2,3), the counterexample is added to the set of examples and the process repeated. If no such violation is found (i.e. the invariant $\mathcal{I}$ is valid), then candidate invariant $\mathcal{I}$ is returned. Of course, both our learner must be capable of handling *opaque* operations.

Algorithm 2 shows the design of the teacher. The procedure Translate() (Algorithm 2, Line 1) translates the validity check of a candidate invariant $\mathcal{I}$ for $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$ (Eqs: 1,2,3) to a reachability problem on a non-deterministic program $\widehat{\mathcal{P}}$.

Algorithm 2: Teacher($\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$, $\mathcal{I}$)

```
1  $\widehat{\mathcal{P}} \leftarrow$ Translate( $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$ , $\mathcal{I}$);
2  $\vec{s} \leftarrow$ ReachOracle($\widehat{\mathcal{P}}$);
3  return $\vec{s}$;
```

The scheme of translation is shown in Algorithm 3: $\widehat{\mathcal{P}}$ is constructed using the program statements , the specification and the candidate invariant . $\widehat{\mathcal{P}}$ non-deterministically chooses to apply the precondition check (Eqn-1) at Line 6, or the others (Eqns-2, 3) at Line 15 and Line 19 respectively. The validity check for a Hoare triple $\{\varphi\}\ Q\ \{\psi\}$ is translated to a branch statement on φ that guards an execution of Q, and returns $\top$ if the resulting program state does not satisfy ψ—indicating that a violation of the Hoare triple is detected (or that the proof of validity with $\mathcal{I}$ is *falsified*). Note that the *opaque* operations in $\mathcal{P}$ are retained in $\widehat{\mathcal{P}}$.

If any execution $\widehat{\mathcal{P}}(\vec{s})$ (where $\vec{s}$ is the initial program state on which $\widehat{\mathcal{P}}$ is invoked) reaches any of the *goal* program points $L = \{L_1, L_2, L_3\}$ (that is, $\widehat{\mathcal{P}}$ returns $\top$), then $\mathcal{I}$ is not valid and $\vec{s}$ is a witness to the failure of the corresponding assertion (among Eqs-1,2,3). More specifically,

- Returning from L_1 (Line 6) indicates Eqn. 1 is violated;
- Returning from L_2 (Line 15) indicates Eqn. 2 is violated;
- Returning from L_3 (Line 19) indicates Eqn. 3 is violated;

If there does not exist any path in $\widehat{\mathcal{P}}(\vec{s})$, for any $\vec{s}$, to any of the above labels (i.e. $\widehat{\mathcal{P}}$ can only return $\bot$), the candidate invariant $\mathcal{I}$ can be taken as a valid witness to the correctness of $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$.

K-bounded exploration. Detecting violations of the invariant condition (Eqn 2) is generally challenging. We use a strategy, inspired by the notion of k-induction [44], to improve the chances of detecting a counterexample: we perform the test of this condition across multiple loop iterations (Algorithm 3, Lines 11 to 15), looping over $\mathcal{P}_{body}$ for a bounded number of times (controlled by the hyperparameter K).

Reachability Oracle. The teacher uses a *reachability oracle*, ReachOracle() (Algorithm 2, Line 2); `ReachOracle(`$\widehat{\mathcal{P}}$`)` is required to create input states such that the provided (non-deterministic) program $\widehat{\mathcal{P}}$ returns $\top$. We assume the following interface:

$$\text{ReachOracle}(\widehat{\mathcal{P}}) = \begin{cases} \vec{s}, & \text{if } \widehat{\mathcal{P}} \text{ returns } \top \text{ on witnessing input } \vec{s} \\ \bot, & \text{if none of the goals in L are reachable} \end{cases}$$

If ReachOracle() (Algorithm 1, Line 5) returns $\bot$, the current candidate invariant is returned, signalling *verification* for $\mathcal{P}$; otherwise, (Algorithm 1, Line 8), the counterexample is added to the set of examples (E) and the active learning loop is reiterated.

Theorem. If ReachOracle is sound and Algorithm 1 terminates, the program $\mathcal{P}$ satisfies the specification $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$, and $\mathcal{I}$ is a valid inductive invariant for the proof of correctness.

Example. For our motivating example (Listing 1), Algorithm 1 would iteratively call the Learner and the Teacher: while the Learner

Algorithm 3: Template for $\widehat{\mathcal{P}}(\vec{s})$ (K: hyperparamater)

```
1  set the program state to $\vec{s}$;
2  if $\star$ then
3      if $\varphi_{pre}$ then
4          $\mathcal{P}_{head}$ ;
5          if $\neg\mathcal{I}$ then
6              return $\top$ ; // goal $L_1$
7  else
8      if $\mathcal{I}$ then
9          if $\mathcal{P}_{guard}$ then
10             k := K; // K-bounded exploration
11             while $\mathcal{P}_{guard} \wedge k > 0$ do
12                 k := k - 1;
13                 $\mathcal{P}_{body}$ ;
14                 if $\neg\mathcal{I}$ then
15                     return $\top$ ; // goal $L_2$
16         else
17             $\mathcal{P}_{tail}$ ;
18             if $\neg\varphi_{post}$ then
19                 return $\top$ ; // goal $L_3$
20 return $\bot$;
```

```
1  void $\widehat{multiply}$ ($\vec{s}$: [a0, b0, supported0], K) {
2    a = a0, b = b0, supported = supported0;
3    if (*) {
4      shift = 0, r = 0; // loop head
5      if(!((b0 >= 0) && (a0 >= 0)
6        && (a == (b0 - b0) * r)))
7          return ⊤; /* goal $L_1$ */
8    } else {
9      if(((b0 >= 0) && (a0 >= 0)
10       && (a == (b0 - b0) * r))) {
11       if (b != 0) { // loop guard
12         k = K; // K-bounded exploration
13         while ((b != 0) && k > 0) {
14           // loop body
15           k = k - 1;
16           shift = __builtin_ctz(b) ;
17           if (shift) {
18             r += a << shift;
19             b -= 1 << shift;
20           } else {
21              r += a; b -= 1; }
22           if(!((b0 >= 0) && (a0 >= 0)
23             && (a == (b0 - b0) * r)))
24               return ⊤; /* goal $L_2$ */
25     }} else { // loop tail
26         if (!(r == a0 * b0)) // post-condition
27           return ⊤; /* goal $L_3$ */ }}}
28   return ⊥; }
```

Listing 3: $\widehat{\mathcal{P}}$ for Multiply Example

proposes an invariant (Line 3) based on the current set of accumulated examples E, the Teacher attempts to invalidate the invariant (Line 4). Say, at some point in this active learning loop, the Learner proposes the following invariant:

Algorithm 4: TeacherHybrid($\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$, $\mathcal{I}$)

```
1  $\mathcal{P}_{cs}, \widehat{\mathcal{P}}_{ro} \leftarrow$ Translate( $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$ , $\mathcal{I}$);
2  $\vec{u} \leftarrow$ TheoremProver($\mathcal{P}_{cs}$);
3  if $\vec{u} == \bot$ then
4      return ReachOracle($\widehat{\mathcal{P}}_{ro}$);
5  return $\vec{u}$;
```

`((b0 >= 0) && (a0 >= 0) && (a == (b0 - b0) * r))`

To invalidate this invariant, the Teacher uses Algorithm 2: at Line 1, it starts off by constructing a non-deterministic program in accordance to Algorithm 3. For our current example, to invalidate the above invariant, it constructs a non-deterministic program as shown in Listing 3. The reachability oracle is invoked with this (Listing 3) non-deterministic program at (Algorithm 2, Line 2); the reachability oracle can produce the following state that reaches the goal L_2 (Line 24 in Listing 3), thereby returning $\top$.

$\top$: `({a0:1, b0:1281, a:1, b:1281, r:0, shift:0})`

Correspondingly, the teacher returns the counterexample to the active learning loop (Algorithm 1, Line 4 and Algorithm 3, Line 15), and the loop is iterated. This process continues till a valid invariant is proposed by the Learner, like the one shown below:

`((b0 >= 0) && (a0 == a) && (r == (b0 - b) * a))`

For the above candidate, ReachOracle will not be able able to reach any of the goal states and will return $\bot$ (Listing 3, Line 28). Algorithm 1, then, returns with the above invariant from Line 6. Please note that algorithm only requires a closed-box access to $\mathcal{P}_{head}$, $\mathcal{P}_{body}$, $\mathcal{P}_{tail}$ and $\mathcal{P}_{guard}$.

3.2 Hybrid Verification via Path Partitioning

Algorithm 4 proposes a synergistic combination of a theorem prover and a reachability oracle for the teacher. In principle, this scheme partitions the paths in the program $\mathcal{P}$ into two programs: (1) $\widehat{\mathcal{P}}_{ro}$ accumulates paths that contain *opaque* operations, and the reachability oracle is used on this; (2) $\mathcal{P}_{cs}$ has the remaining paths, a theorem prover can be in this case. In more detail:

- $\mathcal{P}_{cs}$: We translate the program $\mathcal{P}$ to a program $\mathcal{P}_{cs}$, where we "block" all paths in $\mathcal{P}_{head}$, $\mathcal{P}_{body}$ and $\mathcal{P}_{tail}$ that contain calls to *opaque* operations. We do so by replacing all instances of the *opaque* operation by *assume(false)*. The construct *assume(ψ)* is a common modeling primitive available in verification condition generators `assume`(ψ) 'blocks' all executions of the program that do not satisfy the predicate ψ.

$$\mathcal{P}[assume(false)\ /\ y = opaque(\vec{x})] \rightsquigarrow \mathcal{P}_{cs}$$

- $\widehat{\mathcal{P}}_{ro}$: We compute the *backward slice* [20] on $\widehat{\mathcal{P}}$, with respect to the goal locations *only along paths where some opaque operation is executed*, to generate a non-deterministic program $\widehat{\mathcal{P}}_{ro}$. In other words, all statements that are not in the backward slice, i.e. all statements that are never executed on any path where the *opaque* operation occurs, are eliminated.

$$\widehat{\mathcal{P}}[\{s | s \in BackSlice(\{L_1, L_2, L_3\}) \wedge \text{path}(s) \text{ hits } opaque(.)\}] \rightsquigarrow \widehat{\mathcal{P}}_{ro}$$

Further, as the behavior of $\widehat{\mathcal{P}}_{ro}$ is consistent with that of $\widehat{\mathcal{P}}$ only on paths where at least one *opaque* operation is hit, the *reachability* indicator `reach_flag` captures the fact that if an

```
1  int multiply_cs (s⃗: [a0, b0, supported]) {
2    int a = a0, b = b0, r = 0;
3    while (b != 0) { // loop guard
4      if (supported) { // loop-body
5        assume(false);
6      } else {
7        shift = 0; }
8      if (shift) {
9        r += a << shift;
10       b -= 1 << shift;
11     } else {
12      r += a; b -= 1; }}
13   return r;
14 }
```

Listing 4: $\mathcal{P}_{cs}$ for multiply() program.

```
int multiply_cs (s⃗: [a0, b0, supported]) {
  int a = a0, b = b0, r = 0;
  while (b != 0) { r += a; b -= 1; }
  return r;
}
```

Listing 5: Optimized $\mathcal{P}_{cs}$ for multiply() program.

execution reaches an *opaque* operation; hence, the validity checks are only applied if `reach_flag = true`.

While we translate $\mathcal{P}_{cs}$ to a logical formula to be verified by a theorem prover, $\widehat{\mathcal{P}}_{ro}$ is solved via ReachOracle()—the invariant is deemed valid only when none of the two procedures is able to generate a counterexample. Also, a counterexample generated by any of the two procedures is a valid counterexample.

Example. We discuss our hybrid verification scheme via Listing 1.

$\mathcal{P}_{cs}$: Listing 4 shows the program partition to be verified via theorem proving. The loop head and tail do not contain any *opaque* operation, and so, the precondition and postcondition lemmas can be discharged by a theorem prover. However, the loop-condition check cannot be handled by a theorem prover due to the presence of the *opaque* operation at Line 7 in Listing 1. In this case, the call to `__builtin_ctz()` is replaced with `assume(false)` (Line 5); Listing 5 shows an optimized version of $\mathcal{P}_{cs}$.

$\widehat{\mathcal{P}}_{ro}$: Listing 6 shows the program partition for validity checking via ReachOracle(). As the loop head and tail do not contain *opaque* operations, they need not be included in $\widehat{\mathcal{P}}_{ro}$. The loop body (at Line 7 in Listing 1) contains an *opaque* operation; so we set the reachability indicator, `reach_flag` (Line 11, Listing 6). The check for the loop-condition lemma must now be guarded by `reach_flag` (Line 21, Listing 6).

On running this example with hybrid partitioning on Achar, the following invariant is generated and validated **both** by the theorem prover (for $\mathcal{P}_{cs}$) and ReachOracle() (for $\widehat{\mathcal{P}}_{ro}$).

```
((b0 >= 0) && (a0 == a) && (r == (b0 - b) * a))
```

3.3 Invariants Modulo *opaque* Operations

Listing 7 iterates through all natural numbers till the input integer `n`, and adds either `1` or `2` depending on if `n` is prime or not. The routine `isprime()` is an *opaque* operation, and is available only as an input-output oracle. If the grammar $\mathcal{G}$ over which

```
1  void multiply_ro (s⃗ : [a0, b0, supported0], K) {
2  a = a0; b = b0; r = 0, supported = supported0;
3    reach_flag = false ;
4  if (((b0 >= 0) && (a0 == 0) && (r == (b0 - b) * a))){
5    if (b != 0) { // loop guard
6      k = K; // K-bounded exploration
7      while ((i < n) && k > 0) {
8        k = k - 1;
9        if (supported) {
10         shift = __builtin_ctz(b) ;
11          reach_flag = true;
12       } else {
13         shift = 0; }
14       if (shift) {
15         r += a << shift;
16         b -= 1 << shift;
17       } else {
18         r += a;
19         b -= 1; }
20     if( reach_flag
21    && !((b0>=0) && (a==a0) && (r==(b0 - b) * a)))
22          return  ⊤ ; /* goal L2 */ }}
23 return ⊥; }}
```

Listing 6: $\widehat{\mathcal{P}}_{ro}$ for multiply() program.

the invariant is generated is allowed to include this *opaque* operations (`isprime()`), Achar discharges the following invariant:

```
(n>2 && i<=n) && ((isprime(n)&&out==i) || (!(isprime(n))&&out==2*i))
```

Note that the invariant now includes the primality checker `isprime()` (in Listing 7) **within** the invariant; as `isprime()` would involve complex computations, the use of this *opaque* operation within the invariant relieves the invariant generator from reasoning on this procedure. Verification often fails as the reasoning engine fails to handle an extremely small but complex fraction of the program. The all-or-nothing verification engines fail to capture cases where the program is *almost correct*, modulo the correctness of the *opaque* operations. Further, the generated "almost correct" invariants are excellent aids to program understanding.

This presents an interesting use-case of *almost correct invariants*: if a verification engine fails to reason on a certain complex set of instructions, these instructions can be abstracted into *opaque* operations, allowing the synthesis of relatively simple invariants over abstractions of these complex routines.

The reader may be aware of efficient primality checkers based on the famous conjecture by John Lewis Selfridge [83]. In fact, Selfridge, and now covered by the Number Theory Foundation, along with Samuel S. Wagstaff Jr. and Carl Bernard Pomerance, together offer a reward of $620 for providing a counterexample to the above conjecture. If the `isprime()` routine (in Listing 7) used such conjectures, it will inevitably fail regular verification engines. Checking correctness *modulo the correctness of such conjectures* can significantly advance software reliability.

4 IMPLEMENTATION

We, now, provide details on the implementation of the learner and the teacher, along with a discussion on a set of optimizations that

```
Precondition: (n > 2)
int incr_prime_n(int n) {
   int i = 0, out = 0;
   while (i < n) {
      i = i + 1;
      // Using Selfridge conjecture
      if ( isprime(n) ) {
        out = out + 1;
      } else {
        out = out + 2; }}
    return out;
}
Postcondition:
  ((isprime(n) && (out == n))
    || (!(isprime(n)) && (out == 2 * n)))
```

Listing 7: incr_prime_n() Example

```
S ::= S1 | S2
S1 ::= ( C1 ) | ( C1 "||" C1 ) | ( C1 "||" C1 "||" C1 )
C1 ::= p | ( p && p )
S2 ::= ( C2 ) | ( C2 && C2 ) | ( C2 && C2 && C2 )
C2 ::= p | ( p "||" p )
p ::= var cmp expr
expr ::= opaque ( var , var ) | expr1
expr1 ::= ( var op var ) | ( var op const )
  | ( const op var ) | ( const op const ) | terms
terms ::= var | const
cmp ::= < | > | == | <= | >=
op ::= + | - | *
```

Listing 8: Example Grammar $\mathcal{G}$ with *opaque* operation

were necessary to build an effective engine. We show a schematic of how Achar works with the Learner and Teacher in Figure 3.

Learner. Achar can use any learner that is capable of proposing candidate invariants over a user-provided grammar, learning over a set of counterexamples. Our current implementation, inspired by the recent success of deep learning for invariant inference, builds over the learner of the state-of-the-art engine Code2Inv [74].

Code2Inv encodes the abstract syntax tree (AST) of the program as a graph neural network (GNN), extended with additional edges to capture the control-flow and data-flow. Similar to prior proposals using GNN [4, 21, 26, 68], each node in the network is associated with an embedding vector, which are trained by message passing. Invariant learning is modeled as a reinforcement learning problem: the neural policy is used to generate a candidate invariant, and the policy is iteratively improved by trial-and-error, in accordance to a reward scheme. Generated candidate invariants are dispatched to a theorem prover for validity check, yielding a reward of 1 (valid) or 0 (invalid). As such binary rewards are too sparse to drive effective learning, Code2Inv uses a reward function based on the fraction of counterexamples (from invalid checks) satisfied.

To allow for the use of *opaque* operations within the invariant, we build support to include *opaque* operations within the grammar $\mathcal{G}$; Listing 8 shows an example grammar that allows the use of a *opaque* operation `opaque()` with two parameters.

Teacher. We use HongFuzz [40] to build our reachability oracle. We add assertion failure (`assert(0)`) statements at the goal locations, and challenge HongFuzz to reach these program points. Such a testing-based reachability oracle is inherently incomplete—ReachOracle() will never be able to infer that the goal locations are unreachable; however, if it returns with $\top$, it implies that the provided violation is indeed possible.

To combat the incompleteness of our ReachOracle() procedure, we run the module with a timeout—and *assume* unreachability to all the goal locations if the procedure is not able to detect a counterexample within the budgeted time. Understandably, this renders our falsification engine unsound, and hence, only capable of producing *almost correct invariants*.

For multiple or nested loops, the learner proposes invariants individually for each loop and the teacher searches for one inductive proof though all the invariants. Our algorithm is capable of handling such scenarios.

In summary, our algorithm essentially reduces verification of a program to testing its correctness proof. The program $\widehat{\mathcal{P}}$ can be dispatched to any testing engine of choice, out of the box. Also, reduction to program testing does not mean giving up on symbolic techniques. In fact, Achar also uses the hybrid fuzzer Colossus [58], that also uses symbolic techniques, as an alternative test backend. But, overall, we found that pure evolutionary search with modern fuzzers (like Hongfuzz) to be best performing for our use-case and, if engineered well, quite competitive to SMT engines.

Handling impure opaque operations. We only discuss pure *opaque* operations for ease of presentation. However, Achar can also handle impure functions, like ones that take pointer arguments and modify the heap (eg. `strcat()`, `strcpy()`, `stpncpy()`, `strchr()`, `strncat()`, `strncpy()`, `strpbrk()`, `strrchr()`, `strstr()`). In such cases, a wrapper function is used to capture the reachable-heap, pass it as an additional argument to the fuzzer, and patch the changed heap state (similar to closure-conversion in functional-programs). Functions (eg. system calls) that may modify the operating system state (eg. signal masks) are beyond our current implementation.

Optimizations and hyperparameters. Code2Inv runs all the validity checks (Equation 1,2,3) and communicates a tuple with a record of the status of *all* of these checks back to its teacher. For Code2Inv, this design works quite well as theorem provers are capable of returning both Sat and Unsat verdicts efficiently.

However, this design is impractical for Achar as it is not capable of deciding that a given goal is unreachable. Quite often, at least one of the goals is unreachable; in such cases, the fuzzer will need to spend a time equal to the timeout budget used before a verdict can be reached. So, we modify the algorithm such that ReachOracle() returns as soon as it finds *any* counterexample. Our experiments show that this optimization has a significant impact on the runtime performance of Achar. For example, in Listing 10, the goal at Line 18 is unreachable as $x < 0 \land x > y \land i \le y \land i > 0$ is unsatisfiable. So, though the fuzzer may be able to reach the other locations quickly, it would spend time till the timeout expires in an unfruitful attempt at hitting this infeasible target.

The timeout is an important hyperparameter: a large timeout increases our confidence in the invariant but makes the tool sluggish; a short timeout will render the tool too unsound to be useful.

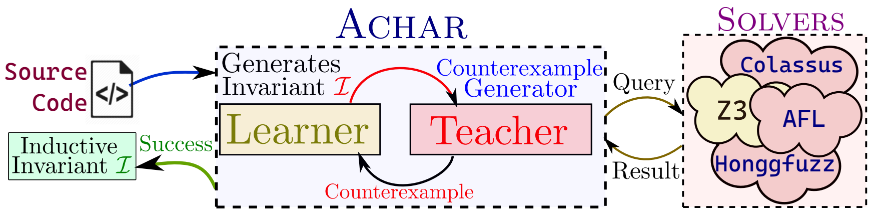

Figure 3: A schematic of the operation of Achar

```
Precondition: ((x >= 0) && (y >= 0)
  && (x >= y) && (i == 0))
int incr_till_y(int i, int x, int y) {
  while (*) { if (i < y)  i = i + 1; }}
Postcondition: (i<=y) ⟹ (i<=0) // fails
```

Listing 9: $\mathcal{P}$ for incr_till_y Example

```
void $\widehat{incr\_till\_y}$ ($\vec{s}$ : [i_0, x_0, y_0], K) {
  i = i_0, x = x_0, y = y_0;
  if (*) {
    if ((x >= 0) && (y >= 0)
      && (x >= y) && (i == 0)) {
      if(!(x < 0 && x > y)) return ⊤; }
  } else {
    if(x < 0 && x > y) {
      if (*) {
        k = K
        while ((*) && k > 0) {
          k = k - 1;
          if (i < y) i = i + 1;
          if(!(x < 0 && x > y)) return ⊤;}
      } else {
        if (i <= y) {
          if (!(i <= 0))
            return ⊤;/* Never Hit */}}}}
  return ⊥;}
```

Listing 10: $\widehat{\mathcal{P}}$ for incr_till_y returning atmost two models.

5 EXPERIMENTS

We conducted our experiments on a six-core Intel i7 machine with 24 GB primary memory running a Debian Linux with kernel version 5.11.0-25-generic. We use benchmarks used in Code2Inv [73, 74]: these are 133 programs from prior literature [33, 57] and SyGuS competition [77]. Each benchmark instances is a program and specification pair, $(\mathcal{P}, \psi)$, that challenges the invariant generator to construct an inductive invariant that **proves** that the program $\mathcal{P}$ satisfies the given specification ψ. Do note that Achar (like Code2Inv) never generates trivial invariants.

Code2Inv was shown to outperform (see [74]) three state-of-the-art solvers on these benchmarks: the stochastic solver (C2I [70]), a data-driven invariant generator with synthesized features (LoopInvGen [57]) and a data-driven invariant generator learning decision trees with manual features (ICE-DT [33]). Hence, we use Code2Inv as a baseline for comparisons (Code2Inv run in default setting).

We created a new benchmark suite by adapting programs from open-source repositories [1, 15, 17–19, 56, 62, 69, 76]. These benchmarks contain *opaque* operations like inline assembly, external library calls, and calls to convolutional neural networks. These programs *cannot be handled* by currently existing invariant generators. We call this suite as **Open-program benchmarks** (refer to §5.1.2). Our experiments answer to the following questions:

RQ1. Is Achar effective at synthesizing invariants?
RQ2. What is the runtime cost of Achar?
RQ3. How Achar performs when using a hybrid strategy?
RQ4. Are the hyperparameters and heuristics of Achar useful?

We found that Achar was quite effective at synthesizing invariants, even in the presence of *opaque* operations: on the Code2Inv benchmarks, it solves **8** more benchmarks than Code2Inv and it solves all instances in the open-program benchmarks in reasonable time. Further, our optimizations are crucial for the success of Achar as it solves fewer benchmarks without them.

Whenever not mentioned, we use the following hyperparameter settings: fuzzer timeout of 10s, K-bounded exploration factor of 64, and the fuzzer returning with a single counterexample (see §4). Unless otherwise state, the time reported for all experiments are over *converged* examples where the learner was able to halt with an invariant. As our learner is randomized, for both the baseline tool and for Achar, we run the benchmarks 3 times and collect the best performance (similar to [12]).

5.1 (RQ1) Synthesis Effectiveness

5.1.1 Code2Inv Benchmarks. For the first set of experiments on benchmarks collected from Code2Inv, we use the **most challenging setting** for Achar:

- *All the statements* in $\mathcal{P}_{head}$, $\mathcal{P}_{body}$, $\mathcal{P}_{tail}$ and $\mathcal{P}_{guard}$ in a program $\mathcal{P}$ are marked as *opaque* operations;
- We force Achar to generate invariants *through* the *opaque* operation (see § 3), thereby coercing Achar to discover properties about the *opaque* operations;
- We run our fuzzer in a *single-threaded mode* to keep our comparisons fair (though fuzzing is embarrassingly parallel).

Table 1 shows the number of correct instances, with Achar against Code2Inv. As the invariants generated by Achar may not be sound, we classify Achar results under three categories:

Correct. We perform a "verification check" on all *converged* instances; we use the "hidden" semantics of all the *opaque* operations and a theorem prover (z3) to check if the proposed invariant **is sufficient to prove** that the benchmark satisfies the specification; all instances that pass this check are marked "Correct" (we refer to such invariants as "valid" invariants). Note that the verification check requires the semantics of the *opaque* operations, so this check is only for evaluation purposes.

Wrong. The number of instances where the inductive loop of Achar converges to an incorrect invariant.

Colossus. The number of instances correctly solved when we use a stronger backend (HonggFuzz+Colossus); the backends are chained: when Hongfuzz **fails** to find a counterexample, we use Colossus [58] (timeout 10 sec) to find a counterexample.

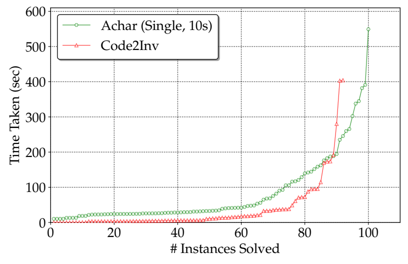

Figure 4: ACHAR vs. CODE2INV (Time)

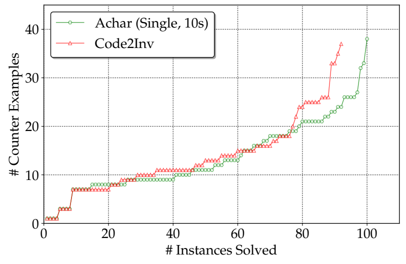

Figure 5: ACHAR vs. CODE2INV (Cex)

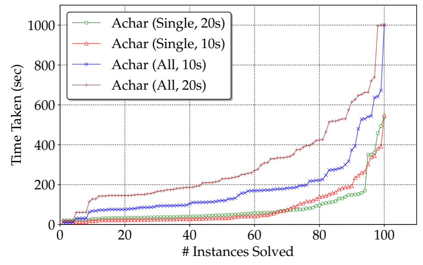

Figure 6: Solved vs. Time (All vs. One)

Table 1: Correct Instances: ACHAR vs. CODE2INV

Total	CODE2INV	ACHAR	
	Correct	Correct	Wrong
133	92	**100**	5

Table 1 shows that ACHAR solves 8 additional benchmarks with its primary backend. An expensive backend (chaining Hongfuzz with Colossus) solves an additional 2 benchmarks. Most of the times (100 out of 133) ACHAR indeed converges to a valid invariant but in some instances (5 out of 133) it generates an incorrect invariant.

Note that increasing the fuzzer timeout or using a more expensive testing engine increases our confidence in the invariant. We do a final verification run for the **5** examples on which ACHAR converged to the wrong invariant. We evaluate two settings: (1) the fuzzer using a 4 hour timeout, and (2) the more expensive HonggFuzz+Colossus backend. A higher fuzzer timeout indeed instills more confidence on the invariant as now ACHAR converges to a wrong invariant for only **1** case, i.e. it successfully falsifies the invariant in the other **4** cases (but fails to converge to a correct one). The HonggFuzz+Colossus backend converges to the correct invariant for **2** of these **5** instances and does not generate even a single incorrect invariant. This shows the value of more expensive backends at the cost of increased execution time. While the TEACHER module in CODE2INV, that uses calls to the z3 [22] solver, has complete access to the source code, the TEACHER in ACHAR is only provided *opaque* access to all the statements in the program. So, even in a significantly more constrained setting, ACHAR solves more instances than CODE2INV.We believe that ACHAR solves more examples than CODE2INV as a fuzzer generates more diverse counterexamples that aids the machine learning algorithm in the learner. SMT engines tend to produce "similar" counterexamples due to lemma learning.

ACHAR may not able to "solve" an example as:

The learner is not able to learn the invariant. The graph neural network based frontend is known to struggle for invariants that can only be represented compactly in the disjunctive normal form (DNF) (as discussed in [73]). Some other engines (like ICE-DT [33]) are better at generating such invariants. As ACHAR can employ any data-driven LEARNER, it could be interesting to integrate a portfolio of learners within it (in the future).

The teacher is not able to produce a counterexample. A given problem instance often yields to more fuzzing time-budget or a more powerful tester. As we see above, both increase in the fuzzer time-budget and the use of the Colossus engine improves our falsification attempts and/or increases the number of correctly solved instances. However, our experiments show that Colossus is about 1.5 to 4 times slower than HongFuzz.

```
Precondition: digit_data[i] is image for digit i/10.
int sum_till_n(int n,
  vector<torch::data::Example<>> digit_data[]){
  int index = 0;
  while (index < n) {
    sum +=  predict(digit_data[index]) ;
    index += 1;
}}
Postcondition: (index == n)  ⟹  (441 <= sum <= 459)
```

Listing 11: $\mathcal{P}$ for RQ3 NN `sum_till_n`

Please note that we did the above experiments to position ACHAR in the invariant generation landscape with existing tools. Our motivation to build ACHAR was to generate invariants for *open* programs that **cannot be handled** by CODE2INV (or any other current tool).

5.1.2 Open-program Benchmarks. We created a new open-programs benchmark suite (Table 3) that contains **15** programs adapted from open-source public repositories ([1, 15, 17–19, 56, 62, 69, 76]). In these programs, **none of the current invariant generators can generate invariants due to presence of *opaque* operations**. Of course, CODE2INV fails on all of them due to the presence of *opaque* operations (Table 3). ACHAR was able to solve these examples in reasonable time to generate the correct invariant expression.

Verifying programs invoking CNN. One of these benchmarks calls a convolutional neural network (CNN); though there has been work on verifying neural networks, computing inductive invariants on programs invoking neural networks still remains challenging.

Listing 11 sums a sequence of handwritten digits: the `predict()` function (Listing 12) calls a pre-trained CNN to recognize the digit. We run it with the dataset `digit_data` consisting of images of 100 handwritten digits organized in the following manner: the first 10 consecutive entries contain images of digits 0, next 10 entries contain contain images of digit 1 and so on; the same pattern is repeated over the whole suite. That is, the i^{th} element of the array is an image of the digit (i / 10). In this setting, the program must produce **450** as `sum`; however, as the CNN may not have 100% accuracy, we allow it to compute this sum within an error margin: $441 \le sum \le 459$. To employ ACHAR, we simply model the CNN

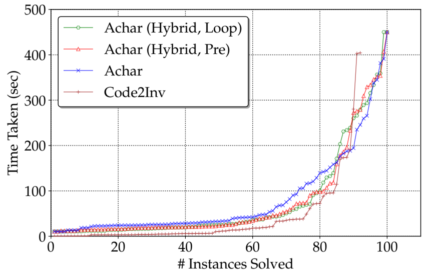

Figure 7: Achar Hybrid (Time)

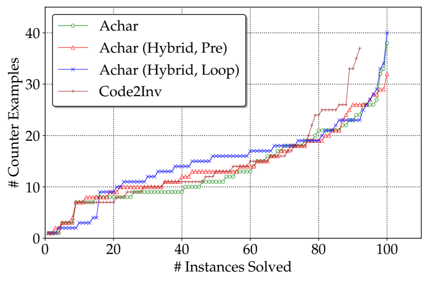

Figure 8: Achar Hybrid (Cex)

Table 2: Achar-Hybrid Summary

	Achar-Hybrid		
	Pre	Loop	Base
Time (s)	7570.4	7564.7	8587.0
Cex	1418	1503	1416
Correct	100	100	100

Table 3: Description for 15 Open-program Examples

#	Description	Type of Opaque Operations	Time (sec)	Cex
1	Sum of Squares	External library Call & Inline Assembly	57.17	7
2	Program using `isprime()`	External library Call	238.62	8
3	Greatest Common Divisor	External library Call & Unsupported operations by theorem prover (LIA Theory)	70.67	7
4	Fibonacci Series	External library Call	253.84	9
5	Exponentiation Modulo *n*	Unsupported operations by theorem prover (LIA Theory)	52.37	7
6	Fast Exponentiation	Unsupported operations by theorem prover (LIA Theory)	63.00	7
7	Integer SQRT	Unsupported operations by theorem prover (LIA Theory) in both pre-body and loop-body of the program.	209.68	7
8	Multiply Example	Compiler Primitive	81.39	7
9	Integer Cube Root	External library Call & Unsupported operations by theorem prover (LIA Theory)	102.20	8
10	Lock based program	External library Call & Inline Assembly	66.81	15
11	Algebric Expression (Cube)	External library Call & Unsupported operations by theorem prover (LIA Theory)	523.94	8
12	Integer Division	External library Call & Unsupported operations by theorem prover (LIA Theory)	67.63	7
13	Summing Handwritten Digits	Invoking a Convolutional Neural Network for predicting handwritten digits (CNN)	180.97	20
14	Fast Factorial	Compiler Primitive	499.05	10
15	Sum of Cubes	External library Call & Inline Assembly	140.53	8

invocation (i.e. the `predict()` function) as an *opaque* operation; Achar successfully emits the following correct invariant:

```
((index <= n) && ((sum >= ((5 * ((index/10) - 1) * (index / 10))
+ ((index/10) * (index - 10 * (index/10))) - 9)) &&
(sum <= ( (5 * ((index/10) - 1)
* (index/10)) +  ((index/10) * (index - 10 * (index/10))) + 9))))
```

This invariant shows that Achar was able to *discover* the pattern of the images in our dataset, reasoning **through** the convolutional neural network (CNN).

5.2 (RQ2) Runtime Cost

Figure 4 captures the essence of runtime costs. A point (`x`,`y`) indicates that the `x` instances took time `y` or less time to complete. The primary inference is that the runtime performance of Achar is comparable to Code2Inv, even though it is driven by a fuzzer that does not require access to the source code. Achar is only slightly slower than Code2Inv, even when employed in a single-threaded mode; the power of modern fuzzers and our engineering innovations allow Achar to match up with fast theorem provers. In Figure 5, the x-axis denotes number of solved instances and y-axis is number of counterexamples needed to solve an instance; eg. the point (80, 20) represents that 80 program instances were solved where 20 or less counterexamples were needed to generate a valid invariant. Even in this aspect, Achar, matches up with Code2Inv.

5.3 (RQ3) Hybrid Strategy

We evaluate the hybrid strategy of Achar, for each of the 133 benchmarks from Code2Inv, we create two instances by marking an operation in the loop head and the loop body (resp.) as *opaque* (the loop tail only contains the assertion condition in most examples).

Table 2 and Figure 7 show the comparision. In terms of time, as expected, the hybrid approach is faster than using only fuzzing. Though, it does not seem to impact the total number of solved instances, these modes are incomparable: the instances `ex45` and `ex65` get solved in the hybrid mode (both `pre` and `loop`), but not by the baseline Achar, perhaps due to the presence of complex branch conditions; however, there are examples (`ex2`,`ex122` for `pre`

```
int32_t predict(torch::data::Example<> image) {
    torch::jit::Module module
        = torch::jit::load(modelPTFile);
    torch::NoGradGuard no_grad;
    module.eval();

    double test_loss=0;
    int32_t correct=0, total=0, pred_num=0, tgt_num=0;
    auto data = image.data;
    auto targets = image.target;

    tgt_num = (targets).template item<int64_t>();
    std::vector<torch::jit::IValue> mnist_input;
    mnist_input.push_back(data);

    auto output = module.forward(mnist_input).toTensor();

    test_loss += torch::nll_loss(output, targets,
    /*weight=*/{}, torch::Reduction::Sum)
    .template item<float>();

    auto pred = output.argmax(1);
    pred_num = pred.template item<int64_t>();
    return pred_num;
}
```

Listing 12: Listing for `predict()` Function

and `ex66`,`ex120` for loop) that fail on Achar-hybrid but get solved by baseline Achar, perhaps due to better counterexample diversity. We believe that hybrid verification will significantly impact programs that involve complex branch conditions and hence, provides a means of tackling them.

5.4 (RQ4) Optimizations and Hyperparameters

We conduct these experiments on the Code2Inv benchmarks.

5.4.1 Fuzzer configuration. As discussed in §4, Achar packs a design modification that returns with the first counterexample model it finds. Figure 6 shows the plot for Achar with multiple configurations on the Code2Inv benchmarks: Achar (X, Y), X∈{All, Single}, represents the setting where Achar returns all violations for X=All (and only the first violation for X=Single) using a Y second timeout. Understandably for the "All" setting, the 20s timeout version is slower than the 10s version. The "Single" setting shows two interesting trends: (1) these variants are faster, (2) there is not much gap between the 10s and 20s variants. This is because, in all but the last invocation (in cases where the invariant converges) does the fuzzer expend time equal to the timeout. In all previous invocations, the fuzzer immediately returns with the very first counterexample. This differs from the "All" case: in all its queries, one of the lemmas may be valid and the fuzzer would need to timeout. This shows that our design modification is quite effective.

5.4.2 K-bounded exploration. We found that k-bounded exploration solves more instances: with this feature enabled `K=64`, Achar solves 100 instances while this drops to 97 instances when this feature is disabled. We found that `K=64` seems to be a stable setting where most of the examples are solved thus we choose it as the default setting for running Achar.

6 RELATED WORK

Synthesis techniques have been quite successful, both for inferring programs as well as program invariants, thanks to the phenomenal advancements in SAT and SMT solving. Program synthesis has been successfully used for multiple tasks: synthesis of complex bit manipulations [37, 43], heap manipulations [31, 60, 63, 80, 81], synthesizing bug repositories [65], skolem functions [34–36], regression-free repairs [8], differentially private programs [64], parsers [48, 75], fence synthesis in concurrent programs [79], automata [5], and even for the security of hardware curcuits [78].

For invariant synthesis, example-guided invariant inference engines [32, 33, 51, 54, 55, 57, 70–72, 86] have mostly dominated the invariant inference landscape. These techniques use the general scheme of active learning, with a *learner* proposing candidate invariants and a *teacher* verifying or refuting them. The community has seen designs of learners using abstractions [23, 29, 41, 42], stochastic search [70], machine learning [10, 47, 49, 50, 73, 74] and systematic enumeration [5]. There have also been techniques combining symbolic and enumerative search [37, 41, 42, 59, 85]. Though the above proposals have used different learners, theorem provers have been the common choice for the teacher.

Astorga et al. [6] learn *stateful preconditions* rather than inductive invariants; in fact, this work does not handle inductiveness. Miltner et al. [52] synthesizes *representation invariants* and uses a custom-designed enumerative tester to generate counterexamples to drive their synthesis. The design of their tester is crucial to the success of their scheme and it cannot operate with any testing engine. At the same time, they solve a very different problem.

Some dynamic invariant generation techniques [27, 84] attempt to learn invariants over a set of tests. They tend to use simple learners (often based on some templates) and do not include a verification/falsification engine, making them incapable of learning inductive invariants against specifications. Nguyen et al. [54] uses KLEE [11] to invalidate proposed invariants generated from traces. Similar to our work, the generated invariants are not guaranteed to be correct, but they were shown to be useful. However, this work is different than ours in many ways: firstly, their technique is *not* "goal-directed", i.e. they cannot be primed to generate invariants to prove correctness against certain asserts or post-conditions. Hence, they are again incapable of generating *inductive* invariants that serve as witnesses to proof of correctness against deep specifications. Secondly, due to the use of a open-box tester, they cannot handle *opaque* operations. Most importantly, in terms of the approach, they fuzz the program and not the proof (as Achar does); hence, they cannot drive inductive learning on the correctness theorem.

In the last few years, fuzzers have gained significant prominance in the software engineering, programming languages and security communities. The stupendous success of fuzzers like AFL [2] at finding deep bugs and security vulnerabilities has encouraged the design of new innovations in fuzzing [13, 38, 51, 53, 61, 82]. However, there is a potential of using more sophisticated coverage metrics [7, 14, 66] for use in such "almost" verification tasks.

7 DISCUSSION

Verification and testing are duals: while verification does not miss bugs, testing does not generate false positives. *Almost correct invariants* attempts to reduce the problem of verifying a program to *testing its proof of correctness*—it is fundamentally different than fuzz testing the original program.

Almost Correct Invariants versus Program Testing. A violation detected by fuzzing $\mathcal{P}$ indicates a bug in the program; a violation detected in $\widehat{\mathcal{P}}$ simply indicates that our falsification attempt on the candidate inductive invariant $\mathcal{I}$ has succeeded. Similarly, inability of the fuzzer to generate a counterexample on $\mathcal{P}$ does not provide any indication of the correctness proof but instills some confidence that $\mathcal{P}$ is *likely* to be correct; an inability of the fuzzer at generating a counterexample on $\widehat{\mathcal{P}}$ indicates that it is *likely* that the candidate inductive invariant $\mathcal{I}$ is valid for its proof of correctness.

The violation of an assertion on the $\mathcal{P}$ is converted to three reachability challenges in $\widehat{\mathcal{P}}$. So, structurally and semantically, $\mathcal{P}$ and $\widehat{\mathcal{P}}$ are quite different: while $\mathcal{P}$ can run long executions, and can potentially even get stuck in infinite loops, each execution path on the program $\widehat{\mathcal{P}}$ is bounded (controlled by the unroll factor hyperparameter K).

Almost Correct Invariants versus Program Verification. Albiet without strong guarantees, *almost correct invariants* provides a way forward towards automatically synthesizing inductive invariants for real-world code invoking multiple *opaque* components that cannot be handled by theorem provers. Note that the alternative, that of writing stubs manually, is also not without errors. There is a fundamental point of difference to fuzzing the non-deterministic program $\widehat{\mathcal{P}}$ versus fuzzing the original program $\mathcal{P}$—a violation detected by fuzzing $\mathcal{P}$ indicates a bug in the program. On the other hand, a violation detected by the fuzzer on $\widehat{\mathcal{P}}$ simply indicates that our verification attempt (with a candidate invariant $\mathcal{I}$ failed). Simiilarly, inability of the fuzzer to generate a counterexample on $\mathcal{P}$ does not provide any indication of the correctness proof; an inability of the fuzzer at generating a counterexample on $\widehat{\mathcal{P}}$ indicates that it is likely that the candidate inductive invariant $\mathcal{I}$ can be used to construct a proof of correctness.

Threats to validity. As fuzzers are sensitive to CPU throughput, alternate system may require a fresh search for optimal hyperparameters; the overall trend, however, should continue to hold.

8 DATA-AVAILABILITY STATEMENT

We provide the source code for Achar and data for the experiments on Zenodo[45]. The docker image of Achar is available on Docker Hub[46].

ACKNOWLEDGEMENTS

The first author is generously supported by TCS PhD Fellowship.

REFERENCES

[1] [n.d.]. Hacker's Delight, Second Edition [Book]. https://dl.acm.org/doi/book/10.5555/2462741.

[2] AFL. 2020. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[3] Aws Albarghouthi and Kenneth McMillan. 2013. Beautiful Interpolants. In *Computer Aided Verification.* 313–329. https://doi.org/10.1007/978-3-642-39799-8_22

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *In Proceeding of ICLR.* https://openreview.net/forum?id=BJOFETxR-

[5] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. Syntax-guided synthesis. In *2013 FMCAD.* 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[6] Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. 2019. Learning Stateful Preconditions modulo a Test Generator. In *Proceedings of the 40th ACM SIGPLAN Conference on PLDI (PLDI 2019).* ACM, 775–787. https://doi.org/10.1145/3314221.3314641

[7] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29).* 46–57.

[8] Rohan Bavishi, Awanish Pandey, and Subhajit Roy. 2016. To Be Precise: Regression Aware Debugging. In *Proceedings of the 2016 ACM SIGPLAN OOPSLA (OOPSLA 2016).* ACM. https://doi.org/10.1145/2983990.2984014

[9] Sam Blackshear and Shuvendu K. Lahiri. 2013. Almost-Correct Specifications: A Modular Semantic Framework for Assigning Confidence to Warnings. *SIGPLAN Not.* (jun 2013), 209–218. https://doi.org/10.1145/2499370.2462188

[10] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning shape analysis. In *SAS'17.* Springer, 66–87.

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

[12] Prantik Chatterjee, Abhijit Chatterjee, Jose Campos, Rui Abreu, and Subhajit Roy. 2020. Diagnosing Software Faults Using Multiverse Analysis. In *Proceedings of the Twenty-Ninth IJCAI.* IJCAI, 1629–1635. https://doi.org/10.24963/ijcai.2020/226 Main track.

[13] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE SP.* 711–725. https://doi.org/10.1109/SP.2018.00046

[14] R. Chouhan, S. Roy, and S. Baswana. 2013. Pertinent path profiling: Tracking interactions among relevant statements. In *Proceedings of the 2013 IEEE/ACM International Symposium on CGO'13.* 1–12.

[15] CodeProject. 2006. Using Inline Assembly in C/C++ - CodeProject. https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C

[16] Michael Colón, S. Sankaranarayanan, and H. B. Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *CAV'13.*

[17] Greatest common divisor. [n.d.]. Greatest common divisor - Rosetta Code. http://rosettacode.org/wiki/Greatest_common_divisor.

[18] CP-Algorithms-Factorial. 2014. Factorial modulo P - Competitive Programming Algorithms. https://cp-algorithms.com/algebra/factorial-modulo.html

[19] CP-Algorithms-GCD. 2014. Euclidean algorithm for computing the greatest common divisor - Competitive Programming Algorithms. https://cp-algorithms.com/algebra/euclid-algorithm.html

[20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* (Oct. 1991). https://doi.org/10.1145/115372.115320

[21] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative Embeddings of Latent Variable Models for Structured Data. In *In Proceedings of The 33rd ICML.* 2702–2711. http://proceedings.mlr.press/v48/daib16.html

[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS.* 337–340.

[23] Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. In *CAV.* 254–278.

[24] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. 2019. VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems. In *CAV'19.*

[25] Parasara Sridhar Duggirala, Sayan Mitra, Mahesh Viswanathan, and Matthew Potok. 2015. C2E2: A Verification Tool for Stateflow Models. In *Tools and Algorithms for the Construction and Analysis of Systems.*

[26] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P Adams. 2015. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *In Proceeding of NIPS'15.* https://proceedings.neurips.cc/paper/2015/file/f9be311e65d81a9ad8150a60844bb94c-Paper.pdf

[27] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69 (dec 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[28] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. 2018. Horn-ICE Learning for Synthesizing Invariants and Contracts. 2, OOPSLA (2018). https://doi.org/10.1145/3276501

[29] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference PLDI'17.* ACM, 15 pages. https://doi.org/10.1145/3062341.3062351

[30] Cormac Flanagan, K. Rustan, and M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME'21.* https://doi.org/10.1007/3-540-45251-6_29

[31] Anshul Garg and Subhajit Roy. 2015. Synthesizing Heap Manipulations via Integer Linear Programming. In *Static Analysis, SAS 2015. Proceedings.* https://doi.org/10.1007/978-3-662-48288-9_7

[32] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *CAV*. 69–87.

[33] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium POPL'16*. ACM, 14 pages. https://doi.org/10.1145/2837614.2837664

[34] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. 2020. Manthan: A Data-Driven Approach for Boolean Function Synthesis. In *CAV'20*. 611–633.

[35] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. 2021. Program Synthesis as Dependency Quantified Formula Modulo Theory. In *In Proceedings of IJCAI'21*. IJCAI, 1894–1900. https://doi.org/10.24963/ijcai.2021/261 Main Track.

[36] Priyanka Golia, Subhajit Roy, Friedrich Slivovsky, and Kuldeep S. Meel. 2021. Engineering an Efficient Boolean Functional Synthesis Engine. In *ICCAD*.

[37] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. *SIGPLAN Not.* 46, 6 (June 2011), 62–73. https://doi.org/10.1145/1993316.1993506

[38] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*. USENIX Association, 49–64.

[39] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

[40] Honggfuzz. 2020. Honggfuzz. https://honggfuzz.dev/.

[41] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015. Adaptive Concretization for Parallel Program Synthesis. In *CAV*. 377–394.

[42] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2017. An empirical study of adaptive concretization for parallel program synthesis. *FMSD* 50, 1 (01 Mar 2017). https://doi.org/10.1007/s10703-017-0269-8

[43] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *In Proceeedings of ICSE'10 (ICSE'10)*. ACM. https://doi.org/10.1145/1806799.1806833

[44] Temesghen Kahsai, Yeting Ge, and Cesare Tinelli. 2011. Instantiation-Based Invariant Discovery *(NFM'11)*. 15 pages.

[45] Sumit Lahiri and Subhajit Roy. 2022. Artifact for Almost Correct Invariants: Synthesizing Inductive Invariants by Fuzzing Proofs on Zenodo. 10.5281/zenodo.6534229. https://doi.org/10.5281/zenodo.6534229

[46] Sumit Lahiri and Subhajit Roy. 2022. Docker Image for Achar Tool. https://hub.docker.com/repository/docker/acharver1/achar.

[47] Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st ICML - Volume 32 (ICML'14)*. JMLR.org, II–1188–II–1196.

[48] Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. (June 2015), 565–574. https://doi.org/10.1145/2737924.2738002

[49] Omer Levy and Yoav Goldberg. 2014. Linguistic Regularities in Sparse and Explicit Word Representations. In *Proceedings of the 18th CCNLL*. Association for Computational Linguistics. https://doi.org/10.3115/v1/W14-1618

[50] Omer Levy and Yoav Goldberg. 2014. Neural Word Embedding as Implicit Matrix Factorization. In *Proceedings of the 27th NIPS - Volume 2 (NIPS'14)*. MIT Press.

[51] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on ESEC/FSE (ESEC/FSE 2017)*. ACM, 627–637. https://doi.org/10.1145/3106237.3106295

[52] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on PLDI (PLDI 2020)*. ACM, 1–15. https://doi.org/10.1145/3385412.3385967

[53] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. 2015. The BORG: Nanoprobing Binaries for Buffer Overreads. In *Proceedings of the 5th ACM Conference CODASPY (CODASPY '15)*. ACM, 87–97. https://doi.org/10.1145/2699026.2699098

[54] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-Guided Approach to Finding Numerical Invariants. In *Proceedings of the 2017 11th Joint Meeting on FSE (ESEC/FSE 2017)*. ACM, 605–615. https://doi.org/10.1145/3106237.3106281

[55] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *Proceedings of the 34th ICSE (ICSE '12)*. IEEE Press, 683–693.

[56] Elements of Programming Interviews. [n.d.]. Elements of Programming Interviews. https://elementsofprogramminginterviews.com/.

[57] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference PLDI'16*. ACM, 15 pages. https://doi.org/10.1145/2908080.2908099

[58] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. 2019. Deferred Concretization in Symbolic Execution via Fuzzing. In *Proceedings of the 28th ACM SIGSOFT ISSTA (ISSTA 2019)*. ACM. https://doi.org/10.1145/3293882.3330554

[59] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference PLDI'16*. ACM, 522–538. https://doi.org/10.1145/2908080.2908093

[60] Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural Synthesis of Provably-Correct Data-Structure Manipulations. In *OOPSLA*. 28 pages. https://doi.org/10.1145/3133889

[61] Sanjay Rawat, Vivek Jain, Ashish Kumar, L. Cojocar, Cristiano Giuffrida, and H. Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*.

[62] Integer Cube Root. [n.d.]. Integer Cube Root - Hacker's Delight [Book].

[63] Subhajit Roy. 2013. From Concrete Examples to Heap Manipulating Programs. In *Static Analysis: 20th International Symposium, SAS 2013, June 20-22, 2013. Proceedings*. https://doi.org/10.1007/978-3-642-38856-9_9

[64] S. Roy, J. Hsu, and A. Albarghouthi. 2021. Learning Differentially Private Mechanisms. In *2021 2021 IEEE SP*. IEEE Computer Society, 852–865. https://doi.org/10.1109/SP40001.2021.00060

[65] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *Proceedings of the 26th ACM Joint Meeting on ESEC/FSE (ESEC/FSE 2018)*. ACM, 224–234. https://doi.org/10.1145/3236024.3236084

[66] Subhajit Roy and Y. N. Srikant. 2009. Profiling K-Iteration Paths: A Generalization of the Ball-Larus Profiling Algorithm. 11 pages. https://doi.org/10.1109/CGO.2009.11

[67] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Non-Linear Loop Invariant Generation Using GröBner Bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on POPL (POPL '04)*. ACM, 318–329. https://doi.org/10.1145/964001.964028

[68] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[69] Fibonacci sequence. [n.d.]. Fibonacci sequence - Rosetta Code. http://rosettacode.org/wiki/Fibonacci_sequence.

[70] Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *CAV*. https://doi.org/10.1007/978-3-319-08867-9_6

[71] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *Programming Languages and Systems*. 574–592.

[72] Rahul Sharma, Aditya V. Nori, and Alex Aiken. 2012. Interpolants as Classifiers. In *CAV*. 71–87.

[73] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Proceedings of the 32nd International Conference NIPS (NIPS'18)*. Curran Associates Inc., 7762–7773.

[74] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2Inv: A Deep Learning Framework for Program Verification. In *CAV*.

[75] Dhruv Singal, Palak Agarwal, Saket Jhunjhunwala, and Subhajit Roy. 2018. Parse Condition: Symbolic Encoding of LL(1) Parsing. In *LPAR-22. 22nd International Conference LPAR (EPiC Series in Computing)*. EasyChair, 637–655. https://doi.org/10.29007/2ndp

[76] Integer square root. 2021. Integer square root - Wikipedia. https://en.wikipedia.org/wiki/Integer_square_root.

[77] SyGus. 2017. SyGuS-Comp 2017. https://sygus.org/comp/2017/.

[78] Gourav Takhar, Ramesh Karri, Christian Pilato, and Subhajit Roy. 2022. HOLL: Program Synthesis for Higher Order Logic Locking. In *TACAS*. 3–24.

[79] Aakanksha Verma, Pankaj Kumar Kalita, Awanish Pandey, and Subhajit Roy. 2020. Interactive Debugging of Concurrent Programs under Relaxed Memory Models. In *Proceedings of the 18th ACM/IEEE International Symposium CGO'20*. ACM, 68–80. https://doi.org/10.1145/3368826.3377910

[80] Sahil Verma and Subhajit Roy. 2017. Synergistic Debug-repair of Heap Manipulations. In *Proceedings of the 2017 11th Joint Meeting on ECSE/FSE (ESEC/FSE 2017)*. ACM. https://doi.org/10.1145/3106237.3106263

[81] Sahil Verma and Subhajit Roy. 2022. Debug-localize-repair: a symbiotic construction for heap manipulations. In *Formal Methods in System Design*.

[82] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, 497–512. https://doi.org/10.1109/SP.2010.37

[83] Wikipedia. 2021. John Selfridge — Wikipedia, The Free Encyclopedia.

[84] Maysam Yabandeh, Abhishek Anand, Marco Canini, and Dejan Kostic. 2011. Finding Almost-Invariants in Distributed Systems. In *2011 IEEE 30th International SRDS*. 177–182. https://doi.org/10.1109/SRDS.2011.29

[85] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations on Hierarchically Structured Data. In *Proceedings of PLDI'16*. ACM, 508–521. https://doi.org/10.1145/2908080.2908088

[86] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-Driven CHC Solver. In *Proceedings of the 39th ACM SIGPLAN Conference on PLDI (PLDI 2018)*. ACM, 707–721. https://doi.org/10.1145/3192366.3192416