Validating Mixed-Integer Programming Solvers

Xintong Zhou
x27zhou@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Zhenyang Xu
zhenyang.xu@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Chengnian Sun
cnsun@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Abstract

Mixed-integer programming (MIP) is a fundamental class of mathematical optimization problems with broad applications in various domains such as finance, engineering, and management science. MIP solvers, software systems that automatically solve MIP problems, serve as the computational backbone for these applications. Given their widespread use, ensuring the correctness of MIP solvers is crucial, as incorrect results, such as falsely determining feasibility or returning incorrect solutions, can lead to serious real-world consequences. Despite its importance, validating the correctness of MIP solvers remains largely unexplored in both theory and practice.

This paper presents the first systematic effort to address this problem. We propose *feasibility-driven instance generation*, a simple yet effective technique to generate random MIP instances for testing solver correctness. The core idea is to systematically synthesize MIP instances that are provably feasible or infeasible by construction. These instances are then fed to MIP solvers to detect potential bugs. We realize this methodology in Flip. To date, Flip has uncovered 67 confirmed bugs in five widely used MIP solvers, spanning both open-source and commercial systems. Among these, 54 have been promptly fixed by the developers. Our efforts and findings have been well acknowledged by the MIP solver community.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Software Testing, Solver Validation, Mixed-Integer Programming

ACM Reference Format:
Xintong Zhou, Zhenyang Xu, and Chengnian Sun. 2026. Validating Mixed-Integer Programming Solvers . In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787827

1 Introduction

Mixed-integer programming (MIP), also known as mixed-integer linear programming (MILP), is a fundamental class of mathematical optimization problems in operations research [63]. Given a group of decision variables (with some or all constrained to integer values), a set of linear constraints, and a linear objective function, the goal of MIP is to seek a set of values for the decision variables, satisfying the constraints while optimizing the objective function. MIP extends the pure linear programming (LP) problem by allowing integrality constraints on decision variables, making it more expressive and capable of capturing the discrete nature of some decisions in real-world. For example, a variable representing the number of products to produce must be an integer. MIP problems are pervasive in real-world production, spanning economics [18, 60], engineering [20, 42], finance [10, 12], and management science [9, 39].

MIP solvers, software systems that automatically solve MIP problems, serve as the computational backbone for these applications. In the past few decades, a number of MIP solvers have been developed, including open-source solvers such as SCIP [4], HiGHS [31], and CBC [13], as well as commercial products like Gurobi [24] and Mosek [46]. The primary difference between these solvers lies in their underlying solving algorithms and heuristics, which are designed to solve MIP problems more efficiently, e.g., handling larger problems and finding solutions faster. However, the correctness of MIP solvers is not fully guaranteed. Just like other complex and highly optimized software systems, such as compilers [35, 69], operating systems [55, 64], and SMT solvers [43, 67], MIP solvers are not immune to bugs. According to the public bug reports of several major solvers, most issues are discovered directly by end users. Such bugs can lead to incorrect results and, in turn, serious real-world consequences [29, 30]. Given the widespread deployment of MIP solvers, validating their correctness is critical, yet remains largely underexplored in both theory and practice.

Challenges in MIP Solver Validation. The primary challenge in validating MIP solvers lies in generating diverse yet nontrivial MIP instances that can effectively exercise the solvers' complex behaviors. General-purpose fuzzing techniques such as AFL [72] and its variants [2, 3] primarily rely on low-level mutation strategies that are ill-suited for producing syntactically valid MIP instances [73], often leading to early rejection by parsers and rarely reaching the solvers' core optimization logic. Moreover, as shown in § 5.2, randomly generating syntactically valid MIP instances based solely on the standard MIP specification cannot ensure feasibility and mostly yields trivial cases. Several methods from the operations research community [5, 37], originally designed to facilitate algorithm design and experimentation, synthesize instances with specific mathematical properties by imposing additional constraints during generation. However, our evaluation shows that these approaches are also of limited effectiveness for solver validation.

Feasibility-Driven Instance Generation. This paper addresses the aforementioned challenges by proposing *feasibility-driven instance generation*, a novel and effective technique for generating random yet nontrivial MIP instances to support MIP solver validation. A correct MIP solver should (1) correctly determine the feasibility of a given MIP instance, and (2) return a valid optimal solution if the instance is feasible. To address both aspects, our

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787827

technique incorporates two generation algorithms: *feasible instance generation* and *infeasible instance generation*, which systematically generate *feasible* and *infeasible* MIP instances by construction, respectively. In general, our approach constructs a feasible instance by incrementally synthesizing constraints that are satisfied by a predefined solution. To generate an infeasible instance, a feasible instance is constructed first, then combined with a feasibility-breaking constraint to form an infeasible instance.

Flip. We realized the proposed technique in Flip, the first specialized fuzzing framework for MIP solvers. We extensively evaluated Flip by applying it to five widely used MIP solvers, namely, SCIP, HiGHS, CBC, Mosek, and Gurobi. To date, Flip has resulted in 80 bug reports, of which 67 have been confirmed and 54 have been fixed by the respective solver developers. Notably, the majority (35/67) of the confirmed bugs are soundness issues. We further show that both *feasible* and *infeasible* instances are necessary for extensive solver validation, as each exposes distinct classes of bugs. Our findings have been acknowledged and appreciated by developers across multiple solvers. In particular, the SCIP development team invited us to join their internal repository to report issues directly, underscoring Flip's practical value and real-world impact.

Contributions. This paper makes the following contributions:

- We propose *feasibility-driven instance generation*, a simple yet effective technique to generate diverse feasible and infeasible MIP instances for extensive solver validation.
- Based on the proposed technique, we design and engineer Flip, the first specialized fuzzing framework for MIP solvers.
- We stress test five popular MIP solvers with Flip to evaluate its effectiveness. To date, Flip has uncovered 67 confirmed bugs, with 54 having been fixed. We present extensive evaluations to demonstrate the effectiveness of Flip in terms of code coverage and bug finding.

We open-source Flip[1] to facilitate further research and practical adoption, and make the artifacts publicly available to support reproducibility [76].

2 Background

This section provides the necessary background on mixed-integer programming (MIP) problems and their solvers.

Mixed-Integer Programming. Mixed-integer programming (MIP) constitutes a fundamental class of optimization problems that aims to identify the optimal solution for a linear-constrained problem. MIP extends linear programming (LP) by allowing integrality constraints on some or all decision variables, which makes it more expressive in modeling the discrete nature of some decision variables in real-world problems. An MIP problem can be mathematically formulated as follows:

$$\begin{aligned} \min_{x,\,y} \quad & c^T x + d^T y \\ \text{subject to} \quad & Ax + By \le b, \\ & x \in \mathbb{R}^n,\ y \in \mathbb{Z}^m, \end{aligned}$$

where x and y are the continuous and integer decision variables, respectively; c and d are coefficient vectors for the objective function; A and B define the constraint matrices; and b is the right-hand side vector. Solving an MIP problem involves finding values of x and y that satisfy all constraints and minimize the objective function. Note that minimization and maximization are mathematically equivalent and can be transformed into one another by negating the objective function. An MIP instance is considered *feasible* if there exists at least one solution that satisfies all constraints, and *infeasible* otherwise. For example, Figure 1a shows a feasible instance, where the assignment $x_0 = -12$ and $x_1 = 64$ satisfies all constraints from `C1` to `C6`. On the contrary, the instance in Figure 1c is infeasible, as no assignment of x_0 and x_1 can satisfy all constraints. For a feasible instance, a solution is termed *optimal* if it makes the objective function attain its minimum value among all feasible solutions. For Figure 1a, the optimal solution is $x_0 = -13.41$ and $x_1 = 65.0$, enabling the objective function to attain its minimum value of 596.02.

MIP Solvers. Solving MIP problems is generally more challenging than solving LP problems due to the additional combinatorial complexity introduced by integer constraints. A common strategy involves first solving the relaxed LP version of the problem and then searching for an integer-feasible solution near the relaxed optimum. Modern MIP solvers typically incorporate an internal LP solver based on classical algorithms such as the simplex method [14, 48] and the interior-point method [32, 45]. To handle integer constraints, advanced algorithms such as branch-and-bound [34] and cutting-plane methods [33] are employed. Widely used MIP solvers include open-source software such as SCIP [4], HiGHS [31], and CBC [13], as well as commercial products like Gurobi [24] and Mosek [46]. Given their crucial role and widespread use in real-world applications, validating the correctness of MIP solvers is of paramount importance.

A correct MIP solver is expected to achieve the following: (1) correctly determine the feasibility of an instance, (2) find the optimal solution for feasible instances. To effectively validate the correctness of MIP solvers, the key challenge lies in generating non-trivial MIP instances that cover both feasible and infeasible cases and examine the solvers' intrinsic solving logic. As the evaluation results shown in § 5.2, randomly synthesizing such instances often results in trivial cases, leading to limited effectiveness of validation. In this context, we introduce *feasibility-driven instance generation*, a novel technique for generating MIP instances with guaranteed feasibility or infeasibility by construction, which serves as a solid foundation for effective MIP solver validation.

3 Illustrative Examples

Our *feasibility-driven instance generation* technique incorporates two generation algorithms, i.e., *feasible* instance generation and *infeasible* instance generation. This section provides two concrete examples, each illustrating one of the algorithms.

3.1 Feasible Instance Generation

Our general idea for constructing a feasible MIP instance is to incrementally synthesize constraints that are satisfied by a predefined solution. We begin by initializing an MIP instance with a set of decision variables. For example, the instance shown in Figure 1a includes two continuous variables: `x0` and `x1`. At this stage, the

[1] https://github.com/zxt5/Flip

```
Minimize
OBJ: 55.59 x0 + 20.64 x1
Subject To
 C1: -33.55 x0 + 55.79 x1 ≤ 4326
 C2: -49.31 x0 + 39.55 x1 ≤ 3259
 C3: -76.66 x0 - 42.31 x1 ≥ -3572
 C4: 79.08 x0 + 58.17 x1 ≥ 2720.4
 C5: 62.57 x0 - 74.56 x1 ≤ -343.6
 C6: 82.76 x0 - 40.36 x1 ≤ 855.48
Bounds
 -100 ≤ x0 ≤ 200
 -100 ≤ x1 ≤ 200
End
```

(a) A feasible MIP instance generated by Flip that exposes a soundness bug in Gurobi [49].

```
Minimize
OBJ: 24.33 x0 + 79.04 x1
Subject To
 C1: -38.46 x0 + 5.74 x1 ≥ -284.26
 C2: -28.05 x0 - 95.78 x1 ≤ -1798.1
 C3: -62.31 x0 - 42.41 x1 ≥ 2293.48
 C4: 71.52 x0 + 41.52 x1 ≤ 525.7
 C5: -45.3 x0 - 43.73 x1 ≤ 3067.13
Bounds
 -200 ≤ x0 ≤ 200
 -200 ≤ x1 ≤ 200
Generals
 x1
End
```

(b) A randomly generated feasible seed MIP instance for infeasible instance generation.

```
Minimize
OBJ: -24.15 x0 + 53.66 x1
Subject To
 C1: -38.46 x0 + 5.74 x1 ≥ -284.26
 C2: -28.05 x0 - 95.78 x1 ≤ -1798.1
 C3: -62.31 x0 - 42.41 x1 ≥ 2293.48
 C4: 71.52 x0 + 41.52 x1 ≤ 525.7
 C5: -45.3 x0 - 43.73 x1 ≤ 3067.13
 C6: 24.33 x0 + 79.04 x1 ≤ 1343
Bounds
 -200 ≤ x0 ≤ 200
 -200 ≤ x1 ≤ 200
Generals
 x1
End
```

(c) An infeasible MIP instance generated by Flip that triggers a soundness bug in SCIP [56].

Figure 1: Illustrative examples of our feasibility-driven instance generation approaches.

instance contains no constraints, i.e., it is unconstrained, and therefore any assignment to the decision variables is trivially feasible. We then randomly select a solution for this empty instance, e.g., $x0 = -12.0$ and $x1 = 64.0$ in this case. Next, we incrementally add constraints to the instance while ensuring that the chosen solution satisfies them. For instance, constraint `C1: -33.55 x0 + 55.79 x1 ≤ 4326` is explicitly constructed to be satisfied by the selected solution, i.e., $-33.55 \times (-12.0) + 55.79 \times (64.0) \leq 4326$. By repeating this process, we can introduce an arbitrary number of constraints while preserving feasibility. In this example, six constraints are injected to the instance. Then, we add bounds on the decision variables to further constrain the solution space, ensuring that a finite optimal solution exists. The bounds should include the chosen solution to maintain feasibility. In this case, we set the bounds of both $x0$ and $x1$ to $[-100, 200]$. In the last step, we synthesize an objective function for the MIP instance, e.g., `55.59 x0 + 20.64 x1`, which is synthesized randomly to exercise the solver's ability to identify the optimal solution. The resulting instance, shown in Figure 1a, reveals a soundness issue in Gurobi, where the solver incorrectly reports the instance as infeasible. We reported this bug to the Gurobi team, who confirmed the issue and fixed it promptly in a subsequent release.

3.2 Infeasible Instance Generation

For an infeasible instance, a solver is expected to correctly report infeasibility. If it instead classifies the instance as feasible and returns an invalid solution, a soundness bug is exposed. Our approach systematically constructs such infeasible instances to uncover these issues. In general, we generate an infeasible MIP instance by transforming a feasible model into an infeasible one. This begins with constructing a feasible instance using the method described in § 3.1, which serves as the base for infeasible instance generation. For example, suppose we obtain the feasible instance shown in Figure 1b. We then solve it using a reference solver to obtain its optimal objective value. In this example, the minimum objective value is 1343.12, achieved with the optimal solution $x_0 = -119$, $x_1 = 53.62$. Assuming the reference solver is correct (otherwise, it would indicate a bug in the solver itself), no variable assignment should yield a value smaller than 1343.12 for the objective function `24.33 x0 + 79.04 x1`.

Leveraging this insight, we introduce a constraint that deliberately renders the instance infeasible. Specifically, we inject the constraint `C6: 24.33 x0 + 79.04 x1 ≤ 1343`. To avoid issues related to floating-point precision, we set the right-hand side value of this constraint to a value slightly below the known minimum objective, which is 1343 in this case. This ensures that the new constraint invalidates all previously feasible solutions, thereby guaranteeing that the resulting instance is infeasible by construction. We then synthesize a new, arbitrary objective function for the infeasible instance, e.g., `-24.15 x0 + 53.66 x1`, which is not tied to any optimization goal but simply used to complete the instance structure. The resulting instance, shown in Figure 1c, uncovers a soundness bug in SCIP, causing the solver to return an invalid solution that violates the constraint `C5`. We reported this issue to the developers of SCIP, who promptly confirmed and fixed the bug [56]. The root cause of this issue lies in the redundancy pair detection mechanism within the variable bound constraint handler, which mistakenly removes the non-redundant constraint `C5`.

4 Approach

This section formally presents our feasibility-driven approach to generating feasible and infeasible MIP instances, and how we apply it to validate MIP solvers. We introduce Flip, a practical framework that realizes our generation techniques for fuzz testing MIP solvers.

4.1 Definitions

We introduce a set of definitions to facilitate presentation. We define a *mixed-integer programming (MIP) instance* $\mathcal{P}$ as a tuple:

$$\mathcal{P} := \langle \mathcal{X}, \mathcal{B}, C, f \rangle,$$

which consists of the following components:

- $\mathcal{X} = \{x_1, \cdots, x_n\}$: A set of *decision variables*, where some are designated as *integer* variables, denoted by the subset $\mathcal{I} \subseteq \mathcal{X}$. Formally, $\forall x_i \in \mathcal{X} \bullet x_i \in \mathbb{Z}$ if $x_i \in \mathcal{I}$, and $x_i \in \mathbb{R}$ otherwise.
- $\mathcal{B}$: A set of *bounds* on the decision variables. For each variable $x_i \in \mathcal{X}$, its bound $b_i = \{l_i, u_i\} \in \mathcal{B}$ restricts its value to the interval $[l_i, u_i]$.
- C: A set of *linear constraints* over the decision variables. Each constraint $c_i \in C$ is of the form:

$$c_i \coloneqq a_i^\top \mathcal{X} \ \Delta_i \ b_i,$$

  where $a_i \in \mathbb{R}^n$ is a vector of coefficients, $b_i \in \mathbb{R}$ is a scalar, and $\Delta_i \in \{\leq, \geq\}$ is a relational operator. We denote by $\overline{\Delta_i}$ the *complementary relation* of Δ_i; that is, if $\Delta_i = \leq$, then $\overline{\Delta_i} = \geq$, and vice versa.
- $f : \mathbb{R}^n \to \mathbb{R}$: A *linear objective function* to be optimized, defined as:

$$f(\mathcal{X}) \coloneqq d^\top \mathcal{X},$$

  where $d \in \mathbb{R}^n$ is a vector of coefficients.

Additionally, we denote by $\mathcal{Z}$ the set of all possible solutions to the MIP instance $\mathcal{P}$, which is defined as:

$$\mathcal{Z} \coloneqq \{z : \mathcal{X} \to \mathbb{R}^n\},$$

where a *solution* $z : \mathcal{X} \to \mathbb{R}^n$ is a mapping that assigns values to all decision variables such that all constraints are satisfied:

$$\forall c_i \in C \bullet a_i^\top z \ \Delta_i \ b_i \models \text{True}.$$

An MIP instance $\mathcal{P}$ is said to be *feasible* if there exists at least one such solution, i.e., $\mathcal{Z} \neq \emptyset$; otherwise, it is *infeasible*. We denote feasible and infeasible instances by $\mathcal{P}_F$ and $\mathcal{P}_I$, respectively.

The *objective value* o of a solution z is the evaluation of the objective function at that point:

$$o = f(z).$$

The *optimal solution* z^* is a feasible solution that minimizes the objective function:

$$z^* = \operatorname*{argmin}_z f(z).$$

The corresponding *optimal objective value* is denoted by $o^* = f(z^*)$.

4.2 Feasible Instance Generation

The process for generating a feasible MIP instance is outlined in Algorithm 1. We begin by randomly determining the number of decision variables n (line 2) and the number of constraints m (line 3) according to user-specified hyperparameters: `MIN_VARS`, `MAX_VARS`, `MIN_CONS`, and `MAX_CONS`. These hyperparameters control the structural complexity of the generated instances, facilitating the approach to be tailored to different solvers and testing environments.

Next, we initialize a set of n decision variables $\mathcal{X}$ (line 4), where each variable is randomly designated as either continuous or integer. We then generate a random assignment z of values to these variables (line 5). Since no constraints exist at this stage, z is trivially feasible for the instance. To restrict the domain of each decision variable to a bounded range, we invoke `addBounds`$(\mathcal{X}, z)$ (line 6), which generates a set of bounds $\mathcal{B}$ for the decision variables. For each variable $x_i \in \mathcal{X}$, the corresponding bounds $b_i = \{l_i, u_i\}$ are chosen such that $x_i \in [l_i, u_i]$ and $l_i \leq z_i \leq u_i$, ensuring the initial solution z lies within the feasible region.

Algorithm 1: Feasible Instance Generation

```
1  Function genFeasibleMIP():
2      n ← rand(MIN_VARS, MAX_VARS)
3      m ← rand(MIN_CONS, MAX_CONS)
4      X ← initVars(n)                    // set decision variables
5      z ← initSolution(X)
6      B ← addBounds(X, z)                // add bounds to variables
7      C ← ∅
8      for i = 1 to m do
9          c_i ← genConstraint(X)
10         if ¬satisfy(z, c_i) then
11             c_i ← reverse(c_i)
12         C ← C ∪ {c_i}                  // add constraints
13     f ← genObjFunc(X)                  // generate objective function
14     return P_F : ⟨X, B, C, f⟩
```

We then construct the constraint set C by iteratively generating m feasibility-preserving constraints (line 7-12). In each iteration, the function `genConstraint()` (line 9) synthesizes a linear constraint of the form $c_i \coloneqq a_i^\top \mathcal{X} \ \Delta_i \ b_i$. We then check whether the current solution z satisfies the constraint (line 10), i.e., whether $a_i^\top z \ \Delta_i \ b_i \models \mathit{True}$. If it does not, we reverse the direction of the constraint by replacing Δ_i with logical complement $\overline{\Delta_i}$ (line 11) before adding it to C (line 12). This guarantees that all constraints in C are satisfied by z, thereby ensuring the feasibility of the generated MIP instance.

Finally, we invoke `genObjFunc()` to synthesize a random linear objective function f over the decision variables (line 13). This objective is used to evaluate the solution's ability to identify optimal solutions within the feasible region. The complete feasible MIP instance is then returned as $\mathcal{P}_F \coloneqq \langle \mathcal{X}, \mathcal{B}, C, f \rangle$.

Lemma 4.1. *The generated instance $\mathcal{P}_F$ is bounded feasible, guaranteeing the existence of an finite optimal solution within its feasible region.*

Proof. We first establish that $\mathcal{P}_F$ is feasible by construction. Specifically, there exists a pre-selected solution z that satisfies all the constraints in C and respects the variable bounds in $\mathcal{B}$. Formally:

$$\exists z \in \mathcal{Z} \bullet \ \forall c_i \coloneqq \left(a_i^\top \mathcal{X} \ \Delta_i \ b_i\right) \in C \bullet a_i^\top z \ \Delta_i \ b_i \models \mathit{True},$$
$$\forall z_i \in z, \ \{l_i, u_i\} \in \mathcal{B} \bullet l_i \leq z_i \leq u_i.$$

This ensures that the feasible region of $\mathcal{P}_F$ is non-empty. Next, we show that the objective function f is bounded over the feasible region. Since each variable $x_i \in \mathcal{X}$ is explicitly bounded by finite lower and upper bounds l_i and u_i and f is a linear function, its value over the feasible region is also bounded. Therefore, $\mathcal{P}_F$ is *bounded feasible*. By the Fundamental Theorem of Linear Programming [63], any MIP instance with a non-empty bounded feasible region attains its optimal value at some point within the region.

It is worth noting that while the pre-selected solution z is guaranteed to be feasible, it is not necessarily optimal and thus cannot serve as a correctness oracle. To address this, we employ differential testing as described in § 4.4.

4.3 Infeasible Instance Generation

The process for generating an infeasible MIP instance is presented in Algorithm 2. We begin by constructing a feasible instance $\mathcal{P}_F := \langle \mathcal{X}, \mathcal{B}, C, f \rangle$ as the base (line 2), using the method described in § 4.2. To break its feasibility, we introduce an additional constraint that invalidates all feasible solutions. To achieve this, we first solve $\mathcal{P}_F$ using a reference solver $\mathbf{s}^\dagger$ (line 3), obtaining the optimal (i.e., minimum) objective value o^*. This result implies that no feasible solution should yield an objective value smaller than o^* (otherwise, it would indicate a bug in the reference solver):

$$\nexists\, z \in \mathcal{Z} \bullet f(z) < o^*.$$

Leveraging this knowledge, we synthesize a *feasibility-breaking constraint* $\tilde{c}$ (line 4) of the form:

$$d^\top \mathcal{X} \leq o^* - \varepsilon,$$

where $\varepsilon > 0$ is a small positive constant introduced to ensure strictness and mitigate floating-point precision issues. This constraint $\tilde{c}$ is then added to the original constraint set C, resulting an updated set C' (line 5). Finally, we synthesize a new objective function f' (line 6), and return the resulting infeasible instance as $\mathcal{P}_I := \langle \mathcal{X}, \mathcal{B}, C', f' \rangle$.

Algorithm 2: Infeasible Instance Generation

```
1 Function genInfeasibleMIP(s†):
2     P_F : ⟨X, B, C, f⟩ ← genFeasibleMIP()
3     o* ← solve(s†, P_F)
4     c̃ ← f ≤ o* − ε                    // feasibility breaking constraint
5     C' ← C ∪ {c̃}
6     f' ← genObjFunc(X)                // new objective function
7     return P_I : ⟨X, B, C', f'⟩
```

Lemma 4.2. *Assuming that the reference solver $\mathbf{s}^\dagger$ returns the correct optimal objective value o^* for $\mathcal{P}_F$, the generated instance $\mathcal{P}_I$ is guaranteed to be infeasible.*

Proof. Assume, for contradiction, that there exists a valid solution z to $\mathcal{P}_I$. By construction, $\mathcal{P}_I$ includes all constraints from $\mathcal{P}_F$, along with an additional constraint $\tilde{c}$ defined as:

$$f(z) = d^\top z \leq o^* - \varepsilon.$$

Since z satisfies all constraints in C (inherited from $\mathcal{P}_F$), it must also be a feasible solution to $\mathcal{P}_F$. Therefore, by the definition of o^*,

$$f(z) \geq o^*.$$

Combining the two inequalities yields a contradiction:

$$o^* - \varepsilon \geq f(z) \geq o^*.$$

This is impossible for any $\varepsilon > 0$. Hence, no such solution z exists, and $\mathcal{P}_I$ is infeasible.

This construction ensures that $\mathcal{P}_I$ is a valid test case for verifying solver correctness: if a solver incorrectly reports $\mathcal{P}_I$ as feasible, it indicates a soundness bug. In practice, however, the reference solver $\mathbf{s}^\dagger$ may itself be flawed, potentially leading to the generation of an $\mathcal{P}_I$ that is actually feasible. *Nevertheless, this does not invalidate our approach.* Since our ultimate goal is to detect bugs in MIP solvers, exposing flaws in the reference solver is equally valuable. In this sense, $\mathcal{P}_I$ remains useful—whether it reveals issues in the target solver or in the reference solver, it contributes to strengthening solver reliability.

4.4 Flip

To leverage our generation techniques for validating MIP solvers, we have designed and engineered Flip. To the best of our knowledge, Flip is the first fuzzing framework specifically tailored for MIP solvers.

Algorithm. The core procedure of Flip is illustrated in Algorithm 3. Flip takes as input a set of MIP solvers under test $\mathcal{S}$, and a reference solver $\mathbf{s}^\dagger$ (which practically can be selected from $\mathcal{S}$). The outputs are two sets `crashes` and `incorrects`, storing the collected crashes and soundness bugs found during testing. Both sets are initialized to empty (line 1). Flip operates in a loop until a termination condition is met, e.g., by a timeout or user intervention. In each iteration, it randomly decides whether to test solvers using a feasible or an infeasible instance via `flipCoin()`. For feasible instances (line 4-11), Flip first invokes `genFeasibleMIP()` to generate a new feasible MIP instance $\mathcal{P}_F$. It then iterates over each solver $\mathbf{s_i}$ in $\mathcal{S}$ and attempts to solve $\mathcal{P}_F$ using $\mathbf{s_i}$. If a solver crashes, $\mathcal{P}_F$ is added to the `crashes` set. Otherwise, the returned objective value o_i^* is collected. If solvers return inconsistent objective values, this is considered a soundness bug, and $\mathcal{P}_F$ is added to the set `incorrects`. The process for testing with infeasible instances (line 13-18) follows a similar workflow. Flip first generates an infeasible instance $\mathcal{P}_I$ using `genInfeasibleMIP(`$\mathbf{s}^\dagger$`)`, and then attempts to solve it using all solvers under tests. As with feasible instances, crashes are recorded in the `crashes` set. A soundness bug is identified and recorded in the `incorrects` set if any solver incorrectly returns a solution that indicates feasibility for $\mathcal{P}_I$, despite the instance being provably infeasible.

Implementation of Flip. We have implemented Flip as a practical fuzzing framework for testing MIP solvers. Flip is built on top of PuLP [54], a Python library for modeling and solving (via invoking external solvers) linear and mixed-integer programming problems. Flip provides a variety of command-line options that allow users to customize the testing process. It accepts both solver binaries and Python-based solver interfaces as test targets, making it compatible with most LP/MIP solvers available in the wild. Moreover, Flip is designed for extensibility: new solvers can be supported by implementing a lightweight interface, which is primarily responsible for selecting solver-specific parameters and invoking the solver. Notably, Flip supports running fuzzing in a multi-threaded mode, which significantly increases the throughput and scalability of the testing process.

5 Evaluation

This section presents our extensive evaluation of Flip, demonstrating the effectiveness of feasibility-driven instance generation in validating real-world MIP solvers.

5.1 Evaluation Setup

Hardware Setup. During the evaluation, Flip has been executed on two Ubuntu 22.04 (64-bit) servers. The first machine is equipped

Algorithm 3: Flip's main process for MIP solver validation.

Input: $\mathbf{s}^{\dagger}$: reference solver for infeasible model generation.
Input: $\mathcal{S}$: MIP solvers under test.
Output: crashes and incorrects found.

```
1  crashes ← ∅, incorrects ← ∅
2  while ¬ shouldTerminate() do
3      if flipCoin() then
           /* Differential testing with feasible MIP instances. */
4          P_F ← genFeasibleMIP()
5          foreach s_i ∈ S do
6              if solve(s_i, P_F) = crash then
7                  crashes ← crashes ∪ {P_F}
8                  continue
9              o*_i ← solve(s_i, P_F)
10         if ¬(o*_1 = o*_2 = ··· = o*_|S|) then
11             incorrects ← incorrects ∪ {P_F}
12     else
           /* Testing with infeasible MIP instances. */
13         P_I ← genInfeasibleMIP(s†)
14         foreach s_i ∈ S do
15             if solve(s_i, P_I) = crash then
16                 crashes ← crashes ∪ {P_I}
17             else if solve(s_i, P_I) = feasible then
18                 incorrects ← incorrects ∪ {P_I}
19 return crashes, incorrects
```

with two Intel Xeon Gold 6348 CPUs, offering 56 physical cores and 112 hardware threads, along with 512 GB of RAM. The second machine features an AMD Ryzen 9 7950X processor with 16 physical cores and 32 threads, and 64GB RAM.

Solvers Under Test. We select five MIP solvers for our evaluation, including three open-source solvers, i.e., SCIP [4], HiGHS [31], and CBC [13], and two commercial solvers, i.e., Gurobi [24] and Mosek [46]. The selection is based on the following considerations: (1) these solvers are *mature and widely used* in both academic and industrial applications, making their quality particularly critical; (2) they are *actively maintained*, with responsive developer communities, that provide timely feedback on reported issues; and (3) they employ *diverse solving techniques*, such as the simplex method, interior-point method, and branch-and-cut, and offer rich command-line options to configure these methods, enabling us to evaluate the effectiveness of our approach across different solving strategies. While the fuzzing campaign is conducted across all five solvers, some subsequent evaluations, such as code coverage analysis, are restricted to the three open-source solvers (HiGHS, SCIP, and CBC), due to the lack of source code access for the commercial solvers. For the open-source solvers, we build them from source with assertions enabled. For the commercial solvers, we use the latest release versions for testing.

Numerical Precision Issues. Due to the prevalence of floating-point arithmetic in MIP solving, handling numerical precision issues becomes both crucial and challenging. A common strategy adopted by modern MIP solvers is to define a set of *tolerance thresholds* to determine solution feasibility and optimality [25, 47]. To ensure the comparability of testing results, it is essential to enforce consistent tolerance thresholds across all solvers; otherwise, discrepancies in solver outputs may stem from differing tolerance settings rather than actual bugs. In our implementation, Flip enforces the following tolerance thresholds uniformly across all solvers under test:

- *Feasibility tolerance* = 10^{-9}: All constraints must be satisfied within this margin. We choose 10^{-9} as it is the strictest setting supported by all solvers under test.
- *Relative & absolute optimality gap* = 0: The solver terminates with an optimal solution only when the relative or absolute gap between the best known bound and the incumbent solution is within this margin. We set both the relative and absolute gaps to zero to enforce the solvers to return exact optimal solutions.

Our experience indicates that these settings are effective in suppressing numerical discrepancies, thereby reducing false positives and minimizing unnecessary manual inspection.

Test Case Reduction. Test case reduction facilitates bug analysis by minimizing the size of test cases while preserving the essential characteristics that trigger the bugs. During our evaluation, we leverage Perses [62], a syntax-guided reduction tool, to minimize bug-triggering instances. Specifically, we extend Perses to support the MPS [65] format (the standard language for modeling LP/MIP problems), by providing it with a formal grammar for MPS files. With this extension, Perses is able to remove unnecessary elements from the instances, e.g., redundant constraints and variables, and returns a minimal instance that still triggers the bug.

Table 1: Count and status of bugs found by Flip.

Status	SCIP	HiGHS	CBC	Mosek	Gurobi	Total
Reported	27	21	21	7	4	80
Confirmed	23	18	19	5	2	67
Fixed	17	14	19	3	1	54
Duplicate	3	2	0	0	0	5
Won't fix	1	1	1	0	0	3

Research Questions. we organize our evaluation around the following research questions:

- **RQ1:** How many and what types of bugs can Flip find?
- **RQ2:** How much code coverage can Flip achieve?
- **RQ3:** How does the feasibility-driven instance generation strategy enhance bug detection?

5.2 Evaluation Results

This section presents the evaluation results by answering the proposed research questions.

RQ1: How many and what types of bugs can Flip find?

To validate the effectiveness of Flip, we applied it to stress test the five MIP solvers. During the evaluation period, Flip generates comparable numbers of feasible and infeasible MIP instances. We

use Gurobi as the reference solver for infeasible instance generation during the evaluation.

Bug Count. Table 1 summarizes the count of previously unknown bugs discovered by Flip in each solver, categorized by reporting status: reported, confirmed, fixed, marked as duplicate, and won't fix. In total, Flip led to 80 bug reports, of which 67 were confirmed, and 54 have been fixed by the developers. A few bug reports were marked as "won't fix" due to rare trigger conditions or issues in inactive third-party libraries. The majority of bugs reported by Flip were promptly confirmed and fixed by the developers, demonstrating their importance and practical relevance. At the time of writing, open source solvers show a higher fix rate (50/60) compared to commercial ones (4/7). This is because commercial solvers typically release bug fixes alongside new versions, which may depend on multiple factors and therefore cannot be delivered as promptly as those in open-source solvers. Examining the distribution of bugs across solvers, more bugs were identified in open-source solvers than in commercial ones, suggesting that commercial solvers may undergo more extensive internal testing prior to release and therefore achieve higher reliability. Nevertheless, Flip is still able to uncover previously unknown bugs in these solvers, demonstrating its effectiveness.

Bug Type. Flip captures two types of bugs during testing:

- *Crash bugs*: An instance exposes a crash bug if the solver terminates unexpectedly while solving it, e.g., due to an assertion failure, memory error, or internal solver error.
- *Soundness bugs*: An instance triggers a soundness bug if the solver returns an incorrect result, including: (1) reporting feasible and returning an invalid solution for an infeasible instance, or (2) reporting infeasible or failing to find the optimal solution for a feasible instance (e.g., returning an invalid or suboptimal solution).

We do not consider performance issues in this evaluation, leaving them as future work. Compared to crash and soundness bugs, performance regressions are generally more difficult to quantify and are less critical from a correctness standpoint. Table 2 summarizes the confirmed bugs found by Flip, categorized by type. Overall, Flip discovered a comparable number of crash bugs (32/67) and soundness bugs (35/67). This pattern is consistent across the three open-source solvers. In contrast, the commercial solvers only confirmed soundness bugs. This is typically because the commercial solvers were tested using release builds, which normally have assertions disabled. These results are encouraging, as they demonstrate that Flip is effective in detecting both types of bugs, particularly *soundness bugs*, which are often more subtle yet more critical than crashes. Unlike crashes, which are immediately observable, soundness bugs often go unnoticed, leading solvers to silently return incorrect solutions. Such issues potentially lead to serious consequences, including financial loss or flawed decision-making. Flip helps mitigate these risks by uncovering numerous soundness bugs.

Bug Findings by Instance Feasibility. To better understand the effectiveness of the *feasibility-driven instance generation* technique employed in Flip, we further categorize the confirmed bugs by the feasibility of the bug-triggering instances, and summarize the results in Table 3. Among the 67 confirmed bugs, 38 were exposed exclusively by feasible instances, while 22 were triggered solely by infeasible ones. The remaining 7 bugs were exposed by instances generated using both strategies (Flip may produces multiple instances that trigger the same bug). All these 7 bugs are crash bugs, whereas all soundness bugs were discovered exclusively by one strategy or the other. These findings highlight the *complementary nature* of Flip's generation strategies, particularly in detecting correctness issues, underscoring the necessity of both for effective and extensive solver validation.

Table 2: Types of confirmed bugs found by Flip.

Type	SCIP	HiGHS	CBC	Mosek	Gurobi	Total
Crash	14	9	9	0	0	32
Soundness	9	9	10	5	2	35

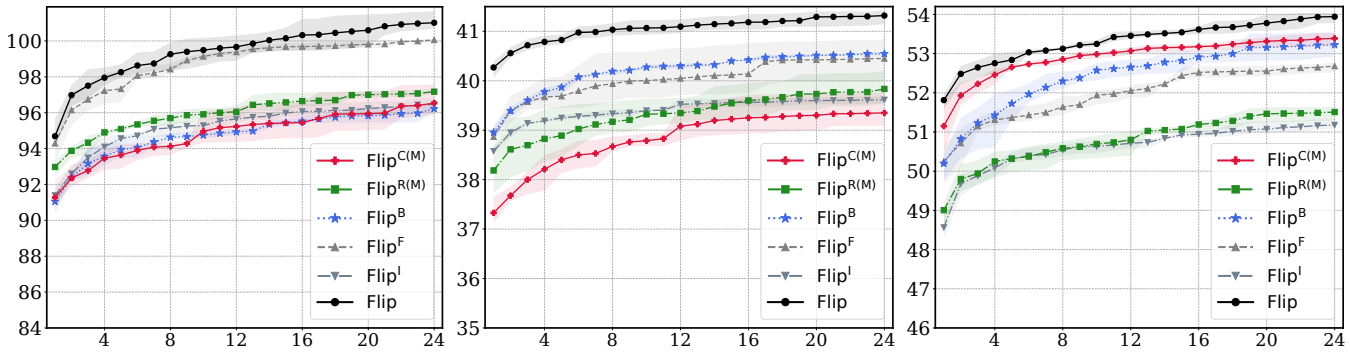
Table 3: Feasibility of bug-triggering instances generated by Flip.

Feasibility	SCIP	HiGHS	CBC	Mosek	Gurobi	Total
Feasible	10	11	12	3	2	38
Both	3	4	0	0	0	7
Infeasible	10	3	7	2	0	22

RQ2: How much code coverage can Flip achieve?

This research question evaluates Flip's instance generation strategies using code coverage, a standard metric for measuring the effectiveness of software testing techniques. The evaluation focuses on three open-source solvers, namely HiGHS, SCIP, and CBC. Although general-purpose fuzzers such as AFL [72] have shown success in domains like file parsers and system utilities, they rely on low-level, syntax-agnostic mutations (e.g., byte flipping), which are poorly suited for software systems that require strictly structured inputs with strong semantic validity. Prior work [73] has shown that these fuzzers struggle to produce meaningful inputs for structured-input programs, including compilers [19] and SMT solvers [58]; the same challenge applies to MIP solvers. For this reason, AFL-like fuzzers are excluded from our evaluation. For a rigorous comparison, we first consider a fully random generation strategy without any feasibility guarantees. In addition, the operations research community has explored automated instance synthesis for the purpose of designing and evaluating optimization algorithms [5, 16, 28, 36, 37] These techniques typically construct instances with specific mathematical properties. We incorporate two representative approaches [5, 37] as baselines in our study. Concretely, we configure three instance generation strategies within Flip as baselines for comparison:

- **Random generation**: This strategy adopts a *blind random instance generation* approach. It produces syntactically and semantically valid MIP instances by randomly synthesizing decision variables and constraints, without any guarantees of feasibility. We refer to this baseline as Flip^{B}.
- **Simon et al.'s approach** [5]: This method generates feasible LP instances with known optimal solutions. We implemented it as an additional instance generation strategy in Flip, referred to as $\text{Flip}^{\text{C(L)}}$. Since the original algorithm targets only LP, we



(a) SCIP (b) HiGHS (c) CBC

Figure 2: Line coverage trend of $\text{Flip}^{\text{C(M)}}$, $\text{Flip}^{\text{R(M)}}$, Flip^{B}, Flip^{F}, Flip^{I}, and Flip over a 24-hour fuzzing campaign ($\text{Flip}^{\text{C(L)}}$ and $\text{Flip}^{\text{R(L)}}$ are not shown for clarity). The x-axis represents the fuzzing time in hours, and the y-axis shows line coverage in thousands of lines (#K lines). The shaded regions represent the coverage range (minimum to maximum) across three repeated trials whereas the line shows the mean value.

extend it to MIP by adding integrality constraints [5]. Notably, the optimality guarantee does not hold in this extended setting, therefore differential testing is still applied as the oracle. We denote the extended version as $\text{Flip}^{\text{C(M)}}$.

- **Li et al.'s approach** [37]: Building upon Simon et al.'s algorithm [5], this method further constrains the rank and condition number of the constraint matrix during LP instance generation. Similarly, we implemented the original algorithm and extended it to MIP within Flip, resulting in two variants: $\text{Flip}^{\text{R(L)}}$ and $\text{Flip}^{\text{R(M)}}$.

We also configure two specialized variants of Flip, namely Flip^{F} and Flip^{I}, which exclusively generate feasible and infeasible instances, respectively. The coverage evaluation then proceeds in two steps:

(1) Run a 24-hour testing session in single-threaded mode with each version of Flip, and collect all the instances generated during each session.
(2) Build the three solvers with instrumentation for coverage measurement. Then, for each variant of Flip, use the corresponding instrumented solver to solve the collected instances in step (1), and collect line, branch, and function coverage.

The coverage collection process (step 2) is conducted separately after the 24-hour testing session (step 1). The reason is that coverage collection is generally not required in real-world testing, and the collection process is time-consuming, which may restrict the throughput of fuzzing. We use gcov [21] as the coverage collection tool. Each experiment is repeated three times to mitigate randomness, and we report the average results across the three runs.

The detailed coverage results of line, branch, and function coverage are summarized in Table 4, while Figure 2 presents a visual comparison of the line coverage trends over the 24-hour testing session. $\text{Flip}^{\text{C(L)}}$ and $\text{Flip}^{\text{R(L)}}$ are excluded from the figures for clarity, as they achieve substantially lower coverage than the other configurations (see Table 4), which is expected since they generate only LP instances and therefore exercise fewer code paths within MIP solvers. Overall, Flip consistently achieves the highest line coverage among the baselines. Compared to $\text{Flip}^{\text{C(M)}}$, Flip covers 4,478, 1,969, and 554 more lines on SCIP, HiGHS, and CBC, respectively. Similarly, Flip outperforms $\text{Flip}^{\text{R(M)}}$ by covering 3,843, 1,484, and 2,433 additional lines on the same solvers. Even when restricted to generating feasible instances only, Flip^{F} still outperforms $\text{Flip}^{\text{C(M)}}$ and $\text{Flip}^{\text{R(M)}}$ in most cases. Compared to Flip^{B}, Flip covers 4,793, 772, and 718 more lines on SCIP, HiGHS, and CBC, respectively. These improvements demonstrate the effectiveness of Flip in exercising more diverse and deeper code paths within MIP solvers.

Moreover, Flip typically achieves higher code coverage than both Flip^{F} and Flip^{I}. This pattern indicates that the two generation approaches of Flip, i.e., generating feasible and infeasible instances, are complementary, each covering a unique set of code that the other does not. This result further strengthens our findings in RQ1, where we observed that both feasible and infeasible instances are necessary for finding certain classes of bugs.

RQ3: How does the feasibility-driven instance generation strategy enhance bug detection?

This research question investigates how the *feasibility-driven instance generation* strategy enhances the bug-detection capability of Flip. To rigorously assess its effectiveness, we conducted a side-by-side comparison between Flip and the baseline configurations introduced in RQ2. All fuzzers were executed under identical experimental conditions—each running continuously for one week (7×24 hours) on the same hardware platform. The bug detection results are presented in Table 5. $\text{Flip}^{\text{C(L)}}$ and $\text{Flip}^{\text{R(L)}}$ are excluded from the table, as they did not identify any bugs, which aligns with their limited code coverage observed in RQ2. Additionally, no bugs were discovered in Gurobi during this evaluation.

The results show that Flip significantly outperforms all baselines in bug detection by finding 2.7x to 4x more bugs. This finding aligns with the code coverage results in RQ2, as higher coverage generally correlates with stronger bug-detection capability. To better understand these results, we further analyzed the characteristics of the instances generated by different approaches, identifying their limitations and illustrating how Flip, through the *feasibility-driven instance generation* strategy, effectively addresses them.

Feasible Instance Generation. Flip^{B} employs a blind random instance generation strategy with no feasibility guarantees. As a

Table 4: Code coverage evaluations on solver SCIP, HiGHS, and CBC. Column *line*, *branch*, and *function* represent line coverage, branch coverage, and function coverage, respectively. Best results in each group are highlighted in bold.

	SCIP			HiGHS			CBC		
	line	*branch*	*function*	*line*	*branch*	*function*	*line*	*branch*	*function*
Flip^{B}	96,221	67,018	8,038	40,547	26,137	14,621	53,224	33,322	4,655
Flip$^{C(L)}$	59,792	37,140	6,289	16,113	8,497	5,829	21,465	10,842	2,068
Flip$^{C(M)}$	965,36	67,448	8,115	39,350	24,995	14,426	53,388	33,505	4,664
Flip$^{R(L)}$	612,93	38,271	6,388	16,110	8,457	5,784	20,950	10,636	2,057
Flip$^{R(M)}$	97,171	67,857	8,159	39,835	25,257	14,565	51,509	32,327	4,635
Flip^{F}	100,054	70,371	**8,262**	40,451	26,055	14,633	52,681	32,670	4,670
Flip^{I}	96,448	67,403	8,038	39,614	25,422	14,570	51,177	31,602	4,555
Flip	**101,014**	**71,298**	8,221	**41,319**	**26,912**	**14,738**	**53,942**	**33,927**	**4,690**

result, it frequently produces contradictory constraints, yielding a low proportion of feasible instances. An analysis of the instances generated during the 24-hour testing session in RQ2 shows that only 15% are feasible. Moreover, most of these are small and structurally simple, since larger random instances are statistically more likely to contain contradictory constraints, leading to infeasibility. The scarcity of nontrivial feasible instances limits the extent to which solver mechanisms for finding optimal solutions are exercised. Consequently, Flip^{B} is less effective at uncovering deeper bugs that require more intricate inputs. In contrast, Flip generates feasible instances spanning a wide range of sizes and structural complexities, enabling it to expose a broader and deeper set of bugs.

Table 5: Bug count found by different approaches in RQ3.

	SCIP	HiGHS	CBC	Mosek	Total
Flip^{B}	1	1	0	0	2
Flip$^{C(M)}$	0	1	1	1	3
Flip$^{R(M)}$	2	1	0	0	3
Flip	2	3	2	1	8

While Flip$^{C(M)}$ and Flip$^{R(M)}$ systematically generate feasible instances, Flip outperforms them by greater randomness during generation. Specifically, Flip$^{C(M)}$ and Flip$^{R(M)}$ were originally designed to enforce additional properties, such as guaranteeing a known optimal solution, which impose constraints on the generation process and thus reduce diversity. Taking Flip$^{C(M)}$ as an example (Flip$^{R(M)}$ is based on it and shares similar framework), after randomly generating the constraint matrix and the solution, it computes the right-hand-side vector and objective coefficients deterministically. As a result, for each pair of matrix and solution, only a single instance can be produced, potentially missing many valid variants that could reveal different solver behaviors. In contrast, Flip fixes only the solution when generating a feasible instance. The rest of the instance remains highly randomized: the constraint matrix is generated freely, and after fixing the matrix and solution, there are infinitely many valid right-hand-side vectors and objective functions. This broader exploration space enables Flip to exercise more code paths and discover more bugs.

Infeasible Instance Generation. Our evaluation also highlights that infeasible instances are crucial for exhaustive solver validation. However, Flip$^{R(M)}$ and Flip$^{C(M)}$ inherently generate only feasible instances, preventing them from extensively exploring solver code paths related to infeasibility detection. Conversely, while the majority (approximately 85%) of instances generated by Flip^{B} are infeasible, they are typically too trivial to effectively exercise the solver's infeasibility-detection mechanisms. *Indeed, Flip^{B} did not uncover any bugs from these infeasible instances during our evaluation.* In contrast, Flip found 3 bugs with infeasible instances. The reason is that randomly generated constraints often produce superficial contradictions within small subsets of the problem, resulting in easily detectable infeasibility. In contrast, Flip generates infeasible instances with n constraints by first constructing a feasible instance with $n-1$ constraints and then injecting a single feasibility-breaking constraint. This approach introduces subtle infeasibility that is more challenging to detect, effectively testing the solver's infeasibility reasoning mechanisms.

5.3 Assorted Bug Samples

Flip has found many bugs across multiple MIP solvers, including soundness issues, assertion failures, and memory errors. This subsection presents a selection of these bugs. The original bug-triggering instances are sometimes too large to be presented in full, so we show reduced versions obtained through test case reduction.

Figure 3a illustrates a soundness bug in HiGHS, where the solver reports infeasibility for this feasible instance in default mode. The issue lies in an incorrect transformation during presolve, a general process that simplifies the input instance using predefined transformations before solving, which causes the solver to incorrectly determine the instance as infeasible.

Figure 3b shows an infeasible instance that triggers a soundness issue in Mosek. The solver incorrectly reports the instance as feasible and returns a solution with $x_0 = 200$ and $x_1 = 204.2$, where the value of x_1 violates its upper bound constraint $x_1 \leq 200$. The developers promptly confirmed this issue, stating: *"This is an issue we will have to look at and fix."*

Figure 3c presents an infeasible instance that triggers SCIP to crash due to multiple memory leaks. The issue was introduced by a previous bug-fix commit that failed to handle a specific case in cutoff generation. The developers later resolved the problem by adding the necessary conditional checks.

```
Minimize
OBJ:-93 x0 + 25 x1 + 17 x2
Subject To
 C1: -89 x0 - 40.7 x1 - 12 x2 ≥ 3994.58
 C2: -0.1 x0 + 77 x1 - 23.7 x2 ≥ -4878
 C3: 53.2 x0 - 52.9 x1 - 77 x2 ≥ 1375
 C4: -97 x0 - 2.7 x1 + 25.6 x2 ≥ 921.2
 C5: -8.6 x0 - 6 x1 + 72.7 x2 ≥ -4930
Bounds
 -200 ≤ x0 ≤ 200
 -200 ≤ x1 ≤ 200
 -200 ≤ x2 ≤ 200
Generals
 x0
 x2
End
```

(a) *Soundness Bug in HiGHS*: HiGHS mistakenly reports infeasibility for this feasible instance in default mode [26].

```
Minimize
OBJ: -24.36 x0 - 39.52 x1
Subject To
 C1: 96.16 x0 - 78.59 x1 <= 3188
 C2: 3.11 x0 - 74.4 x1 <= -4980
 C3: -33.6 x0 - 73 x1 <= 496.6
 C4: 59.6 x0 + 24.9 x1 >= 1550
 C5: 40.3 x0 + 27.5 x1 >= -1403
 C6: -85.37 x0 + 93.1 x1 >= -1011
 C7: -36.3 x0 + 29.19 x1 <= -1299.3
Bounds
 -100 ≤ x0 ≤ 200
 -100 ≤ x1 ≤ 200
Generals
 x0
End
```

(b) *Soundness Bug in Mosek*: Mosek returns an invalid solution that breaks variable bounds for this infeasible instance [2].

```
Minimize
OBJ:92 x0 + 20.3 x1
Subject To
 C1: -35.7 x0 - 74.23 x1 ≥ 1135
 C2: 27.9 x0 + 23.7 x1 ≥ -4595.3
 C3: -56.48 x0 - 30.2 x1 ≥ 2469
 C4: -30 x0 - 87.1 x1 ≤ 3780.5
 C5: 88.94 x0 - 43.13 x1 ≤ -2234
 C6: -13.06 x0 + 6.18 x1 ≤ 333.2
Bounds
 -100 ≤ x0 ≤ 100
 -200 ≤ x1 ≤ 200
Generals
 x0
 x1
End
```

(c) *Crash Bug in SCIP*: This infeasible instance triggers SCIP to crash with memory leaks in default mode [57].

```
Minimize
OBJ: -83.6 x0 + 90.71 x1
Subject To
 C1: 64.92 x0 + 54.46 x1 ≤ 2094.02
 C2: 6.73 x0 + 36.39 x1 ≤ -1501.25
 C3: -50.6 x0 + 77.22 x1 ≥ -4433.86
 C4: -71.05 x0 + 90.75 x1 ≤ 4165.75
 C5: 62.77 x0 + 75.92 x1 ≤ -341.91
Bounds
 -100 ≤ x0 ≤ 100
 -200 ≤ x1 ≤ 200
Generals
 x0
 x1
End
```

(d) *Soundness Bug in Gurobi*: Gurobi returns a non-optimal solution for this feasible instance with user-specified parameters [50].

```
Minimize
OBJ: x0 + x1 + x2
Subject To
 C1: 73.12x0 - 35.8x1 + 79.8x2 ≤ 4120
 C2: 4.5x0 - 82x1 - 98x2 ≤ 2483.99
 C3: 79x0 - 56x1 + 17.1x2 ≥ -326
 C4: 54x0 + 0.5x1 + 30x2 ≤ 468
 C5: -17.92x0 + 60x1 - 47.66x2 ≤ -12910
Bounds
 -200 ≤ x0 ≤ 200
 -200 ≤ x1 ≤ 200
 -200 ≤ x2 ≤ 200
Generals
 x0 x1 x2
End
```

(e) *Soundness Bug in SCIP*: SCIP returns an invalid solution that violates the constraints for this infeasible instance [3].

```
Minimize
OBJ:-74.92 x0 + 65.53 x1
Subject To
 C1: -91.7 x1 - 59 x2 ≤ -3844
 C2: 39.7 x1 - 52.8 x2 - 96 x3 ≤ 4979
 C3: 45.02 x1 - 24.12 x2 ≤ 0
 C4: 69 x0 - 11 x1 + 96 x2 + 62.9 x3 ≤ 2034
 C5: 70 x0 - 85.2 x1 + 14 x2 + 17 x3 ≥ -3673
Bounds
 0 ≤ x0
 -100 ≤ x2
Generals
 x0
 x1
End
```

(f) *Crash Bug in HiGHS*: This feasible instance triggers HiGHS to crash in default mode with an assertion failure [27].

Figure 3: Assorted bug-triggering MIP instances generated by Flip.

Figure 3d illustrates a soundness issue in Gurobi, where the solver returns a non-optimal solution when barrier crossover is disabled and model scaling is set to level one. Although these parameters are designed to improve solving performance, they should not compromise correctness. The developers of Gurobi confirmed this issue as a bug and opened a ticket to track the fixing process.

Figure 3e presents a soundness issue in SCIP, where the solver incorrectly identifies the infeasible instance as feasible, and returns an invalid solution that violates constraint C2. The developer described this issue as *"rather hard to find"* and explained that its root cause lies in a constraint being incorrectly rounded to a relaxed form by the linear constraint handler.

Figure 3f shows a feasible instance that triggers HiGHS to crash due to an assertion failure [27]. According to the developer who fixed the issue, the same crash happened in another bug report from a real-world user, potentially underscoring its practical relevance.

5.4 Discussion

Contributions and Feedback. Flip has uncovered many bugs in both open-source and commercial MIP solvers. After we reported these bugs, most of them have been confirmed and subsequently fixed by the solver developers, contributing to the overall correctness and reliability of these systems. Our efforts and findings have been acknowledged and appreciated by the MIP solver community. Some bug-triggering instances in our reports were integrated into the regression test suites of the respective solvers. Moreover, our

[2] This bug was reported in the internal repository of SCIP.
[3] Bugs in Mosek were reported privately via email and are not publicly disclosed.

reports have received positive feedback from solver developers. For example, after we reported the soundness bug shown in Figure 1a to Gurobi, one developer almost immediately confirmed the issue and responded: *"Thank you again for reporting this to us."* Besides, we received *"Thanks for the nice reproducible bug reports! With small numerically stable instances!"* from a HiGHS developer, and *"Thank you, this one is rather hard to find!"*, *"... is another crucial bug, thank you again!"* from SCIP team. Notably, after we contributed multiple issues to SCIP, their development team invited us to join their internal repository to report issues directly, an endorsement that underscores the practical impact and value of our work.

Limitations and Future Work. While Flip has proven effective, it has limitations. One limitation is that the optimal solutions of feasible instances generated by Flip are unknown, necessitating the use of differential testing as the correctness oracle, and reliance on a reference solver for infeasible instance generation. An important direction for future work is to develop generation techniques that construct feasible instances with known optimal solutions. In addition, when constructing infeasible instances, Flip introduces subtle infeasibility by injecting a single feasibility-breaking constraint. This strategy, while has proven effective, may overlook bugs triggered by other forms of infeasibility, e.g., intricate contradictions among multiple subsets of constraints. Further research could explore alternative infeasible instance generation methods to further enrich the diversity of infeasibility patterns.

6 Related Work

We discuss four lines of related work.

Automated LP/MIP Instance Synthesis. To support the design and evaluation of new optimization algorithms, several prior works in operations research have explored automated synthesis of LP/MIP instances. For example, Delorme et al. [16] generate instances for kidney exchange programs; additional generation techniques have been proposed for knapsack [28], Hamiltonian completion [36], and job-shop scheduling problems [61]. Several learning-based methods synthesize new instances from existing datasets [23, 68, 74]. While these efforts typically target a specific problem class or produce instances that resemble existing ones, Simon et al. [5] introduced a more general approach for generating random LP instances with known optimal solutions. Building on this work, Li et al. [37] further control the condition number and rank of the constraint matrix during instance generation. Our evaluation (Section 5) incorporates these two techniques as baselines and shows that Flip achieves higher code coverage and superior bug-finding effectiveness.

Generation-Based Software Testing. Producing high-quality test inputs is a long-standing challenge in software testing. Generation-based testing, where test inputs are synthesized from scratch (as opposed to mutation-based testing, which mutates existing inputs to create new ones), has been widely explored in the literature. For instance, Csmith [69] and YARPGen [40, 41] generate well-defined C programs to test C/C++ compilers, and found hundreds of bugs in GCC and LLVM. NNSmith [38] synthesizes diverse and valid neural network models to test deep learning compilers. Bugariu and Müller [8] proposed a generation-based approach to synthesize SMT formulas for testing string solvers. Similarly, our technique Flip follows a generation-based approach, employing a *feasibility-driven* strategy to construct well-formed MIP instances.

Differential Testing. The test oracle problem, i.e., determining whether a program's output is correct given an input, is a fundamental problem in software testing. Differential testing [44] is a well-established solution, which addresses this challenge by comparing outputs from multiple implementations of the same specification. This approach has been successfully applied in various domains. For instance, Csmith [69] employs differential testing to uncover hundreds of bugs in C compilers. Differential testing has also been used to validate JIT compilers [11, 53], databases [59, 75], and deep learning frameworks [17, 22, 52]. Flip employs differential testing for validation with *feasible* instances. For *infeasible* instances, differential testing is unnecessary, as an explicit oracle exists, i.e., the solver should always report infeasibility.

Validation of SMT Solvers. In the literature, substantial efforts have been made to validate SMT solvers[7, 43, 51, 58, 66, 67, 70, 71]. For example, FuzzSMT [7] is an early work that combines grammar-based fuzzing and differential testing to find SMT solver bugs. STORM [43] and Yinyang [67] employ blackbox mutation and semantic fusion, respectively, uncovering numerous soundness issues in Z3 [15] and CVC4 [1], two state-of-the-art SMT solvers. Janus [6] introduced the notion of incompleteness bugs, where solvers fail to return a result for solvable problems, and identified dozens of such bugs in Z3 and CVC5 (the successor of CVC4). Flip is related to these works, as both MIP and SMT solvers aim to reason over mathematically defined constraints. A key distinction is that SMT solvers focus solely on *satisfiability*, whereas MIP solvers further seek *optimality* within the feasible region. Flip addresses this challenge with feasibility-driven instance generation, and demonstrates its effectiveness in uncovering real-world bugs.

7 Conclusion

This paper introduced Flip, the first specialized fuzzing framework for MIP solvers. At its core, Flip is powered by *feasibility-driven instance generation*, a novel technique for generating diverse MIP instances that enable effective solver validation. The key idea is to systematically construct MIP instances that are provably feasible or infeasible by design. Our extensive evaluation demonstrates that Flip is effective at uncovering real-world bugs: to date, it has discovered 67 bugs across five widely used solvers. These findings have been acknowledged and appreciated by the MIP solver community. This work marks a promising first step toward systematic MIP solver validation via fuzz testing, and we hope it will motivate continued research in this important area.

Acknowledgments

We thank all the anonymous reviewers in ICSE'26 for their insightful feedback and comments. Our special thanks go to the MIP solver developers, especially Dominik Kamp, John Forrest and Julian Hall, for their useful information and addressing our bug reports promptly. This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, a project under WHJIL, and CFI-JELF Project #40736.

References

[1] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, 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 23*. Springer, 171–177.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

[4] Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, et al. 2024. The SCIP optimization suite 9.0. *arXiv preprint arXiv:2402.17702* (2024).

[5] Simon Bowly, Kate Smith-Miles, Davaatseren Baatar, and Hans Mittelmann. 2020. Generation techniques for linear programming instances with controllable properties. *Mathematical Programming Computation* 12, 3 (2020), 389–415.

[6] Mauro Bringolf, Dominik Winterer, and Zhendong Su. 2022. Finding and understanding incompleteness bugs in SMT solvers. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–10.

[7] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. 1–5.

[8] Alexandra Bugariu and Peter Müller. 2020. Automatically testing string solvers. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 1459–1470.

[9] Abraham Charnes and William Wager Cooper. 1957. Management models and industrial applications of linear programming. *Management science* 4, 1 (1957), 38–91.

[10] Abraham Charnes, William W Cooper, and Merton H Miller. 1959. Application of linear programming to financial budgeting and the costing of funds. *The Journal of Business* 32, 1 (1959), 20–46.

[11] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

[12] Kalman J Cohen and Frederick S Hammer. 1967. Linear programming and optimal bank asset management decisions. *The Journal of Finance* 22, 2 (1967), 147–165.

[13] COIN-OR 2025. *COIN-OR Branch and Cut (CBC)*. COIN-OR. https://www.coin-or.org/Cbc/

[14] George B Dantzig. 2016. Linear programming and extensions. (2016).

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[16] Maxence Delorme, Sergio García, Jacek Gondzio, Jörg Kalcsics, David Manlove, William Pettersson, and James Trimble. 2022. Improved instance generation for kidney exchange programmes. *Computers & Operations Research* 141 (2022), 105707.

[17] Zizhuang Deng, Guozhu Meng, Kai Chen, Tong Liu, Lu Xiang, and Chunyang Chen. 2023. Differential testing of cross deep learning framework {APIs}: Revealing inconsistencies and vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7393–7410.

[18] Robert Dorfman, Paul Anthony Samuelson, and Robert M Solow. 1987. *Linear programming and economic analysis*. Courier Corporation.

[19] Karine Even-Mendoza, Arindam Sharma, Alastair F Donaldson, and Cristian Cadar. 2023. Grayc: Greybox fuzzing of compilers and analysers for c. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1219–1231.

[20] Len L Garver. 2007. Transmission network estimation using linear programming. *IEEE Transactions on power apparatus and systems* 7 (2007), 1688–1697.

[21] GNU Project. 2024. *GCOV: a Test Coverage Program*. Free Software Foundation. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html Part of the GNU Compiler Collection (GCC).

[22] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 739–743.

[23] Ziao Guo, Yang Li, Chang Liu, Wenli Ouyang, and Junchi Yan. 2024. Acm-milp: Adaptive constraint modification via grouping and selection for hardness-preserving milp instance generation. In *Forty-first International Conference on Machine Learning*.

[24] Gurobi Optimization, LLC 2025. *Gurobi Optimizer Reference Manual*. Gurobi Optimization, LLC. https://www.gurobi.com/

[25] Gurobi Optimization, LLC. 2025. *Parameter Groups: Tolerances*. Gurobi Optimization, LLC. https://docs.gurobi.com/projects/optimizer/en/current/concepts/parameters/groups.html#tolerances Accessed June 12, 2025.

[26] HiGHS. 2025. Bug Report. https://github.com/ERGO-Code/HiGHS/issues/2432

[27] HiGHS. 2025. Bug Report. https://github.com/ERGO-Code/HiGHS/issues/2455

[28] R Hill, J Moore, C Hiremath, and YK Cho. 2011. Test problem generation of binary knapsack problem variants and the implications of their use. *Int. J. Oper. Quant. Manag* 18, 2 (2011), 105–128.

[29] Alexander Hoen and Ambros Gleixner. 2025. Analyzing the numerical correctness of branch-and-bound decisions for mixed-integer programming. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 35–50.

[30] Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. 2024. Certifying MIP-based presolve reductions for 0–1 integer linear programs. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 310–328.

[31] Qi Huangfu and JA Julian Hall. 2018. Parallelizing the dual revised simplex method. *Mathematical Programming Computation* 10, 1 (2018), 119–142.

[32] Narendra Karmarkar. 1984. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 302–311.

[33] James E Kelley, Jr. 1960. The cutting-plane method for solving convex programs. *Journal of the society for Industrial and Applied Mathematics* 8, 4 (1960), 703–712.

[34] Eugene L Lawler and David E Wood. 1966. Branch-and-bound methods: A survey. *Operations research* 14, 4 (1966), 699–719.

[35] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

[36] Thibault Lechien, Jorik Jooken, and Patrick De Causmaecker. 2023. Evolving test instances of the Hamiltonian completion problem. *Computers & Operations Research* 149 (2023), 106019.

[37] Anqi Li, Congying Han, Tiande Guo, and Bonan Li. 2024. Generating linear programming instances with controllable rank and condition number. *Computers & Operations Research* 162 (2024), 106471.

[38] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543.

[39] Qian Liu and Garrett Van Ryzin. 2008. On the choice-based linear programming model for network revenue management. *Manufacturing & Service Operations Management* 10, 2 (2008), 288–310.

[40] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[41] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1826–1847.

[42] Jinhua Lyu and Jonathan F Bard. 2025. Weekly crew scheduling for freight rail engineers: A network approach. *Journal of Rail Transport Planning & Management* 34 (2025), 100519.

[43] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 701–712.

[44] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[45] Sanjay Mehrotra. 1992. On the implementation of a primal-dual interior point method. *SIAM Journal on optimization* 2, 4 (1992), 575–601.

[46] MOSEK ApS 2025. *MOSEK Optimization Software*. MOSEK ApS. https://www.mosek.com/

[47] MOSEK ApS. 2025. *Parameters grouped by topic* (11.0.22 ed.). MOSEK ApS. https://docs.mosek.com/latest/pythonapi/param-groups.html Accessed 12 Jun 2025.

[48] John A Nelder and Roger Mead. 1965. A simplex method for function minimization. *The computer journal* 7, 4 (1965), 308–313.

[49] Gurobi Optimization. 2025. Bug Report. https://support.gurobi.com/hc/en-us/community/posts/36377666909713-Gurobi-presolve-mistakenly-reports-infeasibility-for-a-feasible-instance

[50] Gurobi Optimization. 2025. Bug Report. https://support.gurobi.com/hc/en-us/community/posts/35494813103889-Potential-Numerical-Bug-in-MIP-Solving

[51] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative type-aware mutation for testing SMT solvers. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–19.

[52] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[53] Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. 2022. Interpreter-guided differential JIT compiler unit testing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 981–992.

[54] PuLP Developers 2025. *Optimization with PuLP*. PuLP Developers. https://coin-or.github.io/pulp/

[55] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX security symposium (USENIX Security 17)*. 167–182.

[56] SCIP. 2025. Bug Report. https://github.com/scipopt/scip/issues/145

[57] SCIP. 2025. Bug Report. https://github.com/scipopt/scip/issues/155

[58] Joseph Scott, Federico Mora, and Vijay Ganesh. 2020. Banditfuzz: A reinforcement-learning based performance fuzzer for smt solvers. In *Software Verification: 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20–21, 2020, Revised Selected Papers 13*. Springer, 68–86.

[59] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing database systems via differential query execution. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2072–2084.

[60] George J Stigler. 1945. The cost of subsistence. *Journal of farm economics* 27, 2 (1945), 303–314.

[61] Simon Strassl and Nysret Musliu. 2022. Instance space analysis and algorithm selection for the job shop scheduling problem. *Computers & Operations Research* 141 (2022), 105661.

[62] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*. 361–371.

[63] Robert J Vanderbei. 2015. Linear programming. In *Encyclopedia of Applied and Computational Mathematics*. Springer, 796–800.

[64] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. 2021. {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 2741–2758.

[65] Wikipedia contributors. 2024. MPS (format) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/MPS_(format). Accessed: 2025-06-08.

[66] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[67] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on programming language design and implementation*. 718–730.

[68] Tianxing Yang, Huigen Ye, and Hua Xu. 2024. Learning to generate scalable milp instances. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 159–162.

[69] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[70] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT solvers via two-dimensional input space exploration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–335.

[71] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Skeletal approximation enumeration for smt solver testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1141–1153.

[72] Michal Zalewski. 2024. *American Fuzzy Lop*. Google LLC. https://lcamtuf.coredump.cx/afl/ Accessed: 2025-06-10.

[73] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book.

[74] Yahong Zhang, Chenchen Fan, Donghui Chen, Congrui Li, Wenli Ouyang, Mingda Zhu, and Junchi Yan. 2024. Milp-fbgen: Lp/milp instance generation with feasibility/boundedness. In *Forty-first International Conference on Machine Learning*.

[75] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding bugs in Gremlin-based graph database systems via randomized differential testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 302–313.

[76] Xintong Zhou, Zhenyang Xu, and Chengnian Sun. 2026. *Artifact for "Validating Mixed-Integer Programming Solvers"*. https://doi.org/10.5281/zenodo.18136798