Satisfiability Modulo Fuzzing: A Synergistic Combination of SMT Solving and Fuzzing

SUJIT KUMAR MUDULI, Indian Institute of Technology Kanpur, India
SUBHAJIT ROY, Indian Institute of Technology Kanpur, India

Programming languages and software engineering tools routinely encounter components that are difficult to reason on via formal techniques or whose formal semantics are not even available—third-party libraries, inline assembly code, SIMD instructions, system calls, calls to machine learning models, etc. However, often access to these components is available as input-output oracles—interfaces are available to query these components on certain inputs to receive the respective outputs. We refer to such functions as *closed-box functions*. Regular SMT solvers are unable to handle such closed-box functions.

We propose SĀDHAK, a solver for SMT theories modulo closed-box functions. Our core idea is to use a synergistic combination of a fuzzer to reason on closed-box functions and an SMT engine to solve the constraints pertaining to the SMT theories. The fuzz and the SMT engines attempt to converge to a model by exchanging a rich set of *interface constraints* that are relevant and interpretable by them. Our implementation, SĀDHAK, demonstrates a significant advantage over the only other solver that is capable of handling such closed-box constraints: SĀDHAK solves 36.45% more benchmarks than the best-performing mode of this state-of-the-art solver and has 5.72× better PAR-2 score; on the benchmarks that are solved by both tools, SĀDHAK is (on an average) 14.62× faster.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Program verification**; • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: SMT, Fuzzing, Closed-Box Function, Conflict-Driven Fuzz Loop

ACM Reference Format:
Sujit Kumar Muduli and Subhajit Roy. 2022. Satisfiability Modulo Fuzzing: A Synergistic Combination of SMT Solving and Fuzzing. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 169 (October 2022), 28 pages. https://doi.org/10.1145/3563332

1 INTRODUCTION

Thanks to the tremendous success of SMT solvers in the recent years, formula-based verification [Kroening and Tautschnig 2014; Leino 2008], testing [Cadar et al. 2008; Godefroid et al. 2005], repair [Goues et al. 2019; Mechtaev et al. 2016; Verma and Roy 2017] and synthesis [Polikarpova et al. 2016; Solar-Lezama 2013; Torlak and Bodik 2013] applications have gained a lot of traction. Such formula-based techniques encode the program under analysis into a logical formula, in some fragment of first-order logic, and use SMT solvers to reason on them. The success of such techniques in the recent years is not only due to better encodings, but also the significant improvements of SMT solvers.

Perhaps the most significant challenges to formula-driven software engineering systems are open programs—programs where parts of the software system are not formally specified. These could be due to use of third party libraries, inline assembly code, system calls, etc. At other times,

Authors' addresses: Sujit Kumar Muduli, Indian Institute of Technology Kanpur, Uttar Pradesh, India, smuduli@cse.iitk.ac.in; Subhajit Roy, Indian Institute of Technology Kanpur, Uttar Pradesh, India, subhajit@cse.iitk.ac.in.

2475-1421/2022/10-ART169
https://doi.org/10.1145/3563332

some components can be too complex to allow for logical reasoning (like, calls to deep learning models). Access to these components are mostly available as input-output oracles—interfaces to query the functions on certain inputs to get the respective outputs. We refer to such functions as *closed-box functions*; such functions have also been referred to as *oracle* calls in prior work [Polgreen et al. 2022]. The importance to reason *through* such closed-box functions have been identified in the recent years [Argyros et al. 2016; Dinges and Agha 2014; Godefroid 2011; Lahiri and Roy 2022; Mechtaev et al. 2018; Pandey et al. 2019; Polgreen et al. 2022; Păsăreanu et al. 2011].

In this work, we propose a Satisfiability Modulo Theory (SMT) solver, SĀDHAK, to support reasoning over such functions. Our algorithm solves SMT constraints including closed-box (CB) terms by establishing a *separation of concerns*—constraints from SMT theories are solved by an SMT engine, but constraints corresponding to closed-box functions are compiled to a program and handled by a fuzzer. Both the SMT engine and the fuzz engine *learn* from failures; while the SMT solver extracts out information about the closed-box functions from the partial models returned by the fuzzer, the fuzzer learns from conflicts within the SMT engine to only pull SMT theory terms that are *relevant* for satisfying the closed-box constraints. We term our strategy as *conflict-driven fuzz loop* (CDFL) as it uses a fuzzer in synergy with the core SMT engine, thereby adding support for *closed-box functions* (CB theory) within the existing set of theories supported by SMT solvers.

List. 1 shows an SMTLIB file [Barrett et al. 2010] containing `CB` terms (marked in green); we provide the `declare-cb` syntax to declare closed-box functions. The closed-box function **f**, accept two 32-bit integers and returns another 32-bit integer. We assume that the semantics of `f` is not available explicitly (e. g. as a logical formula); however, an input-output oracle for **f** is available that returns a concrete result when queried with some concrete inputs parameters.

Due to the presence of the CB terms, this SMTLIB query cannot be handled by an SMT solver. SĀDHAK handles this query by using the core SMT engine to handle the SMT constraints and a fuzzer to reason on the constraints involving the CB terms. The two engines exchange learnings from each other's failures to converge to a solution for the complete set of constraints. With this strategy, SĀDHAK responds with a satisfiable assignment within 5.59s. However, Delphi, a recent solver for CB theories, is unable to solve this example even within 600s. We also tried a popular strategy [Borzacchiello et al. 2021; Liew et al. 2019; Pandey et al. 2019] of using a fuzzer to reason on the complete set of SMT constraints as a whole (which is also available with SĀDHAK as an additional mode); this too fails to yield a solution within 600s.

Our algorithm is close to the Nelson-Oppen [Nelson and Oppen 1979] algorithm for theory combination, but exchanges a richer set of terms between the SMT core and the fuzzer allowing for more effective learning. We define a *parametric* closed-box (CB) theory that allows developer of decision procedures to define a *translation schema* for translating terms from their theory to program code, that can be consumed by fuzzer.

We build an instantiation of our ideas, SĀDHAK, within the CVC4 [Barrett et al. 2011] SMT solver. By building it within the SMT solver, we were able to achieve a more efficient integration with the core SMT engine and also reuse multiple components from the SMT solver (like the purification engine, SMT parser etc.). SĀDHAK is sound, that is any model returned by SĀDHAK is indeed a satisfiable assignment. To the best of our knowledge, SĀDHAK is the first solver that uses a powerful synergy of SMT solvers and fuzzers to solve SMT constraints with closed-box functions.

Our experimental results show that SĀDHAK is quite effective at solving SMT formulae with closed-box functions, solving 101 of our 107 benchmarks within a timeout of 600s. SĀDHAK significantly outperforms the state-of-the-art solver for closed-box functions, solves 36.45% more instances and

```
(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(declare-const z (_ BitVec 32))
(declare-const p (_ BitVec 32))
(declare-const q (_ BitVec 32))
(declare-const r (_ BitVec 32))

(declare-cb f ((_ BitVec 32) (_ BitVec 32)) (_ BitVec 32))

(define-fun ispow2 ((x (_ BitVec 32))) Bool
    (= (_ bv0 32) (bvand x (bvsub x (_ bv1 32)))))

(assert (bvugt x y))
(assert (and (bvult (_ bv255 32) z)) (bvult z (_ bv65536 32)))
(assert (= z (f x y)))

(assert (and (ispow2 p) (ispow2 q) (ispow2 r)))
(assert (and (bvugt p (_ bv1 32))
             (bvugt q (_ bv1 32)) (bvugt r (_ bv1 32)) ) )
(assert (= (_ bv64 32) (bvmul p q r)))

(check-sat)
```

List. 1. A motivating example.

with a 5.72× better PAR-2[1] score. On benchmarks solved by both SĀDHAK and the state-of-the-art solver, SĀDHAK is 14.62× faster on an average.

Contributions. In this paper we made following contributions:

(1) We introduce a new theory called *Closed-Box* (CB) theory and provide the requirements for establishing a sound interface with other theories;
(2) We propose our solving strategy, *Conflict-Driven Fuzz Loop (CDFL)* in Alg. 1, that allows solving of CB-theory constraints with other SMT theories;
(3) We build a tool, SĀDHAK, that augments a current satisfiability solver with support for closed box functions;
(4) We create a benchmark suite of 95 SMTLIB queries with closed-box constraints that can aid future investigations in this direction;
(5) We present a set of experiments on our benchmark suite of 95 SMTLIB queries and 12 SMTLIB queries from DELPHI [Polgreen et al. 2022] to demonstrate the utility and effectiveness of SĀDHAK.

Terminology. In this paper, we present SĀDHAK, that is implemented within an SMT solver for handling closed-box functions. An SMT solver has many components (like the parser, simplifier, purification engine etc.); we use the term *SMT Engine* to refer to the *core* of an SMT solver that decides the satisfiability of a given set of constraints across multiple first-order theories. We use the term *fuzz engine* to refer to the fuzzing-based solver within SĀDHAK that includes a constraint

[1]PAR-2 score is used scoring solvers in SAT competitions [Balyo et al. 2017]. It is calculated by adding runtimes for solved instances with two times the timeout for unsolved instances and then dividing it by the number of benchmarks. Lower PAR-2 score indicates better performance of the tool.

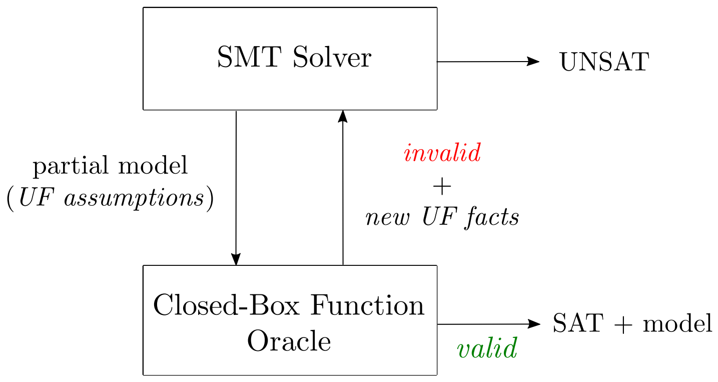


Fig. 1. Closed-Box Function based active learning.

compiler to compile constraints into a program and then fuzzes the program. The term *fuzzer* is used to refer to off-the-shelf fuzzers.

2 OVERVIEW

Our tool, SĀDHAK, is built to reason on constraints involving *closed-box functions*—functions for which *no formal specifications* are available. However, one may query these closed-box functions as an input-output oracle: querying a closed-box functions on a certain set of inputs returns the respective output. Such functions have also been referred to as *oracle* functions in some prior work [Polgreen et al. 2022].

Consider the SMTLIB query shown in List. 1. For the sake of clarity, we show the constraints from this query as a set of constraints in Eq. (1). The function *isPow*2 checks if a given input is power of 2 or not using a bitwise operation. The lambda notation $\lambda x.\ G$ defines a function that accepts an input x; the function is invoked by replacing the actual argument by all free instances of x in the function "body" G.

These constraints involve a call to the closed-box function $\mathbf{f}()$, and hence, it cannot be handled by a regular SMT solver. We assume that we can query the closed-box implementation of `f()`, i.e. we can query `f()` for the output on a provided set of *concrete* inputs. We use $[\![$`f()`$]\!]$ to denote a concrete function invocation of the closed-box function `f`. We show the C program for the function that we use as our closed-box function in List. 3.

$$\begin{aligned} & \mathbf{f}(x,y) > 255 \land \mathbf{f}(x,y) < 65536 \land (x > y) \\ & \land (p > 1) \land (q > 1) \land (r > 1) \\ & \land isPow2(p) \land isPow2(q) \land isPow2(r) \\ & \land (p \times q \times r = 64) \\ & \land (isPow2 = \lambda x.((x \mathbin{\&} (x-1)) = 0)) \end{aligned} \tag{1}$$

We now discuss about three strategies to solve constraints with closed-box functions and discuss their merits and demerits.

2.1 Strategy #1: SMT Solver-Driven Active Learning

Fig. 1 provides an outline of this strategy [Polgreen et al. 2022]. In this strategy, the SMT solver proposes a partial model by assuming the closed box function as an uninterpreted function. The

```c
uint32_t isPow2(uint32_t x);

int main() {
  uint32_t x, y, p, q, r;
  READ_INPUT(x, y, p, q, r);

  if(255 < f(x,y))
    if(f(x,y) < 65536)
      if (x > y)
        if (p > 1 && q > 1 && r > 1)
          if (isPow2(p) && isPow2(q) && isPow2(r))
            assert(0);

  return 0;
}

uint32_t isPow2(uint32_t x) {
    return 0 == (x & (x - 1));
}
```

List. 2. Translation of all constraints in Eq. (1) to be fuzzed.

```c
uint32_t f (uint32_t a, uint32_t b) {
    return a * b;
}
```

List. 3. Closed-Box function `f()` used in List. 1.

possible model is checked against the closed-box functions—if the model is verified against the closed-box functions, the model is returned. Else, the result from the closed box operation is used to construct a counterexample and propagated to the SMT solver. The SMT solver *learns* from this failure by adding an additional constraint to capture the behavior of the closed-box function on the previous partial model. Then, it proposes a new partial model to be checked against the closed-box functions. This process is repeated till a partial model is verified against the closed-box functions.

For our example, modelling the closed-box function `f()` as an uninterpreted function (UF), the SMT solver comes up with the following model: $p = 0, q = 0, r = 0, x = 1, y = 0, f(x, y) = \lambda x, y.\ 127$. This strategy, then, tries to verify this model with respect to the closed-box function by checking whether $f(1, 0) = 127$ is consistent. The closed-box function is executed with the given arguments: in this case, $[\![f(1, 0)]\!]$ returns 0. Thus, the candidate model proposed by the SMT solver is not valid and the SMT solver learns a new fact that $f(1, 0) = 0$. The SMT solver, now, attempts to learn a new model that also satisfies the learnt fact and this process repeats until the closed-box oracle verifies the proposed candidate model is consistent w.r.t $f()$.

In this strategy, the learning is inefficient as every counterexample eliminates a very few candidate models.

2.2 Strategy #2: A Fuzz-Only Strategy

An alternative strategy could be to completely eliminate the SMT solver and only use a *fuzzer* to search for satisfiable models [Borzacchiello et al. 2021; Liew et al. 2019; Pandey et al. 2019]. Such

Table 1. SĀDHAK's state at iteration 1.

Iteration #1	
Fuzz Engine	$z = \mathbf{f}_{cb}(x, y)$
Partial Model	$x = 0, y = 0, z = 0$
SMT Engine	$z = \mathbf{f}_{uf}(x, y) \wedge (z > 255) \wedge (z < 65536) \wedge (x > y)$ $\wedge (p > 1) \wedge (q > 1) \wedge (r > 1)$ $\wedge\ isPow2(p) \wedge isPow2(q) \wedge isPow2(r) \wedge (p \times q \times r = 64)$

Table 2. SĀDHAK's state at iteration 2.

Iteration #2	
Fuzz Engine	$z = \mathbf{f}_{cb}(x, y) \wedge (x > y)$
Partial Model	$x = 1, y = 0, z = 0$
SMT Engine	$z = \mathbf{f}_{uf}(x, y) \wedge (z > 255) \wedge (z < 65536) \wedge (x > y)$ $\wedge (p > 1) \wedge (q > 1) \wedge (r > 1)$ $\wedge\ isPow2(p) \wedge isPow2(q) \wedge isPow2(r) \wedge (p \times q \times r = 64)$ $\wedge\ \mathbf{f}_{uf}(0, 0) = 0$

Table 3. SĀDHAK's state at iteration 3.

Iteration #3	
Fuzz Engine	$z = \mathbf{f}_{cb}(x, y) \wedge (x > y) \wedge (z > 255)$
Partial Model	$x = 65536, y = 1, z = 65536$
SMT Engine	$z = \mathbf{f}_{uf}(x, y) \wedge (z > 255) \wedge (z < 65536) \wedge (x > y)$ $\wedge (p > 1) \wedge (q > 1) \wedge (r > 1)$ $\wedge\ isPow2(p) \wedge isPow2(q) \wedge isPow2(r) \wedge (p \times q \times r = 64)$ $\wedge\ \mathbf{f}_{uf}(0, 0) = 0 \wedge \mathbf{f}_{uf}(1, 0) = 0$

proposals translate the complete set of constraints to a program and submit it to a fuzzer to find a set of inputs that can reach a "goal" program location. This strategy reduces a satisfiability query on a set of constraints to a reachability query on a program.

For example, List. 2, shows a translation of constraints in Eq. (1) to a C program. Note that inputs that reach the `assert(0)` (crashing input) will also satisfy the set of SMT constraints and hence, can be extracted as a satisfying assignment.

However, this scheme, too, is inefficient as it requires the fuzzer to get past complex relational constraints—constraints that an SMT solver is more adept at solving.

2.3 Strategy #3: Conflict-Driven Fuzz Loop (CDFL)

In contrast to the above strategies, our proposal combines an SMT solver and a fuzzer in a synergistic combination. The SMT solver loads the constraints corresponding to the respective SMT theories while the fuzzer only reasons on the closed-box functions. The fuzzer starts by creating a partial model: if this models from the fuzzer conflicts with the constraints within the SMT solver, the fuzzer analyzes the conflict and *lazily* borrows only the *relevant* constraints that led to the conflict. It, then, proposes a new model to the SMT solver. This process continues till the fuzzer is able to construct a model that is consistent with the constraints loaded in the SMT solver. As the fuzzer

Table 4. Sādhak's state at iteration 4.

Iteration #4	
Fuzz Engine	$z = \mathbf{f}_{cb}(x, y) \wedge (z > 255) \wedge (x > y) \wedge z < 65536$
Partial Model	$x = 256, y = 1, z = 256$
SMT Engine	$z = \mathbf{f}_{uf}(x, y) \wedge (z > 255) \wedge (z < 65536) \wedge (x > y)$ $\wedge (p > 1) \wedge (q > 1) \wedge (r > 1)$ $\wedge\, isPow2(p) \wedge isPow2(q) \wedge isPow2(r) \wedge (p \times q \times r = 64)$ $\wedge\, \mathbf{f}_{uf}(0, 0) = 0, \mathbf{f}_{uf}(1, 0) = 0, \mathbf{f}_{uf}(65536, 1) = 65536$

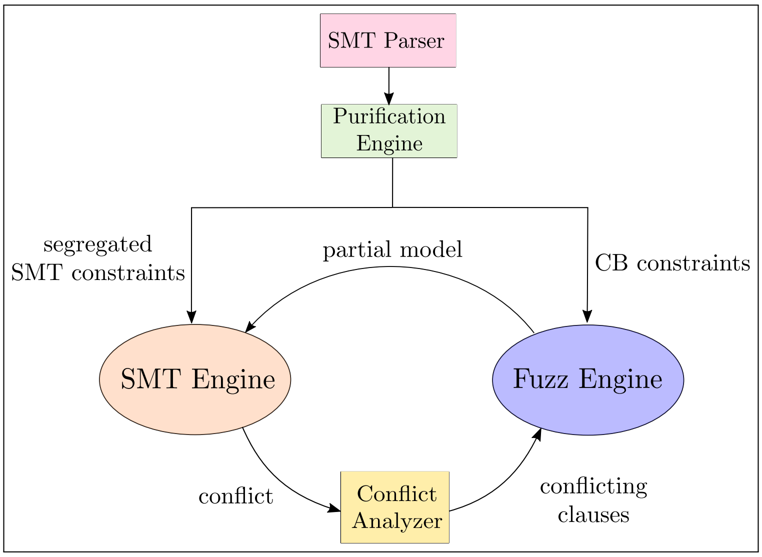


Fig. 2. Overview of Sādhak's architecture.

iteratively proposes new models guided by conflicts from the SMT solver, we refer to this strategy as *conflict-driven fuzz loop* (CDFL).

We show an overview of our strategy, conflict-driven fuzz loop (CDFL), in Fig. 2. We, now, describe its operation via an example:

Theory Segregation. The set of constraints are segregated into individual terms—those that correspond to the closed-box functions and those pertaining purely to SMT theories. All the terms corresponding to the SMT theories are dispatched to the SMT engine. All constraints corresponding to closed-box (CB) terms are handled as follows:

(1) a copy of the CB constraints are translated to UF (uninterpreted functions) and handled by the SMT engine. This ensures that any candidate model from the SMT solver satisfies the function axioms on the CB functions. We designate the instance of the closed-box function within the SMT engine as $\mathbf{f}_{uf}$ (as the closed-box function is being interpreted as an UF in these constraints.
(2) the CB constraints are added to the queue of the fuzz engine. We denote the instance of the closed-box function within the fuzz engine as $\mathbf{f}_{cb}$.

After theory segregation on the set of constraints in Eq. (1), the segregated constraints are shown in Tab. 1 for SMT and fuzz engines.

Fuzz Solving. Sādhak constructs a program from the closed-box (CB) constraints and attempts to construct a partial candidate model. The CB constraints in Tab. 1 get translated to a C program shown in List. 4. The program is compiled to an executable and is fuzzed with invocations to the closed-box functions at the respective locations.

For our example, the fuzz engine finds a crashing input $(x = 0, y = 0, z = 0)$ for the C program. The candidate partial model $(x = 0, y = 0, z = 0)$ is then propagated to the SMT engine.

Sādhak also learns some facts about the CB functions from the partial model communicated from the fuzz engine. For our example, Sādhak learns that $(\mathbf{f}_{uf}(0, 0) = 0)$, which is added to the SMT engine constraints queue.

SMT Theory Solving. The SMT engine, then, attempts to search for a *completion* of the partial candidate model. The respective theory solvers within the SMT engine handle the respective constraints to search for a model. If a completion is found, the full model is returned; else, we progress to the next stage.

In our example, the SMT solver returns unsatisfiable and we progress to the next stage.

Conflict Analysis. In case the SMT engine returns unsatisfiable for the partial model, we analyze the conflict to identify the conflicting terms. The fuzz engine learns from this conflict and augments its set of constraints with these conflicting terms.

For our example, the partial model $(x = 0, y = 0, z = 0)$ from the fuzz engine led to a conflict due to the constraint $(x > y)$. The conflicting clause for fuzz engine's partial model is highlighted with red in Tab. 1. The conflict learnt from this phase is propagated to fuzz engine to find a new partial candidate model which will satisfy $(x > y)$.

Repeat. The fuzz engine is fired again with the augmented set of constraints and the loop is repeated. Tab. 2 shows the state of theory solvers and fuzz engine at iteration #2 after conflict propagation from the previous iteration. The constraints learnt from previous iteration is highlighted with blue. The constraint set at the fuzz engine's queue in Tab. 2 gets translated to List. 5 and is fuzzed for another model.

Let us walk through the next set of iterations en route to reaching a satisfiable model:

- The fuzz engine now returns another candidate partial model $(x = 1, y = 0, z = 0)$ and propagates it to the SMT engine. As the set of constraints contains $\mathbf{f}_{uf}(x, y) = z$ and the partial model includes $(x = 1, y = 0, z = 0)$, Sādhak learns a new fact $\mathbf{f}_{uf}(1, 0) = 0$. This time the SMT engine finds that this partial model conflicts with the constraint $(z > 255)$ in the SMT engine's queue (see Tab. 2). The SMT engine now propagates this new conflicting constraint to the fuzz engine.
- In iteration #3, the fuzz engine's queue in Tab. 3 gets translated to a C program as shown in List. 6 and fuzzed for a model. The candidate partial model $(x = 65536, y = 1, z = 65536)$ is propagated to the SMT engine and Sādhak also learns the fact $\mathbf{f}_{uf}(65536, 1) = 65536$ from this partial model. Now the partial model conflicts with the constraint $(z < 65536)$ in the SMT engine's queue (see Tab. 3).
- In iteration #4, the fuzz engine tries to find another candidate partial model for the set of CB constraints at fuzz engine's queue in Tab. 4. The constraint set at fuzz engine's queue is translated into a C program shown in List. 7 and fuzzed for a new model. This time a candidate partial model $(x = 256, y = 1, z = 256)$ is found and propagated to the SMT engine.

```
uint32_t f (uint32_t, uint32_t);

int main() {
    uint32_t x,y,z;
    READ_INPUT(x,y,z);

    if(z == f(x,y))
      assert(0);

    return 0;
}
```

List. 4. Program generated by fuzz-engine at iteration #1.

```
uint32_t f (uint32_t, uint32_t);

int main() {
    uint32_t x,y,z;
    READ_INPUT(x,y,z);

    if(z == f(x,y))
      if (x > y)
        assert(0);

    return 0;
}
```

List. 5. Program generated by fuzz-engine at iteration #2.

```
uint32_t f (uint32_t, uint32_t);

int main() {
  uint32_t x,y,z;
  READ_INPUT(x,y,z);

  if(z == f(x,y))
    if (x > y)
      if(z > 255)
        assert(0);

  return 0;
}
```

List. 6. Program generated by fuzz-engine at iteration #3.

```
uint32_t f (uint32_t, uint32_t);

int main() {
    uint32_t x,y,z;
    READ_INPUT(x,y,z);

    if(z == f(x,y))
      if (x > y)
        if(z > 255)
          if(z < 65536)
            assert(0);

    return 0;
}
```

List. 7. Program generated by fuzz-engine at iteration #4.

This partial model does not conflict with any of the constraints in other SMT theory solvers. The SMT engine finally returns a complete model ($x = 256, y = 1, z = 256, p = 2, q = 8, r = 4$).

Our algorithm establishes a *separation of concerns*: the SMT engine attempts to solve the SMT constraints (in what it is adept at), the fuzz engine uses fuzzing to search for models satisfying the closed-box constraints (that cannot be handled by the SMT solver).

Our strategy has the following advantages:

- The fuzzer is not overloaded with a large number of complex SMT constraints (in contrast to strategy #2), but only a smaller set of constraints that are relevant to the CB constraints;
- The SMT solver exchanges a rich set of constraints with the fuzzer to allow for efficient learning (in contrast to strategy #1);
- The fuzzer gets the advantage of lemma learning capabilities of the SMT engine; in many cases, instead of the base constraints, more informative lemmas learnt by the SMT engine is pulled into the fuzzer;

Table 5. Constraints solved by fuzzing in fuzz-only and CDFL mode.

Fuzz-Only	**CDFL**
$\mathbf{f}_{uf}(x,y) > 255 \land \mathbf{f}_{uf}(x,y) < 65536$ $\land (x > y) \land (p > 1) \land (q > 1) \land (r > 1)$ $\land isPow2(p) \land isPow2(q) \land isPow2(r)$ $\land (p \times q \times r = 64)$	$z = \mathbf{f}_{cb}(x,y) \land (x > y)$ $\land (z > 255) \land z < 65536$

- The fuzzer can leverage the support for theory combination within the SMT solver. Thanks to theory combination algorithms, SĀDHAK can support multiple theories.

For example, in the fuzz-only mode, the fuzzer needs to find a satisfying solution for the full set of constraints (as shown in the first column of Tab. 5). However, the set of constraints that the fuzzer needs to handle, even in the last itertaion of SĀDHAK (shown in second column of Tab. 5), is much smaller.

Readers may find similarities of our algorithm with the popular Nelson-Oppen theory combination algorithm. However, while Nelson-Oppen [Nelson and Oppen 1979] only exchanges interface equalities across the decision procedures, SĀDHAK exhanges a richer set of constraints between the core SMT engine and the fuzz engine, allowing the fuzzer to propose better candidates for the partial models.

To summarize, strategy #1 (SMT-driven active learning strategy) leads to inefficient learning as it communicates via examples and, most often, may fail to generalize. However, SĀDHAK uses a theory combination stratergy inspired by the Nelson-Oppen to communicate via constraints over interface variablesand lead to more efficient learning. On the hand, strategy #2 (fuzz-only strategy) translates the whole set of constraints into a program to be fuzzed, while SĀDHAK *lazily* adds only the *relevant* constraints to the program to be fuzzed.

SĀDHAK solves the above example (List. 1) **in less than 6 seconds**. However, both Delphi [Polgreen et al. 2022] (a tool that uses the active learning strategy) and the fuzz-only strategy (implemented within SĀDHAK), **fail to solve** the example within a 10 min (600s) timeout.

SĀDHAK implements both the fuzz-only and the CDFL strategies, but we found the CDFL strategy to be superior and is the focus of this paper.

3 THE THEORY OF CLOSED-BOX FUNCTIONS (CB THEORY)

SĀDHAK provides support for including closed-box functions with any theories supported by the baseline SMT engine. To allow for efficient learning and more informative exchange of facts, SĀDHAK allows the developer of decision procedures a mechanism to exchange a richer set of interface constraints. This improves the fuzzer to learn better from the failures of the SMT engine, leading to faster convergence. Without this, the fuzz engine will only be able to exchange information via interface equalities (as in the Nelson-Oppen algorithm for theory combination).

CB Theory. Given a first-order theory $\mathcal{T}$, the theory fragment for *closed-box functions* over the theory $\mathcal{T}$ (denoted as $CB_{\mathcal{T}}$) has the following signature:

$$\Sigma_{CB^{\mathcal{T}}} = \langle \mathcal{S}_{\mathcal{T}}, C_{\mathcal{T}}, \mathcal{F}_{\mathcal{T}}, \mathcal{F}^{cb}_{\mathcal{T}}, \mathcal{B}_{\mathcal{T}}, \mathcal{R}_{\mathcal{T}} \rangle$$

where,

- $\mathcal{S}_{\mathcal{T}}$: set of sorts in theory $\mathcal{T}$;
- $C_{\mathcal{T}}$: set all (sorted) constants in $\mathcal{T}$;
- $\mathcal{F}_{\mathcal{T}}$: set of all (sorted) function symbols in $\mathcal{T}$;

Table 6. Example of a bit-vector theory translation schema ($\mathcal{R}_{BV}$) defined in CB theory.

Symbol	BitVector expression	C language expression
Sort	`(_ BitVec 8)`	`uint8_t`
	`(_ BitVec 32)`	`uint32_t`
	`(_ BitVec 64)`	`uint64_t`
Function	`(bvadd a b)`	`a + b`
	`(bvand a b)`	`a & b`
	`(bvmul a b)`	`a * b`
Predicate	`(bvuge a b)`	`(a >= b)`
	`(bvugt a b)`	`(a > b)`
	`(bvult a b)`	`(a < b)`

- $\mathcal{F}^{cb}_{\mathcal{T}}$: set of (sorted) closed-box functions and each function's sort is constructed from the sorts in $S_{\mathcal{T}}$;
- $\mathcal{B}_{\mathcal{T}}$: predicates from theory $\mathcal{T}$;
- $\mathcal{R}_{\mathcal{T}}$: schema for translating sorts and expressions in $\mathcal{T}$ to a program code.

We lift the theory of closed-box functions over a set of first-order theories, $\mathcal{T}_1, \ldots, \mathcal{T}_n$, by taking a point-wise union over the respective components in the signature. Formally, $\mathrm{CB}^{\{\mathcal{T}_1,\ldots,\mathcal{T}_n\}} = \bigcup_{i=1}^{n} \mathrm{CB}_{\mathcal{T}_i}$ has a signature

$$\Sigma_{CB^{\{\mathcal{T}_1,\ldots,\mathcal{T}_n\}}} = \langle \mathcal{S},\ C,\ \mathcal{F},\ \mathcal{F}^{cb},\ \mathcal{B},\ \mathcal{R} \rangle$$

where, $\mathcal{S} = \bigcup_{i=1}^{n} \mathcal{S}_{\mathcal{T}_i}$, $C = \bigcup_{i=1}^{n} C_{\mathcal{T}_i}$, $\mathcal{F} = \bigcup_{i=1}^{n} \mathcal{F}_{\mathcal{T}_i}$, $\mathcal{F}^{cb} = \bigcup_{i=1}^{n} \mathcal{F}^{cb}_{\mathcal{T}_i}$, $\mathcal{B} = \bigcup_{i=1}^{n} \mathcal{B}_{\mathcal{T}_i}$ and $\mathcal{R} = \bigcup_{i=1}^{n} \mathcal{R}_{\mathcal{T}_i}$. For simplicity we drop the superscript of $CB^{\{\mathcal{T}_1,\ldots,\mathcal{T}_n\}}$ and use CB in the future references.

Translation Schema. Any SMT theory that intends to operate efficiently with closed-box functions needs to provide the following:

- **Augmented CB vocabulary ($\mathcal{S}_{\mathcal{T}}$, $C_{\mathcal{T}}$, $\mathcal{F}_{\mathcal{T}}$, $\mathcal{B}_{\mathcal{T}}$).** The CB vocabulary can be extended to include a subset of the vocabulary of the given theory. This provides a richer set of interface constraints for information exchange between the SMT and fuzz engines.
- **Translation schema ($\mathcal{R}_{\mathcal{T}}$).** For the augmented CB vocabulary from the theory, a translation schema needs to be provided to translate SMT constraints into program code. The translation schema contains two types of translations:
  - Sort translation: This describes how sorts in a theory $\mathcal{T}$ are translated to program types;
  - Term translation: This describes how terms in a theory $\mathcal{T}$ are translated to program expressions;

Sādhak includes a complete translation schema for the BV (bitvector) theory. Tab. 6 shows some examples from the translation schema. We omit details for brevity.
We require that the function symbols $\mathcal{F}^{cb}$ are *functional*, i.e. they return a unique, deterministic value for a given set of arguments. The closed-box functions can also be partial functions (that may not guarantee termination) but must be deterministic. Formally, we assume the axiom of function congruence on any closed-box function symbol $f_{cb} \in \mathcal{F}^{cb}$ (Eq. (2)):

$$\forall x_1, \ldots, x_n,\ y_1, \ldots, y_n.\ (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n) \longrightarrow f_{cb}(x_1, \ldots, x_n) = f_{cb}(y_1, \ldots, y_n) \qquad (2)$$

The translation schema and the implementations of the closed-box functions must ensure that the above axiom is satisfied. If this condition is met, it is easy to see that formulae in the CB theory satisfy the following property:

```
int cabs(int x) {
    return (x < 0)? -x : x;
}
```

List. 8. Definition of closed-box function cabs.

Property. If a formula Ω is satisfiable on the CB theory (say, on an interpretation $\mathcal{I}$), then, $\mathcal{I}$ will also satisfy the formula Ω_{uf}, where the CB symbols are interpreted as UF symbols (the theory of uninterpreted functions):

$$\forall \mathcal{I}.\mathcal{I} \models \Omega \Rightarrow \mathcal{I} \models \Omega_{uf}$$

In other words, all models of Ω are also models of Ω_{uf}.

For example, consider the constraint $\Omega = (x < 0) \land (\mathtt{cabs}(x) > 0)$, where **cabs** is a closed-box function (defined in List. 8). Let $\mathtt{cabs}_{uf}$ be an uninterpreted function, we define Ω_{uf} as $\Omega_{uf} = (x < 0) \land (\mathtt{cabs}_{uf}(x) > 0)$. We can see that:

- One possible satisfiable assignment for Ω is $(x = -1)$, and it also satisfies Ω_{uf}, where $\mathtt{cabs}_{uf}$ could be simply $\lambda x.1$;
- However, the implication may not work in the other direction. For example, $(x = -1, \mathtt{cabs}_{uf} = \lambda x.2)$ is a valid model for Ω_{uf} but it does not satisfy Ω.

SMTLIB syntax. We augment the SMTLIB syntax with the `declare-cb` construct to define closed-box functions. For example, a closed-box function `cfun` that takes two 32-bit bitvector arguments and returns a 32-bit bitvector can be written as:

```
(declare-cb cfun ((_ BitVec 32) (_ BitVec 32)) (_ BitVec 32))
```

4 ALGORITHM

We show our algorithm in Alg. 1. Given a formula Ω, the first step involves segregating (line 1) the terms into two sets, ω_{smt} is presented to the SMT engine queue and ω_{fuzz} is added to the queue of the fuzz engine. Next, the algorithm uses the fuzz engine to "guess" a partial model for the CB terms (line 3), and, then, it seeks a *completion* of this model via the SMT engine (line 6). If the model completion is successful the algorithm returns SAT with a complete model (line 8). If partial model generated by fuzz engine conflicts with the terms in the SMT engine, model completion fails (line 11). In this case, the algorithm analyzes the conflict to *pull relevant terms* (within the CB theory) into the fuzz engine to refine its search (line 15). We now discuss the algorithm in detail.

4.1 Segregate

The Segregate method is a two-step process:

Purification. Our purification step is similar to the purification algorithm in the Nelson-Oppen theory combination strategy [Nelson and Oppen 1979]. Given a formula φ that has terms from both SMT theories and closed-box function invocations, the purification step decomposes the formula φ into a set of (conjunction over) simpler terms such that all terms are either purely from the SMT theories or are simply invocations of some closed-box function. This is achieved by adding new temporary variables that decompose complex terms into a conjunction of simpler terms, constrained by equalities.

Separation. In the second step, Segregate propagates only the CB terms to our fuzz engine while the rest of the terms are added to the queue of the SMT engine. However, all the closed-box

Algorithm 1: Sādhak solver algorithm.

```
Input: $\Omega$: formula to be checked
1  $\omega_{smt}, \omega_{fuzz} \leftarrow$ SEGREGATE($\Omega$)
2  while True do
3      $r_1, m_1 \leftarrow$ FUZZENGINE($\omega_{fuzz}$)
4      if $r_1$ = TIMEOUT then
5          return UNKNOWN
6      $r_2, m_2, \omega_{lem}, \omega_c \leftarrow$ SMTENGINE($\omega_{smt}, m_1$)
7      if $r_2$ = SAT then
8          return (SAT, $m_1 \cup m_2$)
9      else if $r_2$ = TIMEOUT then
10         return UNKNOWN
11     else
12         if $\omega_c \cap m_1 = \emptyset$ then
13             return UNSAT
14         $\omega_{smt} \leftarrow \omega_{smt} \cup$ INDPENDENT($\omega_{lem}, m_1$) $\cup$ CONVERTCBTOUF($\omega_{smt}, m_1$)
15         $\omega_{fuzz} \leftarrow \omega_{fuzz} \cup$ FILTER($CB_{\mathcal{T}}$, INDEPENDENT($\omega_c, m_1$))
```

Algorithm 2: FuzzEngine

```
Input: $\omega$: constraint for satisfiability checking
1  $P \leftarrow$ COMPILE($\omega$)
2  $result, m \leftarrow$ FUZZ($P$)
3  if result = TIMEOUT then
4      return TIMEOUT, None
5  else
6      return SAT, $m$
```

function invocations are also rewritten as UF (uninterpreted functions) invocations and are added to the queue of the SMT engine. This ensures that any model generated by the SMT engine satisfies function congruence over the CB functions. For a closed-box function **f**, we denote the closed-box invocation added to the fuzz engine queue as $\mathbf{f}_{cb}$ and that added to the SMT engine queue (as an UF) as $\mathbf{f}_{uf}$.

For example, Fig. 3 shows the *purification* of the formula $\boxed{x + foo(y) > 0}$ (shown in yellow box). The green box shows the set of formulas generated after our purification step; an additional variable t is introduced to separate the constraints. In the *separation* step, the closed-box function foo (in the green box) is re-written as an uninterpreted function $\texttt{foo}_{uf}$ for the SMT engine constraints queue (red box) and $\texttt{foo}_{cb}$ for the fuzz engine constraints queue (shown in the blue box).

4.2 SMTEngine

SMTEngine could be a standard DPLL(T)[Ganzinger et al. 2004; Nieuwenhuis et al. 2006] based solver that is capable of solving for satisfiability over a set of theories. The theories are combined

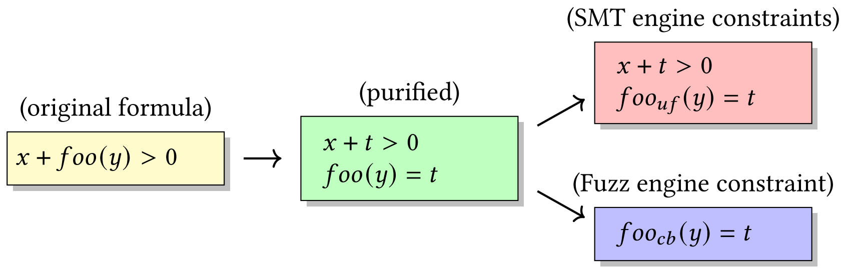


Fig. 3. Example showing segregation.

using a theory combination algorithm like Nelson-Oppen [Nelson and Oppen 1979]. The SMT engine accepts the set of constraints ω and a set of assumptions (φ). SMTEngine(ω, φ) computes the following:

- a result r, that could be SAT, UNSAT, UNKNOWN;
- if the result is SAT, it returns a model m as a set of assignments to the variables appearing in the formula that satisfies ω and φ;
- a set of lemmas, ω_{lem}, that are additional terms it learns while running DPLL(T) and the theory combination algorithms; these lemmas are learnt by theory solvers during theory reasoning;
- if the result is UNSAT, it returns a set of *conflict terms* (sometimes referred to as an *unsatisfiable core*) ω_c; intuitively, ω_c captures the *reason* of unsatisfiability of $\omega \cup \varphi$;

4.3 FuzzEngine

FuzzEngine implements the core decision procedure for the closed-box theory. FuzzEngine attempts to solve the set of constraints presented to its constraint queue to compute a model such that the constraints are satisfied modulo the execution semantics of the participating closed-box functions. FuzzEngine (see Alg. 2) has two main steps, viz. Compile, and Fuzz.

4.3.1 Compile. This procedure uses our *constraint compiler* to compile the presented set of constraints into an executable program that invokes calls to the closed-box functions. Sect. 5.3 provides the compilation schematic: the participating variables in the set of constraints are declared and read as inputs. Then it compiles an if-ladder where each contraint is translated to its respective executable semantics as per the provided translation rules $\mathcal{R}_{\mathcal{T}}$ (see Sect. 3) and instantiated in the conditional guards. Note that every assignment of the input variables that reaches the assertion (crashing input) will be a satisfiable assignment for the respective set of constraints. Further, as the executable calls to the closed-box functions are available, this program can be linked and executed *through* these closed-box functions. In summary, our constraint compiler reduces the problem of satisfiability over a set of constraints to the problem of reachability in an executable program containing closed-box functions.

4.3.2 Fuzz. We, then, use an off-the-shelf fuzzer to fuzz the program generated by the constraint compiler in a search for crashing inputs (and hence, a satisfiable model for the presented set of constraints). If such inputs are found, it returns these inputs as the model with a *SAT* verdict. However, if the fuzzer runs out of the budgeted time, FuzzEngine returns *Timeout*.

4.4 Conflict-Driven Fuzz Loop (CDFL)

Our core algorithm (Alg. 1) makes use of some auxiliary functions:

- ConvertCBToUF(ω, m). Given a set of constraints ω and a model m, ConvertCBToUF learns new UF theory facts from the model from UF term in ω and the corresponding assignments in m. For example, let ω contain two UF theory terms $w = g(u, v)$ and $z = f(x, y)$ and the model contains assignment $(u = 1, v = 2, w = 5, x = 0, y = 3, z = 7)$. ConvertCBToUF($\omega$, m) will return new facts $g(1, 2) = 5, f(0, 3) = 7$.
- Independent(ω, m). Given a set of constraints ω and a model m, Independent(ω, m) extracts constraints from ω that hold irrespective of the model m (i.e. those that are independent of m). We explain the implementation of Independent in Sect. 5.4.
- Filter($\mathcal{T}$, ω). The function Filter is used to extract out terms corresponding to a certain theory ($CB_{\mathcal{T}}$ in this case) from the set of constraints ω. For example, let ω be {(bvugt x y), u + v > 0, g(w) = 1, (bvuge z 0)} then Filter($\mathcal{T}_{BV}$, ω) will return {(bvugt x y), (bvuge z 0)}.

We are now in a position to explain our core algorithm (see Alg. 1). After the initial segregation step (line 1) that populates the constraints for the queues of the SMT (ω_{SMT}) and fuzz (ω_{fuzz}) engines, we enter into the conflict-driven fuzz loop (CDFL) in a search for a satisfiable assignment to the set of constraints modulo closed-box functions (line 2 to line 15). We start off (line 3) by using FuzzEngine on ω_{fuzz} to compute a partial model m_1 for $\omega_{fuzz} \subseteq \Omega$. If our solver runs out of its budgeted time (line 5), we exit with an UNKNOWN verdict.

Otherwise, we invoke SMTEngine (line 6) in seach of a completion for the partial model m_1 on the set constraints ω_{SMT} (by invoking SMTEngine with constraints ω_{SMT} and assumptions m_1). If such a model is found, we return the completed model $m_1 \cup m_2$ with the SAT verdict (line 8). On the other hand, if the SMT engine runs out of the budgeted time, we exit with an UNKNOWN verdict (line 10).

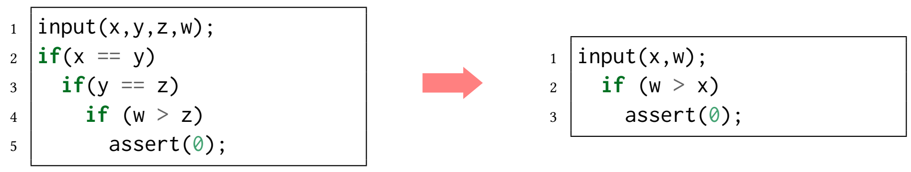
However, if the SMT engine detects a conflict, we attempt to analyze the reason of the conflict. If the set of conflict terms do not include the assignments from the partial model (line 13), it indicates that the constraints are unsatisfiable irrespective of the closed-box functions. In this case, we return the verdict as UNSAT. For example, in Tab. 1, the unsat core ω_c is $\{(x > y), x = 0, y = 0\}$ which includes the model assignments $\{x = 0, y = 0\}$; hence, in this case we cannot return with UNSAT and must move to line 14.

Otherwise (i.e. when the assignments in the model are involved in the conflict), we *learn* from this failure by augmenting the set of constraints, *both for the SMT and the fuzz engines*:

- **SMT Engine.** In this case, we add all the new lemmas that were discovered during the last invocation of the SMT engine *that do not depend on the assumptions*. The function Independent extracts all the set of constraints from ω_{lem} that hold irrespective of m_1. Also, the outputs of the closed-box functions discovered during the fuzzer runs, that are captured in the model m_1 are also extracted and added to ω_{SMT} as UF terms. This refines the definitions of these functions within the SMT engine (line 14).
- **Fuzz Engine.** All the terms in the conflicting terms ω_c *that belong to the CB theory* are relevant for the CB terms (as they caused conflicts). Hence, we add such constraints to ω_{fuzz} (line 15). In this case the function Filter extracts out all the terms corresponding to the $CB_{\mathcal{T}}$ from the set of constraints in ω_c. We again use the function Independent to extract constraints from ω_c that hold irrespective of the model m_1.

Theorem. Alg. 1 is sound, i.e. it returns a model m only if it satisfies the set of constraints Ω with respect to the executable semantics of the participating closed-box functions.

We omit the detailed proof for brevity. However, the primary observation used in the proof is that a model $(m_1 \cup m_2)$ is returned only when:



```
input(x,y,z,w);
if(x == y)
  if(y == z)
    if (w > z)
      assert(0);
```

```
input(x,w);
  if (w > x)
    assert(0);
```

Fig. 4. Handling equality constraints in target program.

(1) m_1 is discovered by the FuzzEngine at line 3 (so m_1 satisfies the executable semantics of the closed box functions), and
(2) m_1 is consistent with the completion m_2 from the SMT engine at line 6 - line 8 (and hence, $m_1 \cup m_2$ also satisfy the rest of the constraints in Ω).

Also, our algorithm ensures progress as at least one of ω_{smt} or ω_{fuzz} is definitely augmented in every iteration of the loop. Due to the use of fuzzing, Sādhak does not guarantee termination; users can run Sādhak with a timeout to ensure that their runs terminate.

5 IMPLEMENTATION

We build Sādhak within the popular CVC4 [Barrett et al. 2011] SMT solver. Sādhak reuses many of its components (e. g., SMTLIB parser, purification engine etc.) to provide support for CB-theories.

We use AFL++ [Fioraldi et al. 2020] for fuzzing C files generated from CB constraints. The generated C file is compiled with the compiler wrapper afl-clang-lto along with `AFL_LLVM_CMPLOG`, `AFL_LLVM_LAF_SPLIT_COMPARES`, `AFL_LLVM_LAF_TRANSFORM_COMPARES` to optimize fuzzing of the target C file. These flags help us in tackling hard constraints like those involving magic-byte and equality comparisons. Sādhak runs AFL++ in the persistent mode.

5.1 Equivalence Class Optimizations

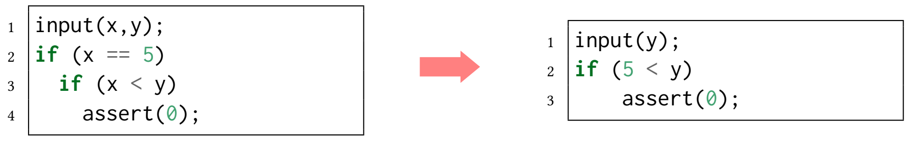
The performance of a fuzzer depends on the type and complexity of constraints presented to it. Let us discuss two classes of constraints that fuzzers generally find hard to solve and the optimizations used within Sādhak to handle them.

5.1.1 Equality Constraints. It is well-known that equality constraints are challenging for fuzzers. However, many equality constraints can be handled by identifying equivalence classes and applying a transformation that assigns all members of an equivalence class to a class representative. This technique is inspired by a similar mechanism used in the UF decision procedures in SMT solvers.

For example, let $\{x = y, y = z, w = z\}$ be a set of constraint at the fuzz engine's queue. One can identify two equivalence classes $\{x, y, z\}$ and $\{w\}$ based on the equality constraints. This allows a simple transformation as shown in Fig. 4 that chooses a representative from each class and replaces all uses of members of the class with the class representatives.

5.1.2 Magic-Byte Comparisons. Magic byte comparison are known to be even harder for the fuzzer to get through. A similar technique of identifying equivalence classes work well in such scenarios as well. This transformation is illustrated in Fig. 5.

5.1.3 Further Optimizations Possible. Similar to equivalence class optimizations for variables and magic-bytes, we could also consider forming equivalence classes for terms along with variables and magic-bytes. This can be a powerful optimization that can significantly simplify equality comparisons with terms. Sādhak currently misses this optimization and we plan to include it sometime soon in the future.



```
input(x,y);
if (x == 5)
  if (x < y)
    assert(0);
```

```
input(y);
if (5 < y)
    assert(0);
```

Fig. 5. Avoiding magic byte comparison in program to fuzz.

5.2 Fuzzer Seed Optimizations

As the terms in the fuzz engine only get augmented by additional constraints (i.e. constraints are never removed, see Sect. 4), the fuzzer can attempt to learn from its previous runs (on the same SMT file). We collect some inputs from the previous runs of the fuzzers that we add as *seeds* for subsequent runs. As the fuzzer uses these seeds to initiate exploration in a run, a good set of seed inputs can significantly bolster the performance of our fuzz engine.

There are two potential sources for such seed inputs:

- **Crash Inputs.** These are inputs that are encountered when the fuzzer is able to discover a failure in the program under inspection; in SĀDHAK, these correspond to the inputs that yield a satisfiable assignment to constraints presented to the fuzzer queue.
- **Interesting Inputs.** These are inputs that the fuzzer records whenever it discovers new coverage.

Our preliminary experiments demonstrated that all of the above were useful seed inputs for subsequent fuzzer invocations (on the same SMTLIB query). Hence, we seed our fuzzer runs with both crash and interesting inputs from previous runs. In our current implementation, we keep the seeds collected from previous iterations. Our hypothesis is that there is some correlation between the seeds across iterations that can help the fuzzer in subsequent rounds to quickly find a partial model (even though there may be change in the equivalence classes). We intend to investigate on this and better seed selection strategies in the future.

5.3 The Constraint Compiler

Our constraint compiler (see Sect. 4.3.1) generates C files to fuzz for satisfiable assignments. The generated C files (roughly) have a structure as shown in List. 10. The respective sections are instantiated by the translator while generating the C program using a user-provided translation schema $\mathcal{R}$ (see Sect. 3).

Translation Schema. We assume the translation schema to provide definitions to two interface functions:

- `TRANSLATE_SORT`: It provides a schema for translating sorts from the set of constraint to appropriate C language datatypes;
- `TRANSLATE_CONSTRAINT`: It provides a schema for translating terms from the set of constraints to appropriate C language expressions. The translation schema ($\mathcal{R}_\mathcal{T}$) is used to drive this translation for function symbols ($\mathcal{F}_\mathcal{T}$), predicates ($\mathcal{B}_\mathcal{T}$) and constants ($C_\mathcal{T}$) for each participating theory $\mathcal{T}$ in these constraints;

For example, below are some example translations for BitVector (BV) theory terms to expressions in the C programming language[2] below,

[2]to keep the exposition simple, we assume that the width of the BV variables is 8/32/64 bits for which standard datatypes in C are available; otherwise, additional "masking" operations may be required

```
(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(declare-const z (_ BitVec 32))


(declare-cb f
    ((_ BitVec 32) (_ BitVec 32))
    (_ BitVec 32))

(= z (f x y))
(bvugt x y)
(bvugt z (_ bv255 32))
(bvult z (_ bv65536 32))
```

(a) Constraints in bit-vector theory

```
uint32_t f (uint32_t, uint32_t);

int main() {
  uint32_t x,y,z;

  READ_INPUT(x,y,z);

  if (z == f(x, y))
    if (x > y)
      if (z > 255)
        if (z < 65536)
        {
          assert(0);
        }

  return 0;
}
```

(b) C program translation of List. 9a

List. 9. Constraint compilation for bit-vector terms.

- `TRANSLATE_SORT((declare-const x (_ BitVector 32)))`: translates to a variable declaration `uint32_t x` is C program;
- `TRANSLATE_CONSTRAINT((bvugt x y))`: predicate `bvugt` in BV theory is mapped to > symbol in the target language C and the expression is translated to `(x > y)`;
- `TRANSLATE_CONSTRAINT((bvadd x y))`: `bvadd` function symbol is mapped with + symbol and the expressions gets translated to `(x + y)`;
- `TRANSLATE_CONSTRAINT((= x (_ bv0 32)))`: constraint with constant term (_ bv0 32) gets translated to `(x == 0)`;
- `TRANSLATE_CONSTRAINT((= (foo x y) z))`: translates to `(foo(x, y) == z)`, where `foo` is a closed-box function.

Modus Operandi. When a set of constraints are presented to the queue of the fuzz engine, our constraint compiler operates as follows: to begin with, the variables and their sorts are extracted from the presented constraint set. Then, the equivalence classes (see Sect. 5.1) are computed and representative variables selected. The `TRANSLATE_SORT` interface function is used to instantiate the respective section in the template by declaring all the representative variables from each equivalence class. The `DECLARE_CB_FUNCTIONS` hole is filled with forward declarations of all closed-box functions symbols in $\mathcal{F}_{\mathcal{T}}^{cb}$.

Then, the constraints are rewritten in accordance to the representative variables, and the respective C program code is materialized via the `TRANSLATE_CONSTRAINT` interface function.

List. 9 shows an example of how the constraint compiler compiles a set of constraints to a C program. The colors in the figure map elements from the set of constraints (List. 9a) to the respective C program element (List. 9b). These constraints correspond to our motivating example, Eq. (1), at its fourth iteration (Tab. 4).

```
DECLARE_CB_FUNCTIONS

int main() {

    TRANSLATE_SORTS(V1, V2, V3, ...)

    READ_INPUTS(V1, V2, V3, ...)

    if ( TRANSLATE_CONSTRAINT(C1) )
      if ( TRANSLATE_CONSTRAINT(C2) )
        if ( TRANSLATE_CONSTRAINT(C3) )
        ⋮
          {
            assert(0);
          }

    return 0;
}
```

List. 10. Template file to be used by C translator.

5.4 Miscellaneous Details

Though Sect. 4 attempts to discover unsatisfiability within the pure SMT terms within the conflict-driven fuzz-only loop, we found that unsatisfiable benchmarks can be solved quickly by invoking the SMT engine on ω_{SMT} before entering the loop. Our implementation uses this setting by default. The benchmark suite used in evaluation does not include such instances that are unsatisfiable on pure SMT terms (i.e. ω_{SMT} is not unsatisfiable for any of our benchmarks) and so, SĀDHAK enters fuzz loop at least once.

Further, our preliminary experiments demonstrated that adding the closed-box function assignments from m_1 to ω_{SMT} via the CONVERTCBTOUF function (Alg. 1. line 15) has little impact but complicates the implementation. Hence, we turned this setting off in our implementation.

We implement the INDEPENDENT function (in Alg. 1 via incremental SMT solving. We store all lemmas available in the context of the SMT engine before asserting partial model from the fuzz engine. After asserting the partial model, the new lemmas inferred could now be dependent on the partial model. We only use the lemmas available in the stored context as the (safe underapproximation of) independent lemmas.

6 EVALUATION

Benchmarks. We evaluate SĀDHAK on a set of 107 benchmarks in the SMTLIB format. All these benchmarks include CB terms and hence, cannot be handled by SMT solvers. We used the following four sources for creating our benchmarks suite:

- We ran the symbolic execution engine KLEE [Cadar et al. 2008] on programs from [esb 2021], that contained external function calls to `cstdlib` functions. We collected the path constraints from KLEE from these executions. We link against the respective C libraries for definitions of these functions to be used as closed-box functions.

- We identified certain functions in SMTCOMP [smt 2015] benchmarks and removed their function definitions. We redefined them in C to serve as closed-box functions instead;
- We adapted some SyGuS-Comp benchmarks [syg 2019] as SMTLIB files with CB functions. A SyGuS-Comp benchmark consists of a set of constraints in the SMTLIB format and a grammar definition. We extracted the SMTLIB constraints from these files. We defined the function synthesized from the benchmark in C to be used as a closed-box function.
- Benchmarks from Delphi [Polgreen et al. 2022] have also been used for evaluating Sādhak.

All our benchmarks are available in the supplementary material. We intend to contribute them to SMTLIB [Barrett et al. 2010] in the future.

We ran all experiments on an Intel(R) Xeon(R) 2.00GHz E5-2620 CPU with 32GB RAM, running Ubuntu 16.04. We use a timeout of 600 seconds for all tools (Sādhak and the baseline tools) across all our experiments. We report the median of eleven runs.

Research Questions. We attempt to answer the following research questions:

- How do the the different modes of Sādhak compare?
- How does Sādhak compare with the current state-of-the-art tools?
- Are the optimizations in Sādhak effective?

We found that the CDFL mode of Sādhak that uses the conflict-driven fuzz loop (CDFL) algorithm to be the best mode of Sādhak; it solves 26.17% more benchmarks than the fuzz-only mode. We compare Sādhak with Delphi, the only other tool that is capable of handling CB constraints; Sādhak solves 36.45% more benchmarks than Delphi and has a PAR2 score that is 5.72× better. We found that the optimizations in Sādhak are effective, allowing us to solve 28% more instances and improving the PAR2 score by 4.71×.

6.1 Sādhak: Fuzz-Only v/s CDFL Modes

We implement two modes in Sādhak:

- an **CDFL mode** that uses conflict-driven fuzz loop (CDFL) to compute models (see Sect. 4), and
- a **fuzz-only mode** that does not employ the SMT engine; all the constraints are translated to a C program and the fuzzer is used to search for a satisfiable assignment (as discussed in Sect. 2.2).

Fig. 6 shows the performance of the two modes on all of our 107 benchmarks on a cactus plot. A point (x,y) on the plot signifies that solving x benchmarks took less than y seconds of time. The green colored plot in Fig. 6 shows the cactus plot of the CDFL mode and the red colored plot shows the cactus plot for the fuzz-only mode.

It can be seen in the plot, the CDFL mode significantly outperforms the fuzz-only mode. While the CDFL mode solves 101 of our 107 benchmarks, the fuzz-only mode solves only 73 instances. On the instances that get solved in both modes, on an average, CDFL is 20.5× faster.

From the evaluation we observed that, the fuzz-only mode struggles to solve benchmarks that have complex SMT constraints along with the CB functions. For example in Eq. (1), the constraints require a factorization of 64 into p, q, r (each being a power of two) – a fuzzer will find it difficult to solve. However, on a few instances, the fuzz-only mode performs better than CDFL. These instances had relatively small and simple SMT constraints that the fuzzer could handle well; the performance of the fuzz-only mode deteriorate as the number and complexity of the SMT constraints increase. Furthermore, the fuzz-only mode is limited as it can only be used when translation rules are available for all possible terms of the participating theories (see Sect. 3).

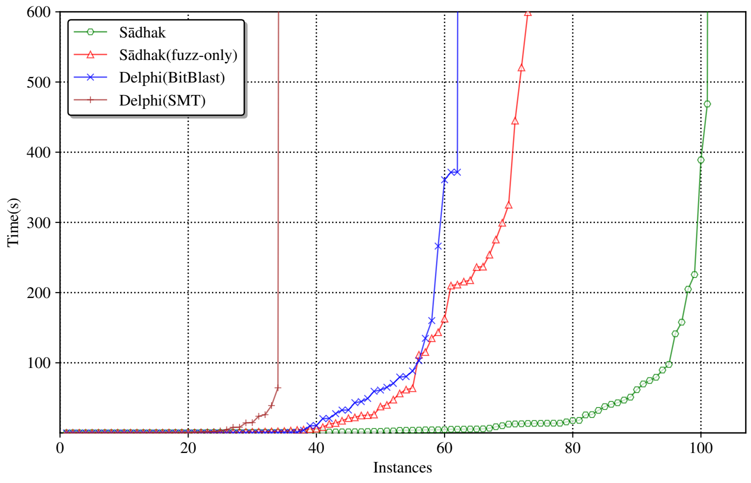


Fig. 6. Sādhak compared to Delphi

Table 7. Sādhak and Delphi performance summary. # solved represents the number of solved instances and PAR-2 represents the score we compute to compare two solvers. The best values are shown in bold fonts.

	Sādhak		Delphi	
	Fuzz mode	CDFL mode	SMT mode	BitBlast mode
# solved	73	**101**	34	62
PAR-2	484.70	**104**	924.36	595.51

The fuzz-only mode of Sādhak is inspired by a set of recent proposals [Borzacchiello et al. 2021; Liew et al. 2019; Pandey et al. 2019] that created such a solver to fuzz certain kinds of constraints like path conditions from symbolic execution engines, floating-point constraints and symbolic expressions. The fuzz-only mode of Sādhak now provides a mechanism to handle any arbitrary SMTLIB query.

For the rest of the evaluation, Sādhak refers to the CDFL mode and Sādhak(fuzz-only) refers to the fuzz-only mode.

6.2 Sādhak Versus State-of-the-Art Tools

Delphi [Polgreen et al. 2022] is the only other tool that can handle CB constraints, and hence we use it as our baseline (Delphi refers to CB operations as oracle functions). Delphi provides two modes: an SMT mode (that uses an SMT solver) and a bit-blasting mode (that bit-blasts the constraints to propositional logic and uses a SAT solver).

Fig. 6 shows a cactus plot of both the modes of Delphi. The blue colored plot shows the cactus plot for Delphi in bit-blast mode and brown plot shows the cactus plot for Delphi in SMT mode. As can be seen from the plot, the bit-blasting mode is comparatively faster than the SMT mode.

Sādhak, however, significantly outperforms all modes of Delphi on each of its supported modes. Even the fuzz-only mode of Sādhak is faster than both the modes in Delphi.

Table 7 presents a summary of the performance of both Sādhak and Delphi, across all their modes. We summarize performance via PAR-2 scores [Balyo et al. 2017], which is a common metric used for comparing SAT/SMT solvers. PAR-2 score is defined by adding running time of solved

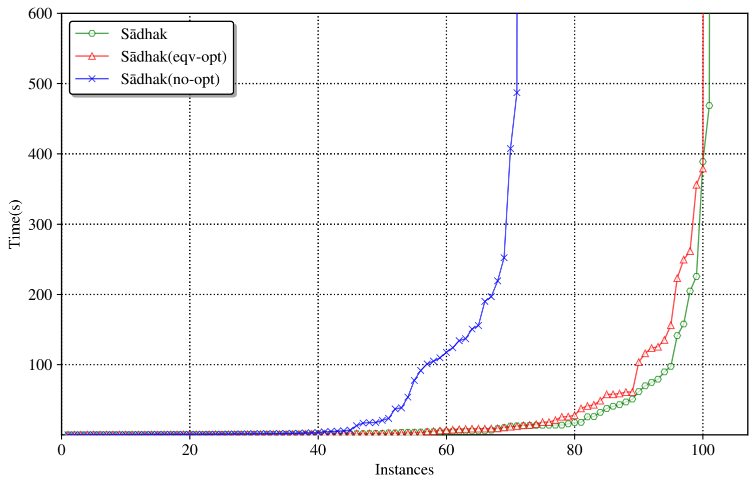


Fig. 7. SĀDHAK with equivalence class optimization from Sect. 5.1 and seed optimization from Sect. 5.2

instances and twice the timeout for each unsolved instances, then deviding the number with total number of benchmarks.

Overall, SĀDHAK solves 36.45% more benchmarks than the best performing mode of Delphi and has a 5.72× higher PAR-2 score. On benchmarks that were solved by both the tools (SĀDHAK and bit-blast mode of Delphi), on an average, SĀDHAK is 14.62× faster.

This experiment shows the value of combining an SMT solver and fuzzer in a CDFL scheme instead of using CB operations simply as an oracles.

6.3 Impact of Optimizations

6.3.1 Equivalence Class Optimizations. The performance of SĀDHAK improves significantly, SĀDHAK solving 29 (27.1%) more instances when the equivalence class optimizations are enabled. On the common set of benchmarks that are solved even with these optimizations disabled, on an average, SĀDHAK gets a average speedup of 26.34× when the optimization is turned on.

6.3.2 SĀDHAK with Fuzzer Seed Optimization. Both the set of crashing inputs and the set of interesting inputs from previous runs of the fuzzer can serve as viable seed input candidates for subsequent runs; we use both of these to seed the fuzzer.

6.3.3 Summary. Fig. 7 summarizes the improvements in the performance of SĀDHAK due to the different optimizations: The blue cactus plot shows the performance of SĀDHAK without any of the optimizations (equivalence class or seed optimizations); we denote it by SĀDHAK(no-opt). The red plot is the case where only the equivalence class optimizations are tuned on, denoted as SĀDHAK(eqv-opt). Finally, the green colored plot shows the performance of SĀDHAK with all the optimizations turned on (denoted as SĀDHAK).

7 RELATED WORK

Reasoning on programs that contain closed-box functions has been a challenging problem. As regular SMT solvers are incapable of handling such closed-box functions, many of the program analysis techniques like bounded model checkers and symbolic execution engines, attempted to invent their own strategies to combat this problem.

In the context of bounded model checking, *angelic verification* [Das et al. 2015; Das and Lal 2017; Joshi et al. 2012; Lahiri et al. 2020] emerged as an interesting proposal. In bounded model checking, the sound strategy is to overapproximate all closed-box functions; however, this is not practical as it explodes the number of false positives. Angelic verification uses a set of (mostly domain-specific) *underapproximation heuristics* to reduce the number of false positives. However, in the process the tool now loses soundness guarantees as the heuristics may suppress real bugs. Nevertheless, this technique has been demonstrated as effective in practice.

Symbolic execution [Cadar et al. 2008; Godefroid et al. 2005] uses *concretization* to handle closed-box artifacts: whenever a closed-box function is encountered, the symbolic arguments passed to the closed-box function are solved (in the context of the path condition) using an SMT solver for possible concrete arguments that are potential arguments of the invocation. The closed-box function is, then, concretely executed with these arguments and symbolic execution proceeds with the value returned from the invocation. Pandey et al. [2019] showed that a symbolic execution suffers significant loss in coverage, incurs path divergence and can even generate false positives. To combat this challenge, the authors proposed Colussus, a symbolic execution engine that used a fuzz solver—a fuzzer that attempted to solve path conditions, including the closed-box functions, in an attempt to discover concrete values that could cover the paths lost due to concretization; on path conditions that do not contain closed-box functions, an SMT solver is employed. Their fuzz solver is limited as it only handled constraints in the form of path conditions from symbolic execution engines, and solely uses fuzzing as the strategy to solve the constraints. FuzzySAT uses a similar scheme that proposes to completely abandon SMT solvers in lieu of a fuzzing driven solver to solve path conditions arriving out of symbolic execution. FuzzySat applies multiple optimizations to ensure that such constraints (from symbolic execution engines) can be effectively solved by fuzzers. In contrast to our proposal of building a general-purpose SMT solver, these techniques aimed to built specialized solvers for solving only path conditions and symbolic expressions from symbolic execution. Further, as these techniques completely eliminate SMT solvers and use only a fuzz-only approach with closed-box functions, they struggle when complex SMT constraints are encountered (at which SMT solvers are more adept).

Achar [Lahiri and Roy 2022] uses the fuzz-only approach for inferring inductive loop invariants for *almost verification* over closed-box components (referred to as *opaque* components in their work). Achar adopts a teacher-learner model where the teacher uses *proof fuzzing* to invalidate candidate invariants proposed by the learner. Any counterexamples are fed back to the learner for refining its proposals. Achar also allows for a *hybrid* scheme, where the set of paths in the program are partitioned; while a fuzzer is used for “almost" verification on paths that contain closed-box functions, an SMT-based formal verification is applied to the rest of the program.

Higher-order test generation [Godefroid 2011] models closed-box functions as uninterpreted functions and uses tests from the validity proofs of first-order logic formulas from the path conditions (rather than from satisfiability assignments). They pose their work more as a requirement specification for such saturation based solvers in the absence of efficient validity proof generators. Mechtaev et al. [2018] modelled closed-box functions within path conditions of symbolic execution runs as an existential second-order constraints. They, then, solve these second order constraints via syntax-guided synthesis [Alur et al. 2013]. However, resorting to an expensive program synthesis engine during symbolic execution raises questions on its scalability. Also, like any synthesis task, their engine must be supplied well-crafted grammars with primitives that well-captures the semantics of the closed-box functions. Most importantly, the synthesized specification is still an approximation. Instead of learning a program, Argyros et al. [2016] attempt to learn a symbolic automata for filters and string sanitizers by treating them as closed-box oracles.

There has also been interest in building more general purpose solvers, ableit to handle constraints of a certain kind. There have been some earlier proposals that attempt to handle non-linear constraints and/or closed-box functions via randomized search [Dinges and Agha 2014; Păsăreanu et al. 2011]; however, recently, fuzzers have emerged the de facto choice for randomized search through the program state space. [Liew et al. 2019] attempts to solve floating-point SMT constraints using fuzzing. They package multiple optimizations (e. g., the equivalence class optimization that is commonly used for UF theories, and that is also employed by Sādhak). They, too, use a fuzz-only approach and solely focus on the floating-point theory.

Delphi [Polgreen et al. 2022] is a state-of-the-art SMT solver that is capable of handling closed-box functions. Delphi uses the SMT driven active learning scheme (see Sect. 2) and solely uses an SMT solver—that is, no randomized search or fuzzing is employed for the closed-box constraints. Instead, the SMT solver attempts to verify solutions over the rest of the constraints with the closed-box functions; if a solution is consistent with the closed-box constraints, the solution is returned. Otherwise, the SMT solver learns from the failure and attempts to find another solution.

In contrast to all the above proposals, Sādhak is, to the best of our knowledge, the first solver that successfully combines an SMT solver and a fuzzer in a synergistic loop. Operating in a counterexample-driven fuzz loop (CDFL), Sādhak uses a fuzzer to search over the closed-box constraints, while an SMT solver reasons on the SMT theories. These two engines exchange a rich set of interface constraints, learning from each other's failures. Our experimental results demonstrates the strength our CDFL scheme.

With Sādhak to answer to satisfiability queries, program synthesis with closed-box components becomes an interesting research direction. Program synthesis has seen applications in multiple domains, like bitvectors [Gulwani et al. 2011; Solar-Lezama et al. 2005], heap manipulations [Garg and Roy 2015; Polikarpova and Sergey 2019; Roy 2013; Verma and Roy 2017; Verma and Subhajit Roy 2021], bug synthesis [Roy et al. 2018], differential privacy [Roy et al. 2021; Wang et al. 2021], Skolem functions [Golia et al. 2020, 2021a,b], synthesis of fences and atomic blocks [Verma et al. 2020], synthesizing parsers [Kalita et al. 2022; Leung et al. 2015; Singal et al. 2018] and even in hardware security [Takhar et al. 2022]. The above works assume that all program components available in synthesis have formally-defined semantics. Delphi [Polgreen et al. 2022] proposes a *synthesis modulo oracle* framework that allows synthesis over closed-box components. With satisfiability solvers and specialized algorithms to drive it, synthesis over closed-box components is a potentially fruitful direction for synthesis research.

Fuzzing has been used effectively in many general-purpose applications such as discovering performance bottlenecks [Lemieux et al. 2018], side-channel analysis [Nilizadeh et al. 2019], hardware testing [Laeufer et al. 2018; Muduli et al. 2020], etc., other than finding software vulnerabilities. The success of Sādhak is largely due to the recent improvements in fuzzing too. Sādhak uses AFL++ [Fioraldi et al. 2020], an improved variant of AFL [Zalewski 2019], to implement its fuzz engine. As we use this component off-the-self, any other popular fuzzer, like libFuzzer [Serebryany 2015] or Honggfuzz [Google Inc. 2020], can also be employed instead. At the same time, as all programs from our constraint compiler uses a well-defined structure, we believe that Sādhak can be significantly improved via a domain-specific fuzzer. Frameworks like FuzzFactory [Padhye et al. 2019] facilitate writing fuzzers with custom objective functions, we intend to pursue this direction to build a custom fuzzer for Sādhak in the future.

8 CONCLUSION

We tackle the problem of solving SMT queries with *closed-box functions* (also called *oracle* functions [Polgreen et al. 2022] or *opaque* components [Lahiri and Roy 2022]). Our algorithm, *conflict-driven fuzz learning*, places a fuzz engine and an SMT engine in a loop—each learning from each other's failures. We also create a benchmark suite of such SMT queries and evaluate an instantiation of our idea, SĀDHAK, on it. SĀDHAK solves 36.45% more benchmarks and achieves a PAR2 score that is 5.72× better than the current state of the art. SĀDHAK shows an average speed up of 14.62× compared to state-of-the-art solver (Delphi [Polgreen et al. 2022]) in its best mode, on benchmarks solved by both the tools.

Though we collected benchmarks from a variety of sources, there are some threats to validity. Experiments on benchmarks from more sources can be conducted. Further, like any other computational tasks, the experimental results are subject to variations with change of architecture and memory of the computing platforms. However, we believe that the trends in the results should continue to hold.

9 DATA-AVAILABILITY STATEMENT

A Docker image containing source code of SĀDHAK, our benchmark suite, and experimental scripts is available [Muduli and Roy 2022].

REFERENCES

2015. SMTCOMP SMTLib2 benchmarks. (2015). https://smtlib.cs.uiowa.edu/benchmarks.shtml

2019. Benchmarks for SyGuS Competition. *SyGuS-Comp* (2019). https://github.com/SyGuS-Org/benchmarks

2021. *ESBMC.* https://github.com/esbmc/esbmc/tree/master/regression

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

George Argyros, Ioannis Stais, Aggelos Kiayias, and Angelos D. Keromytis. 2016. Back in Black: Towards Formal, Black Box Analysis of Sanitizers and Filters. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 91–109.

Marijn J.H. Balyo, Tomáš Järvisalo, Matti, and Heule (Eds.). 2017. Proceedings of SAT Competition 2017 : Solver and Benchmark Descriptions. (2017). http://hdl.handle.net/10138/224324

Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, Vol. 13. 14.

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. https://doi.org/10.1109/ICSE43902.2021.00071

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

Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. 2015. Angelic Verification: Precise Verification Modulo Unknowns. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 324–342. https://doi.org/10.1007/978-3-319-21690-4_19

Ankush Das and Akash Lal. 2017. Precise Null Pointer Analysis Through Global Value Numbering. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10482)*. Springer, 25–41. https://doi.org/10.1007/978-3-319-68167-2_2

Peter Dinges and Gul Agha. 2014. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/2635868.2635889

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL (T): Fast decision procedures. In *International Conference on Computer Aided Verification*. Springer, 175–188. https://doi.org/10.1007/978-3-540-27813-9_14

Anshul Garg and Subhajit Roy. 2015. Synthesizing Heap Manipulations via Integer Linear Programming. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9291)*, Sandrine Blazy and Thomas P. Jensen (Eds.). Springer, 109–127. https://doi.org/10.1007/978-3-662-48288-9_7

Patrice Godefroid. 2011. Higher-Order Test Generation. *SIGPLAN Not.* 46, 6 (jun 2011), 258–269. https://doi.org/10.1145/1993316.1993529

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN PLDI Conference* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. 2020. Manthan: A Data-Driven Approach for Boolean Function Synthesis. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 611–633. https://doi.org/10.1007/978-3-030-53291-8_31

Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. 2021a. Program Synthesis as Dependency Quantified Formula Modulo Theory. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Zhi-Hua Zhou (Ed.). ijcai.org, 1894–1900. https://doi.org/10.24963/ijcai.2021/261

Priyanka Golia, Friedrich Slivovsky, Subhajit Roy, and Kuldeep S. Meel. 2021b. Engineering an Efficient Boolean Functional Synthesis Engine. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021*. IEEE, 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643583

Google Inc. 2020. *HonggFuzz: Security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based)*. https://honggfuzz.dev/

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. https://doi.org/10.1145/3318162

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. 2012. Underspecified harnesses and interleaved bugs. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 19–30. https://doi.org/10.1145/2103656.2103662

Pankaj Kumar Kalita, Miriyala Jeevan Kumar, and Subhajit Roy. 2022. Synthesis of Semantic Actions in Attribute Grammars. In *Formal Methods in Computer Aided Design (FMCAD '22)*. https://doi.org/10.34727/2021/isbn.978-3-85448-053-2_37

Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (San Diego, CA, USA). IEEE Press, 1–8. https://doi.org/10.1145/3240765.3240842

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* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 352–364. https://doi.org/10.1145/3533767.3534381

Shuvendu K. Lahiri, Akash Lal, Sridhar Gopinath, Alexander Nutz, Vladimir Levin, Rahul Kumar, Nate Deisinger, Jakob Lichtenberg, and Chetan Bansal. 2020. Angelic Checking within Static Driver Verifier: Towards high-precision defects without (modeling) cost. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 169–178. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_24

K Rustan M Leino. 2008. This is boogie 2. *manuscript KRML* 178, 131 (2008), 9.

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3213846.3213874

Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. *SIGPLAN Not.* 50, 6 (jun 2015), 565–574. https://doi.org/10.1145/2813885.2738002

Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. 2019. Just Fuzz It: Solving Floating-Point Constraints Using Coverage-Guided Fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 521–532. https://doi.org/10.1145/3338906.3338921

Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic execution with existential second-order constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 389–399. https://doi.org/10.1145/3236024.3236049

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701. https://doi.org/10.1145/2884781.2884807

Sujit Kumar Muduli and Subhajit Roy. 2022. *Satisfiability Modulo Fuzzing: A Synergistic Combination of SMT Solving and Fuzzing (Artifact)*. https://doi.org/10.5281/zenodo.7066264

Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. 2020. Hyperfuzzing for SoC Security Validation. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3400302.3415709

Greg Nelson and Derek C Oppen. 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 2 (1979), 245–257.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM* 53, 6 (nov 2006), 937–977. https://doi.org/10.1145/1217856.1217859

Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DifFuzz: Differential Fuzzing for Side-Channel Analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 176–187. https://doi.org/10.1109/ICSE.2019.00034

Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (oct 2019), 29 pages. https://doi.org/10.1145/3360600

Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. 2019. Deferred Concretization in Symbolic Execution via Fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. https://doi.org/10.1145/3293882.3330554

Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. 2022. Satisfiability and Synthesis Modulo Oracles. In *Verification, Model Checking, and Abstract Interpretation*. https://doi.org/10.1007/978-3-030-94583-1_13

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 72 (jan 2019), 30 pages. https://doi.org/10.1145/3290385

Corina S. Păsăreanu, Neha Rungta, and Willem Visser. 2011. Symbolic Execution with Mixed Concrete-Symbolic Solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 34–44. https://doi.org/10.1145/2001420.2001425

Subhajit Roy. 2013. From Concrete Examples to Heap Manipulating Programs. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7935)*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer, 126–149. https://doi.org/10.1007/978-3-642-38856-9_9

Subhajit Roy, Justin Hsu, and Aws Albarghouthi. 2021. Learning Differentially Private Mechanisms. In *2021 IEEE Symposium on Security and Privacy (SP)*. 852–865. https://doi.org/10.1109/SP40001.2021.00060

Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 224–234. https://doi.org/10.1145/3236024.3236084

K Serebryany. 2015. libFuzzer a library for coverage-guided fuzz testing. *LLVM project* (2015).

Dhruv Singal, Palak Agarwal, Saket Jhunjhunwala, and Subhajit Roy. 2018. Parse Condition: Symbolic Encoding of LL(1) Parsing. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing, Vol. 57)*, Gilles Barthe, Geoff Sutcliffe, and Margus Veanes (Eds.). EasyChair, 637–655. https://doi.org/10.29007/2ndp

Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by Sketching for Bit-Streaming Programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1065010.1065045

Gourav Takhar, Ramesh Karri, Christian Pilato, and Subhajit Roy. 2022. HOLL: Program Synthesis for Higher Order Logic Locking. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 3–24. https://doi.org/10.1007/978-3-030-99524-9_1

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Aakanksha Verma, Pankaj Kumar Kalita, Awanish Pandey, and Subhajit Roy. 2020. Interactive debugging of concurrent programs under relaxed memory models. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*. ACM, 68–80. https://doi.org/10.1145/3368826.3377910

Sahil Verma and Subhajit Roy. 2017. Synergistic debug-repair of heap manipulations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 163–173. https://doi.org/10.1145/3106237.3106263

Sahil Verma and Subhajit Roy. 2021. Debug-localize-repair: A symbiotic construction for heap manipulations. *Formal Methods Syst. Des.* 58, 3 (2021), 399–439. https://doi.org/10.1007/s10703-021-00387-z

Yuxin Wang, Zeyu Ding, Yingtai Xiao, Daniel Kifer, and Danfeng Zhang. 2021. DPGen: Automated Program Synthesis for Differential Privacy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 393–411. https://doi.org/10.1145/3460120.3484781

Michal Zalewski. 2019. *American Fuzzy Lop*. http://lcamtuf.coredump.cx/afl

Received 2022-04-15; accepted 2022-09-01