Demystifying Template-based Invariant Generation for Bit-Vector Programs

Peisen Yao[*], Jingyu Ke[†], Jiahui Sun[*], Hongfei Fu[†✉], Rongxin Wu[‡], and Kui Ren[*]
[*]Zhejiang University, China. Email: {pyaoaa, jasonj, kuiren}@zju.edu.cn
[†] Shanghai Jiao Tong University, China. Email: {Windocotber, jt002845}@sjtu.edu.cn
[‡]School of Informatics, Xiamen University, China. Email: wurongxin@xmu.edu.cn

***Abstract*—The template-based approach to invariant generation is a parametric and relatively complete methodology for inferring loop invariants. The relative completeness ensures the generated invariants' accuracy up to the template's form and the inductive condition. However, there has been limited in advancing the approach to bit-precise reasoning, which involves modeling integers using bit-vector arithmetic. This is unfortunate because bit-precise reasoning is crucial for faithfully and accurately modeling machine integer semantics and, thus, for ensuring sound and precise program verification.**

In this experience paper, we present an experimental study of bit-precise, template-based invariant generation on three fronts: the precision of different invariant templates, the performance of different constraint solvers for solving the constraints, and the effectiveness of the template-based approach compared to existing bit-precise verification techniques. Through an extensive experimental evaluation over a wide range of benchmarks, we find that (1) the choices of invariant templates and constraint solvers have varying degrees of impact on the precision and efficiency of invariant generation; (2) the template-based approach can handle benchmarks that other approaches for bit-vectors cannot handle. The results also reveal several guidelines for advancing future research on template-based invariant generation.

***Index Terms*—Invariant generation, constraint solving, comparison and analysis**

I. Introduction

An assertion at a program location is an *invariant* if it is always satisfied by the values of the program variables whenever the location is reached during program execution. Invariants are essential in program analysis and verification because they offer a sound overestimation of the program's reachable states. As a result, they have been widely used in reasoning about properties of programs, such as safety, non-interference, and complexity. Over the last few decades, various techniques have been proposed for automatically generating invariants, such as abstract interpretation [1–4], template-based approach [5–7], SMT-based model checking [8, 9], recurrence analysis [10–13], and machine learning techniques [14–17].

A significant amount of work on invariant generation follows the template-based approach [5–7, 18–26, 26–30].[1] This approach first establishes a template with unknown parameters for the target invariant and then finds invariants that match the template by extracting and solving constraints on the unknowns. For example, consider the invariant template $x \leq a$, where x is the program variable, and a is the unknown parameter. The approach involves generating and solving constraints in the form of $\forall x, x'.\Phi(x, x', a)$, where Φ is encoded from the program structure (detailed in § II-B).

The template-based approach offers several notable advantages for invariant generation. First, the approach is relatively complete, provided that the underlying logic of the constraint is complete. This means that the approach can find the desired invariant if it is expressible in the template. This completeness guarantee is vital for generating precise invariants [5] and is often not offered by conventional abstract interpretation [31, 32]. Second, the approach is goal-directed, as it only computes the invariants that are necessary for establishing program correctness. Specifically, the constraint solver combines information from the pre-condition, program, and post-condition into a single quantified constraint. Third, unlike many existing SMT-based model-checking algorithms [8, 9, 33, 34], the approach does not rely on advanced SMT solver features such as interpolant generation.

Table I categorizes representative algorithms in the template-based invariant generation literature by their targeted programs and invariant templates. The template-based approach has been extensively employed in generating both linear and non-linear invariants for affine programs, with some supporting uninterpreted functions, arrays, and probabilistic programs. Besides, most of these efforts treat integer program variables with a limited range (e.g., 32-bit integers) as mathematical integers or reals rather than machine integers (bit-vectors).

Unfortunately, modeling integer program variables using unbounded integers/reals does not faithfully capture all the properties of machine integers, such as wrap-around behavior caused by under- and overflows. This becomes especially problematic in the verification of low-level system code, as it may compromise the soundness (e.g., by discovering an invalid invariant) and completeness (e.g., by missing a critical invariant) of the verifier. For example, consider the program in Figure 1. A verification tool based on unbounded integers will conclude that the assertion must hold because a value starting at one remains positive no matter how many times it is incremented. However, this conclusion fails to reflect the actual behavior of machine integer semantics, as Line 3 could lead to an integer overflow.

⋆ Peisen Yao and Kui Ren are also with the ZJU-Hangzhou Global Scientific and Technological Innovation Center.

[1]Note that in the literature, some works also refer to this methodology as "constraint solving-based approach".

```
1     assume int x = 1;
2     while (*) {
3       x = x + 1;
4     }
5     assert x > 0;
```

Fig. 1: A code snippet.

TABLE I: A summary of representative studies on template-based invariant generation.

Programs	Invariant Templates	Algorithms
linear arithmetic	polyhedron	[5–7, 18–21]
linear arithmetic	non-linear	[22–26, 26–30]
linear arithmetic + UF	polyhedron	[43]
linear arithmetic + array	polyhedron over array accesses	[44]
linear arithmetic + probability	polyhedron	[45]
bit-vector arithmetic	polyhedron	[46, 47]

Indeed, both the hardware model checking and software verification communities have dedicated significant efforts to extending various verification techniques to support bit-vector arithmetic, such as abstract interpretation, CEGAR [8, 33], IC3/PDR [9, 34–37], and syntax-guided synthesis [38, 39]. However, while there are a few studies on template-based ranking function synthesis for bit-vector arithmetic [40–42], relatively little progress has been made in lifting the approach to invariant generation for bit-vector programs.

This paper contributes to the first empirical evaluation of template-based invariant generation for bit-vector programs. First, we compare the efficiency and effectiveness of the template-based approach when using different invariant templates and constraint solving strategies. Second, we compare the performance of the approach against other invariant generation techniques for verifying bit-vector programs. An in-depth study will assist in choosing the appropriate verification engine and guide the further exploration of existing verification techniques.

As a first step towards achieving the goal, we present EAGLE, a template-based invariant generator for bit-vector programs. EAGLE takes as input SyGuS or CHC files annotated with assertions. It offers various invariant templates, such as interval, zone, octagon, and bounded polyhedrons, along with two precision-enhancing strategies: disjunctive completion and property strengthening (§ IV-A). Moreover, we conduct a systematic review of existing techniques for solving the quantified bit-vector constraints in the template-based approach (§ IV-B) and integrate a diverse group of constraint solving engines into EAGLE.

We collect a set of 240 benchmarks from the invariant track of SyGuS-COMP and previous literature on bit-precise verification. We run EAGLE using different invariant templates and constraint solvers, and compare it against other verification techniques, including CEGAR, IC3/PDR, and enumerative synthesis. Our study demonstrates that the template-based approach implemented in EAGLE can handle benchmarks beyond the reach of previous approaches. This highlights the effectiveness of the approach in generating invariants for bit-vector programs. Furthermore, our findings reveal several guidelines for advancing future research on template-based invariant generation, such as the choice of invariant templates and potential directions for algorithmic improvements (§ V-C).

To summarize, this paper makes the following contributions:

- We present a comprehensive framework for the template-based invariant generation of bit-vector programs and collect a new data set including 240 programs, which can be used for future research on bit-precise verification. The implementation of our framework, along with the benchmarks suite, is publicly available at the web site: https://tinyurl.com/3ajsbtbr.
- We conduct an extensive study of different invariant templates and constraint solvers for the template-based generation of bit-vector programs, and compare them with other SMT-based invariant generation techniques. The results provide valuable insights into the strengths and limitations of various methods in this domain.

II. PRELIMINARIES

In this section, we first present the basic concepts of inductive invariants (§ II-A) and then introduce the template-based approach to invariant generation (§ II-B).

A. Inductive Loop Invariant

To ease the presentation, we consider programs of the following form throughout this paper:

assume $Pre(X)$; **while** $(G(X))$ $\{$ S; $\}$ **assert** $Post(X)$;

where X is the set of variables occurred in the program, $Pre(X)$ and $Post(X)$ denote the pre- and post-conditions, respectively, $G(X)$ represents the loop condition, and S is the loop body.

Definition 1: An *inductive loop invariant* (loop invariant for short) with respect to the pre-condition $Pre(X)$ and the post-condition $Post(X)$ is an assertion Inv such that:

$$\begin{aligned} \phi_1 : \forall X. Pre(X) \rightarrow Inv(X) \wedge \\ \phi_2 : \forall X, X'. Inv(X) \wedge G(X) \wedge T(X, X') \rightarrow Inv(X') \wedge \\ \phi_3 : \forall X. \neg G(X) \wedge Inv(X) \rightarrow Post(X) \end{aligned} \quad Equation\ (1)$$

where ϕ_1 specifies that Inv must include the program states resulted from initial states, ϕ_2 says that Inv is preserved after every iteration of the loop,[2] and ϕ_3 states that the post-condition must hold when leaving the loop.

Example 1: Consider the following simple integer program:

```
1     assume  int x = 0;
2     while(x < 10) { x = x + 1; }
3     assert x = 10;
```

[2]The two sets X and X' are copied of the sequences of program variables at the beginning and end of a loop iteration, respectively.

Let $Inv(x)$ be the desired inductive loop invariant. We have

$$\forall x.x = 0 \rightarrow Inv(x) \land$$
$$\forall x, x'.Inv(x) \land x < 10 \land x' = x + 1 \rightarrow Inv(x') \land$$
$$\forall x.\neg x < 10 \land Inv(x) \rightarrow x = 10$$

For example, the relation $0 \leq x \leq 10$ is an inductive loop invariant that can verify the program because:

$$\forall x.x = 0 \rightarrow 0 \leq x \leq 10 \land$$
$$\forall x, x'.0 \leq x \leq 10 \land x < 10 \land x' = x + 1 \rightarrow 0 \leq x' \leq 10 \land$$
$$\forall x.\neg x \geq 10 \land 0 \leq x \leq 10 \rightarrow x = 10$$

The loop invariant generation problem is to find an Inv such that Equation (1) holds.

B. Template-based Invariant Generation

To solve the invariant generation problem, the key idea of the template-based approach [5–7] is to find invariants that match a pre-defined template by extracting and solving constraints. Specifically, the approach involves three steps:

1) First, it fixes a desired *template* for the invariants, in which there are unknown quantities. Typically, the template is a fixed expression $F(X, Y)$ over program variables X and *template parameters* Y. Intuitively, the template restricts the syntactical form of the invariants.
2) Second, it generates constraints over X, X', and Y from the structure of the program, following the specification of inductive loop invariants (Definition 1), yielding:

$$\forall X.Pre(X) \rightarrow F(X, Y) \land$$
$$\forall X, X'.F(X, Y) \land G(X) \land T(X, X') \rightarrow F(X', Y) \land$$
$$\forall X.\neg G(X) \land F(X, Y) \rightarrow Post(X)$$
$$Equation \quad (2)$$

For simplicity, we use $\forall X, X'.\Phi(X, X', Y)$ to denote the above constraint in the rest of this paper.
3) Finally, the constraint $\forall X, X'.\Phi(X, X', Y)$ is handed to a constraint solver to compute feasible values of the template parameters Y. The invariant can be obtained by instantiating Y in the template $F(X, Y)$ with the feasible values.

Next, we use Example 1 to illustrate the above steps.

Setting the Invariant Template. First, suppose that we use $a \leq x \leq b$ as the invariant template, where a and b are the template parameters. Intuitively, an assignment to a and b qualifies a candidate interval invariant, e.g., $1 \leq x \leq 3$ with $\{a = 1, b = 3\}$. But a random assignment may not necessarily yield a feasible invariant because it may falsify Equation (2).

Generating Constraints. Second, we extract the constraint following the definition of loop invariants (Equation (1)). Specifically, we replace each occurrence of Inv by the template $a \leq x \leq b$, which yields

$$\forall x.x = 0 \rightarrow a \leq x \leq b \land$$
$$\forall x, x'.a \leq x \leq b \land x < 10 \land x' = x + 1 \rightarrow a \leq x' \leq b \land$$
$$\forall x.\neg x < 10 \land a \leq x \leq b \rightarrow x = 10$$

The above constraint serves as a declarative specification of the desired invariants.

Solving Constraints. Finally, we solve the above constraint to obtain feasible values of a and b. For example, a feasible solution is $\{a = 0, b = 10\}$. After replacing a, b in the template $a \leq x \leq b$ with the solution, we can obtain the invariant $0 \leq x \leq 10$ in Example 1.

The central problem in the above steps is how to solve the constraint $\forall X, X'.\Phi(X, X', Y)$ with universal quantifiers. Despite the research progress, decision procedures for quantified formulas can be fragile and may become the performance bottleneck of the template-based approach.

To solve the constraint in template-based invariant generation, most existing studies employ domain-specific reduction techniques to remove the universal quantifiers. These reduction methods typically involve lightweight rewriting procedures such as Farkas' Lemma for linear arithmetic [7], and Ackermann's reduction for uninterpreted functions [43].[3] After removing the universal quantifiers, we can solve the resulting quantifier-free formulas and obtain the invariant.

Remark 1: A relevant line of research is SyGuS-based invariant generation, which typically constrains the syntactic structures of invariants by utilizing either a user-supplied grammar [39] or an automatically generated one [48]. Various approaches are employed by existing SyGuS solvers, such as machine learning and enumerative synthesis [39]. In comparison, the template-based approach is *relatively complete* for certain first-order theories and invariant templates; however, the completeness guarantee is absent in most existing SyGuS-based invariant generation techniques.

III. MOTIVATION

In this section, we first highlight the benefits of adapting the template-based approach to bit-vector programs (§ III-A) and then discuss the obstacles to the adaption (§ III-B).

A. Template-based Invariant Generation for Bit-Vectors

The advances in SMT solving have enabled significant progress in automated program verification. Among the various theories, fixed-sized bit-vector arithmetic holds particular importance. Table II lists a typical set of operations in this theory, which allow for faithfully encoding machine integer semantics, such as non-linear computations, bitwise operations, and wrap-around behaviors.

Although the template-based approach to invariant generation has been extensively studied for integer or real arithmetic, there has been a notable lack of investigation into its application for bit-vector arithmetic. In this paper, we consider the problem of template-based invariant generation for bit-vector programs, which can offer several significant advantages:

- First, the approach is *relatively complete* concerning a wide variety of invariant templates, owing to the

[3] An alternative strategy is to use quantifier elimination. However, quantifier elimination is usually computationally expensive, while these rewriting procedures are lightweight.

TABLE II: A set of bit-vector operators and their corresponding SMT-LIB-2 syntax.

Symbol	SMT-LIB Syntax
$=, <_u, >_u, <_s, >_s$	=, bvult, bvugt, bvslt, bvsgt
$\sim, -$	bvnot, bvneg
$\&, \|, <<, >>, >>_a$	bvan, bvor, bvshl, bvlshr, bvashr
$+, \cdot, mod, div$	bvadd, bvmul, bvurem, bvudiv
$\circ$	concat
$[u : l]$	extract

decidability of the quantified bit-vector theory [49]. By "relatively complete", we mean that if there exists an invariant that is expressible within the template and can verify a program, then the approach can find the invariant. Such a guarantee of completeness is rarely provided in conventional abstract interpretation [31, 32].

- Second, similar to CEAGR, the approach is *goal-directed* in that it computes only the invariants that are necessary for proving the program's correctness. The approach can combine the information from the pre-condition, program, and post-condition in one constraint, which is solved via a constraint solver.[4]
- Third, the approach does not necessitate the use of advanced SMT solver features like interpolant generation, which is not widely available for the bit-vector theory. In comparison, many other verification algorithms such as IC3/PDR [50, 51] and trace abstraction [52] typically rely on efficient interpolant generation, posing a common obstacle in utilizing them for bit-vector programs.

B. Obstacles to Adopting the Template-based Approach

Effectiveness of Invariant Templates. As previously mentioned, the existing studies on the template-based approach have primarily focused on integer or real arithmetic. Despite the benefits discussed in § III-A, there has been comparatively limited research on template-based invariant generation for bit-vector programs. While it is possible to modify existing templates to accommodate bit-vector arithmetic, the effectiveness of this adaptation lacks empirical evidence.

Comparison of Different Invariant Templates. On one hand, the accuracy and computational cost of different invariant templates can vary. While some templates offer higher precision, they may require a greater computational cost, and vice versa. The trade-offs between precision and performance in template-based bit-precise verification are not yet well understood. Additionally, the current template-based approach mainly focuses on conjunctive invariants, which may not provide sufficient precision for verification purposes. To enhance precision, various strategies exist, such as using disjunctive templates. Disjunctive invariants are important for capturing program characteristics such as mode transitions and multiple phases [53, 54]. However, it is uncertain how these improvements could elevate the capabilities of invariant templates for verifying bit-vector programs.

Comparison with other Invariant Generation Techniques. On the other hand, it is important to explore how the template-based approach for bit-vector programs compares to other existing verification techniques, such as CEGAR [8, 33, 55], IC3/PDR [9, 34–37], and syntax-guided synthesis [38, 39]. Indeed, several efforts have been made to extend these verification techniques to bit-vector arithmetic. By comparing the template-based approach with other invariant generation techniques, we can gain a comprehensive understanding of their respective strengths and limitations in terms of precision and scalability.

Scalability of Constraint Solving. As discussed in § II-B, existing studies following the template-based approach often rely on domain-specific reductions to efficiently remove universal quantifiers in Equation (2), such as Farkas' Lemma for linear arithmetic [7] and Ackermann's reduction for uninterpreted functions [43]. However, these lightweight reductions do not apply to bit-vector arithmetic. Consequently, we have to solve the quantified bit-vector formula, i.e., $\forall X, X'.\Phi(X, X', Y)$, where Y represents the set of template parameters. To date, there are three primary categories of approaches to solving such constraints.

Quantifier Instantiation Approach. First, we can utilize existing SMT solvers that are capable of handling quantifiers. Z3, for example, was the first SMT solver to support quantified bit-vector formulas [49]. Since then, there have been several advances in solving quantified bit-vector constraints [49, 56–58]. These solvers leverage different strategies for quantifier instantiation, repeatedly instantiating universally quantified variables in the Skolemized formula with ground terms until an unsatisfiable quantifier-free formula or a model of the original formula is obtained. One of the key benefits of the quantifier instantiation approach is that it can handle formulas with arbitrary quantifier alternations.

Bit-Blasting Approach. Second, we can reduce the bit-vector constraint to Boolean-level constraints, which can be modeled and solved using different ways.

1) Binary Decision Diagram (BDD): The quantifier-free part $\Phi(X, X', Y)$ of the formula can be translated into a BDD, and quantifiers can be handled using approximations [58];
2) QBF solving: The formula $\Phi(X, X', Y)$ can be bit-blasted into a Boolean formula, with the quantifiers added accordingly, resulting in a QBF formula. We can solve the QBF formula via QBF solvers.
3) SAT solving: The quantifiers in the QBF formula can be eliminated through quantifier elimination, resulting in an equi-satisfiable SAT formula.

It is important to note that the bit-blasting approach can significantly increase the size of the formula and may lead to the loss of structural information.

Iterative Synthesis Approach. Third, we can use an SMT-based, counterexample-guided inductive synthesis (CEGIS)-style algorithm that searches for assignments to Y [46, 59].

[4]Some studies on the template-based approach do not consider post-condition when encoding the constraint. Thus, their implementations are not goal-directed.

At a high level, this approach involves a CEGIS loop, where a learner finds solutions to the template variables Y, and a verifier checks the solutions and produces counterexamples of invalid candidates. This process continues until a sufficient assignment is found or the constraint is proven to be unsatisfiable. There are two noteworthy points. First, both the learner and verifier rely on an SMT solver to solve quantifier-free bit-vector formulas. Unlike the quantifier instantiation approach, the SMT solver itself does not need to handle quantifiers. Second, the CEGIS-based approach is complete for bit-vector constraints because the search space is finite.

However, the above three categories of approaches have primarily been developed independently for application domains other than template-based invariant generation, or for general forms of quantified bit-vector formulas (as opposed to our specific setting of $\forall X, X'.\Phi(X, X', Y)$). As such, their comparative effectiveness in our context remains unclear. To bridge the gap, we aim to explore benchmarking these approaches to gain a better understanding of their respective strengths and limitations.

IV. Experimental Study Design

In this paper, we aim to understand the accuracy and efficiency of the template-based approach for verifying bit-vector programs by investigating the following research questions:

- **RQ1:** How does the effectiveness of the template-based approach vary in generating bit-vector program invariants under different template types and solving methods?
  - **RQ1.1**: How do different linear invariant templates perform in verifying bit-vector programs?
  - **RQ1.2**: How do the precision-enhancing strategies impact the precision of the approach?
  - **RQ1.3**: How do different constraint solving strategies perform in handling the quantified bit-vector constraints from the template-based approach?
- **RQ2:** How does the template-based approach perform, in terms of accuracy and time cost, compared to other invariant generation techniques?

To answer these questions, we have developed EAGLE, an invariant generation tool for bit-vector programs. EAGLE takes as input verification problems encoded using Constraint Horn Clauses (CHC) or SyGuS 2.0 languages, which are commonly used as intermediate representations for verification.

We proceed by providing the implementation details of EAGLE, including the invariant templates (§ IV-A) and constraint solving engines (§ IV-B). Following this, we discuss the third-party invariant generation tools used in our experiments (§ IV-C). Lastly, we explain the details of our experimental setup (§ IV-D).

A. *Invariant Templates in* EAGLE

In the template-based approach, the expressiveness of the obtainable invariants is determined by the templates used. In this section, we first present the basic invariant templates used in our study, followed by two strategies for potentially improving the precision of these basic templates.

Linear Templates. EAGLE supports several template linear domains that have been extensively studied in the literature.

- The interval domain [1] is a widely-used, non-relational domain about single-variable-inequalities.
- The zone domain [60] is a relational domain supporting predicates of the form $x - y \leq c$ where x and y are variables, and c is a constant.
- The octagon domain [61] is a relational numerical abstract domain that supports binary inequalities of the form $ax - by \leq c$, where x and y are variables, $a, b \in \{-1, 0, 1\}$ are coefficients, and c is the bound of the inequality.
- The polyhedron domain [2] is a relational domain used to approximate linear inequalities in the form of $a_0 + a_1 * x_1 + \cdots + a_n * x_n \leq 0$, where $x_1, \ldots, x_n$ are variables, and $a_0, \ldots, a_n$ are coefficients.[5]

All of the above templates can be regarded as elements in the family of Template Constraint Matrix (TCM) domains. In our setting, the variables in the invariant templates are bit-vectors representing finite-size integers.

Improving Precision with Disjunctive Completion. Disjunctive completion is a well-known technique in abstract domain refinement that enhances an abstract domain to make it disjunctive. Disjunctive invariants are important in capturing various program characteristics, such as mode transitions, multiple phases, and other disjunctive features of programs. To generate disjunctive invariants, we follow the idea of bounded disjunctive domains in abstract interpretation [53]. This involves fixing the number of disjuncts in the invariant. For instance, instead of using the interval template $F \equiv l \leq x \leq u$, we can introduce its disjunctive variant $F' \equiv l_1 \leq x \leq u_1 \vee \cdots \vee l_k \leq x \leq u_k$, which consists of k disjuncts.

It is important to note that when employing a disjunctive template, both the number of template parameters (e.g., $\{l_1, u_1, \ldots, l_k, u_k\}$) and the size of the generated constraint increase proportionally with the value of k. Consequently, this would increase the overhead of the constraint solver.

Improving Precision via Property Strengthening. In addition to improving precision through disjunction, we have developed a simple optimization strategy called *property strengthening*, which has the potential to enhance the precision of a given template. Our idea follows previous research that rephrases the invariant synthesis problem as strengthening the post-condition [62, 63]. Existing techniques typically generate new lemmas iteratively to strengthen the post-condition, employing methods such as machine learning, interpolation, or abductive inference [62]. However, most of these techniques are designed for linear integer or real arithmetic and are not directly applicable in our context.

Our property strengthening strategy directly uses the subformulas in the translated constraint (following Equation (1)). To illustrate this approach, let us consider an original template $F \equiv a \leq x \leq b$. Instead of generating invariants

[5] Note that in our evaluation, we only use a single linear inequality as the template. However, in general, the polyhedron domain can allow for multiple linear inequalities.

TABLE III: Evaluated invariant templates, optimization strategies, and their abbreviations.

Basic template	Abbreviation
Interval template	int
Zone template	zone
Octagon template	oct
Polyhedron template	poly
Optimization strategy	**Abbreviation**
Disjunctive completion	DC
Property strengthening	PS

TABLE IV: Constraint constraint solving engines.

Approach	Constraint Solving Engine
Quantifier Instantiation	QI-Z3, QI-Bitwuzla, QI-CVC5, QI-Yices2
Bit Blasting	BDD-based: BB-Q3B QBF-based: BB-caqe SAT-based: BB-CaDiCaL, BB-Glucose, BB-Lingeling
Iterative Synthesis	IS-Z3, IS-CVC4, IS-MathSAT, IS-Yices2, IS-Boolector

using this template, we can choose a strengthened template $F' \equiv a \leq x \leq b \wedge (\neg G(X) \rightarrow Post(X))$, where $G(X)$ is the loop condition and $Post(X)$ is the post-condition. Importantly, this approach involves only syntactic changes to the template and does not require additional logical reasoning for the strengthening process.

Similar to the disjunctive completion strategy, the property strengthening strategy increases the size of the translated constraint. However, it does not introduce new variables to the constraint since F' uses the same set of variables as the original template F.

In summary, Table III provides an overview of the basic templates and optimization strategies used in our experiments. We will discuss the impact of these strategies in § V.

B. Constraint Solving Engines in EAGLE

In the template-based approach, the key enabling component is the decision procedure for solving the quantified bit-vector constraints $\forall X, X'.\Phi(X, X', Y)$, where Y represents the set of template parameters. Table IV lists all the constraint solving engines utilized in our study. As discussed in § III-B, EAGLE is configurable for using three categories of approaches.

Quantifier Instantiation Approach. In the SMT solving community, significant progress has been made in recent years in solving quantified bit-vector formulas [49, 56, 57]. While these solvers are capable of handling formulas with arbitrary quantifier alternations, our particular focus is on constraints of the form $\forall X, X'.\Phi(X, X', Y)$. For our evaluation, we have selected five state-of-the-art SMT solvers:

- Z3 [49] combines a set of effective word-level simplifications, and instantiates universally quantified variables with constants or subterms of the original formula.
- Boolector [56] uses a counterexample-guided synthesis approach for quantifier instantiation, generating a ground term for instantiation based on a predefined grammar.
- CVC5 [57] employs predetermined rules based on invertibility conditions to directly provide terms that can prune many spurious models, without using potentially expensive counterexample-guided synthesis.
- Bitwuzla [64] implements a combination of counterexample-guided quantifier instantiation and syntax-guided synthesis. It also employs a dual approach by applying the same technique to the negation of the input formula in separate threads.
- Yices2 [65] is based on the counterexample-guided quantifier instantiation (CEGQI) framework, using invertibility conditions [57] and generalization-by-substitution [66] for the generalization.

Bit-Blasting Approach. As discussed in § III-B, we can reduce the quantified bit-vector constraint $\forall X, X'.\Phi(X, X', Y)$ into Boolean-level constraints, which can be handled in different ways:

- BDD: We have chosen Q3B [58], the state-of-the-art BDD-based technique for solving quantified bit-vector formulas. Q3B employs approximations to handle quantifiers.
- QBF: We implement a bit-blaster that translates a quantified bit-vector formula into QBF. We have chosen caqe for its top ranking in the QBFEVAL competition.
- SAT: We use the quantifier elimination algorithm (the“qe2” tactic in Z3) to eliminate the quantifiers in the quantified formula The resulting SAT formula can be solved via several existing SAT solvers, including CaDiCaL, Glucose, Gluecard, Minicard, and MiniSat.

Iterative Synthesis Approach. Finally, we use PySMT to implement the CEGIS-style algorithm to solve the quantified bit-vector constraint [46, 59]. The algorithm involves a learner and a verifier that iteratively finds and validates assignments to the template parameters Y using SMT solvers. The performance of this approach heavily relies on the underlying SMT solver used by both the learner and verifier. Hence, our implementation allows for using different solvers as the SMT backend, including Z3, CVC4, MathSAT, Yices2, and Boolector.

C. Other Invariant Generation Techniques

We compare EAGLE against several state-of-the-art CHC and SyGuS solvers listed in Table V. These solvers are specifically designed to support bit-precise reasoning and utilize different verification techniques:

- Spacer: a CHC solver inside the Z3 SMT solver, which is based on IC3/PDR (property-directed reachability) and supports model-based projection for bit-vectors.
- Eldarica [8] [6]: a CHC solver based on CEGAR, which supports bit-vector interpolant generation [33] for abstraction refinement.
- CVC5Sy [39]: a SyGuS-based invariant generation engine inside the CVC5 SMT solver. It combines several novel bounded term enumeration strategies for enumerating candidate invariants.

[6]https://github.com/uuverifiers/eldarica

TABLE V: Compared third-party invariant generation tools.

Tool	Technique	Interface
Spacer	IC3/PDR (Property-Directed Reachability)	CHC
Eldarica	Interpolation-based CEGAR	CHC
CVC5Sy	Syntax-guided, enumerative synthesis	SyGuS

To ensure a fair comparison, we do not include other tools that target C/C++ programs, as they would require additional symbolic execution or verification condition generation to encode programs to symbolic constraints. Besides, we do not include other SyGuS-based tools (e.g., [48]), as they primarily focus on integer or real arithmetic and are not applicable to our specific setting.

D. Experimental Setup

Benchmarks. We have collected a set of 240 verification tasks from various sources to serve as benchmarks for our evaluation: (1) 214 instances from the authors of LoopInvGen,[7] who collect several benchmarks from SyGuS-COMP, SV-COMP, and other verification literature (e.g., [62, 67]); (2) 26 instances from [68], a state-of-the-art work for disjunctive invariant generation. These problems are specified via SyGuS(LIA), meaning that integer variables are treated as unbounded. We translate them to SyGuS(BV) and CHC(BV) formats to evaluate bit-vector invariant generation capability. Following the settings of the previous work [17, 62], we only consider safe benchmarks to focus on loop invariant inference instead of bug finding.

Environment. All experiments are conducted on a 128-core server with Intel(R) Xeon(R) Gold 6338 3.2GHz CPUs, 512 GB RAM, running Ubuntu 22.04. Following [68], we impose a time limit of 300 seconds for running each benchmark under a particular verifier.

Methodology. Our experiments proceed as follows. First, to answer RQ1.1, we compare the four basic templates. Second, to answer RQ1.2, we assess the effectiveness of two precision-enhancing strategies, namely disjunctive completion and property strengthening. Third, to answer RQ1.3, we study the performance of all the constraint solving engines in Table IV. Finally, to answer RQ2, we compare EAGLE against three different invariant generation approaches in Table V.

V. RESULTS AND ANALYSIS

In this section, we present the results of the experiments and discuss our findings.

A. Impact of Different Strategies (RQ1)

To explore the influence of different strategies, we examine the effectiveness of EAGLE under various configurations, including different invariant templates and constraint solvers. It is worth noting that the number of successfully verified instances can significantly vary depending on the configuration used. For RQ1.1 and RQ1.2, we present the results of three

[7]https://github.com/SaswatPadhi/LoopInvGen

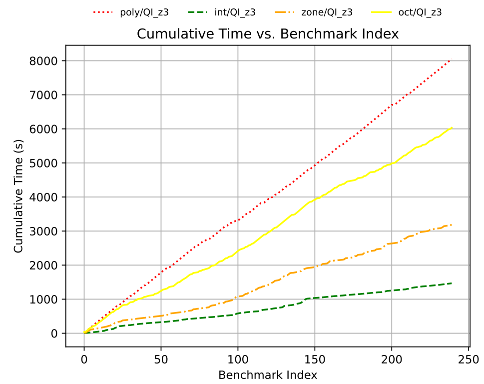


Fig. 2: Cumulative time of all benchmarks verified as **safe**.

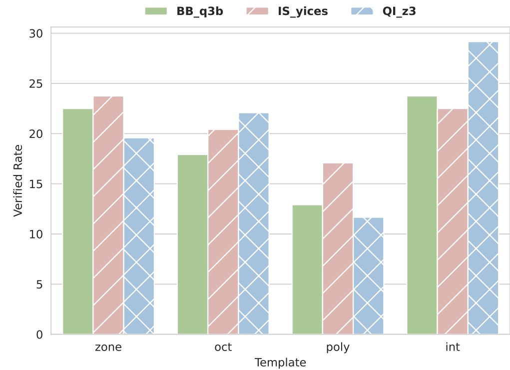


Fig. 3: Verified rates of using the four basic templates and the three best constraint solvers. The "Verified Rate" refers to the rate of benchmarks verified as safe.

solvers that perform relatively well in each category, namely QI-Z3 for quantifier instantiation, BB-Q3B for bit blasting, and IS-Yices2 for iterative synthesis. For RQ1.3, we provide a comprehensive analysis of the performances of various solvers.

Precision of Basic Templates (RQ1.1). We first evaluate the effectiveness of the four basic invariant templates, namely interval, zone, octagon, and polyhedron, without employing the two precision-enhancing strategies. The high-level results of the experiment are summarized in Figures 2 and 3, while more detailed results are presented in Table VI. Overall, we make the following observations.

First, overall, the interval template outperforms other templates for each of the three solvers. On average, it solves 8, 12, and 27 more instances than zone, octagon, and polyhedron templates, respectively. This finding is somewhat counter-intuitive since other relational domains are typically perceived as more precise than the interval domain. In conventional abstract interpretation, the phenomenon could be due to several reasons. For one thing, the abstract transformers of different domains may not necessarily be the most precise, leading to non-optimal invariants [3]. For the other thing, widening operations in fixed-point iteration can result in uncontrolled loss of precision [69]. However, in the template-based approach

TABLE VI: Comparison of EAGLE's variants that use different invariant templates. "DIS_k" represents disjunctive completion with k disjuncts. "PS" means property strengthening. "Verified" represents an instance is verified as safe. An item x, y in the "Failed" column represents that the solver either (1) returns "unsat" (which means the template is not expressive enough to verify the instance) on x instances, or (2) returns "unknown" abnormally (within a relatively short time) on y instances. "Timeout" simply means that the tool runs out of time limit. The configuration that verifies the most instances in each column is shaded.

Solver	Configuration	**Interval**			**Zone**			**Octagon**			**Polyhedron**		
		Verified	Failed	Timeout	Verified	Failed	Timeout	Verified	Failed	Timeout	Verified	Failed	Timeout
QI-Z3	Base	70	0,170	0	47	0,193	0	53	0,187	0	28	0,212	0
	DIS_2	63	0,177	0	36	0,204	0	20	0,220	0	25	0,215	0
	DIS_5	34	0,206	0	33	0,207	0	12	0,228	0	23	0,217	0
	DIS_{10}	26	0,214	0	29	0,211	0	8	0,232	0	26	0,214	0
	PS	99	0,141	0	93	0,147	0	74	0,166	0	73	0,167	0
	DIS_2+PS	87	0,153	0	77	0,163	0	25	0,215	0	68	0,172	0
	DIS_5+PS	45	0,195	0	62	0,178	0	12	0,228	0	64	0,176	0
	DIS_{10}+PS	28	0,212	0	50	0,190	0	12	0,228	0	64	0,176	0
BB-Q3B	Base	57	0,183	0	54	0,186	0	43	0,197	0	31	0,209	0
	DIS_2	38	0,202	0	28	0,212	0	6	0,234	0	31	0,209	0
	DIS_5	6	0,234	0	15	0,225	0	6	0,234	0	27	0,213	0
	DIS_{10}	2	0,238	0	16	0,224	0	5	0,235	0	21	0,219	0
	PS	86	0,154	0	100	0,140	0	58	0,182	0	73	0,167	0
	DIS_2+PS	38	0,202	0	58	0,182	0	8	0,232	0	68	0,172	0
	DIS_5+PS	10	0,230	0	27	0,213	0	8	0,232	0	56	0,184	0
	DIS_{10}+PS	5	0,235	0	24	0,216	0	5	0,235	0	48	0,192	0
IS-Yices2	Base	54	143,0	43	57	163,0	20	49	115,0	76	41	112,0	87
	DIS_2	62	104,0	74	47	156,0	37	32	70,0	138	34	53,0	153
	DIS_5	61	13,0	166	35	117,0	88	27	7,0	206	36	32,0	172
	DIS_{10}	46	0,0	194	36	91,0	113	19	0,0	221	38	24,0	178
	PS	81	92,0	67	109	81,0	50	72	79,0	89	90	33,0	117
	DIS_2+PS	87	71,0	82	100	58,0	82	46	42,0	152	88	9,0	143
	DIS_5+PS	83	8,0	149	80	31,0	129	45	3,0	192	88	6,0	146
	DIS_{10}+PS	74	0,0	166	71	17,0	152	27	0,0	213	88	6,0	146

that is relatively complete, we expected the interval template to have a weaker strength than other templates.

Upon examining the instances, we find that the reason for the superior performance of interval is due to the nature of the bit-vector theory. This theory faithfully models machine integer semantics, including integer overflow. However, the presence of arithmetic overflow may lead to the loss of precision. Specifically, expressions in other templates (e.g., $x + y$) can overflow, resulting in a value of $\top$ in the sense of abstract interpretation. While it is possible to prevent overflows in the templates by encoding additional constraints, such constraints restrict the search space and, thus, affect the completeness of the template-based approach.

Second, on average, the interval, zone, octagon, polyhedron templates only verify 60, 52, 48, and 33 instances, respectively. After manual inspection of the failed instances, we find that the limited expressiveness of the templates is the main reason for the failures. For example, the benchmarks from the multi-phase dataset [68] require disjunctive invariants. Besides, several benchmarks require invariants with bit-wise operations, which may not not expressible in the basic templates.

Effectiveness of Precision-Enhancing Strategies (RQ1.2). We continue to examine the effectiveness of the two optimization strategies (§ IV-A) for potentially enhancing the precision of the basic templates, by evaluating four variants of EAGLE:

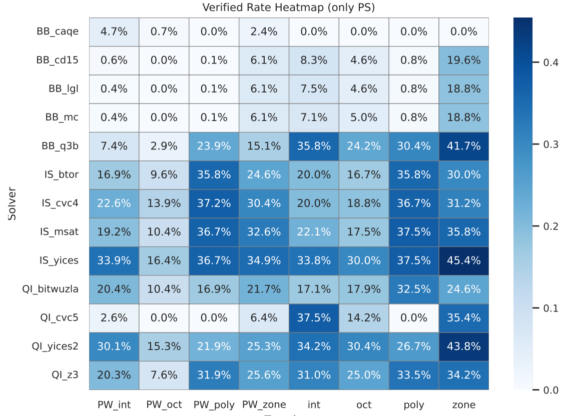
- EAGLE-Base: the default configuration that uses the basic templates (same as the previous experiment);
- EAGLE-DIS_k: the variant augmenting the basic templates via disjunctive completion, where k is the number of disjuncts in the disjunctive templates;
- EAGLE-PS: the variant that uses the property strengthening strategy;
- EAGLE-DIS_k + PS: the variant using both of the two optimization strategies. We first apply the disjunctive completion, followed by the property strengthening.

Table VI displays the number of verified instances for the four variants. The table showcases the results of three solvers that exhibit relatively strong performance across three categories of constraint solving approaches: quantifier instantiation (QI-Z3), bit blasting (BB-Q3B), and iterative synthesis (IS-Yices2). We have the following key observations.

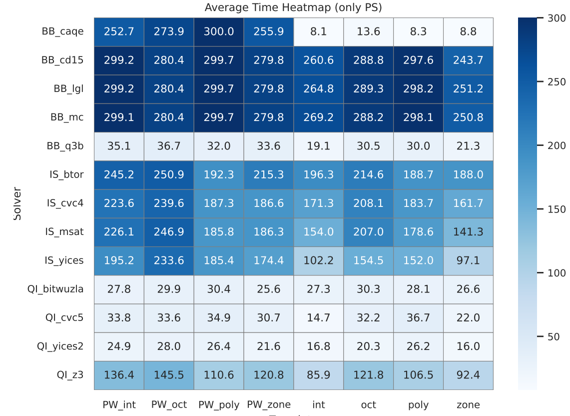
First, with the enhanced expressiveness of the templates, disjunctive completion can handle some specific cases (e.g., using the interval template and IS-Yices2 solver). However, this comes at the cost of efficiency. After using the strategy, EAGLE-DIS_2 verifies an average of 6 ,29 ,3 and 15 fewer instances for interval, octagon, polyhedron and zone, respectively than the Base. When the number of k increases, the overhead increases significantly. When setting $k = 10$, the number of verified instances reduce by $15.0\% - 77.9\%$, hampering the usability of the template-based approach.

Second, using the property strengthening strategy, EAGLE-PS verifies an average of 28, 48, 20, and 45 more instances for interval, zone, octagon, and polyhedron, respectively. The overhead incurred by using the property strengthening optimization is significantly less than that of the disjunctive completion strategy. The primary factor contributing to this difference is that although the property strengthening optimization does increase the size of the translated constraint, it does not introduce any additional variables into the constraint.

Third, choosing property strengthening and disjunctive



(a) Verified rate of the different solving strategies



(b) Average time of the different solving strategies

Fig. 4: Comparison of all the constraint solving engines and eight typical invariant templates with property strengthening (We omit the abbreviation "PS" in the figure).

completion together gains little improvement compared to using property strengthening alone. For example, EAGLE DIS_2 + PS solves fewer instances in almost all cases of the different templates and constraint solving engines as compared with PS.

Overall, we find that both the strategies can improve the successful rates of verification. The property strengthening strategy demonstrates better performance in our benchmarks. Besides, we remark that this paper only employs a simple inductive strengthening strategy that directly uses sub-formulas in the verification condition (§ II-B). We anticipate that the this optimization and its advanced variants be promising for the template-based verification of bit-vector programs.

Impact of Constraint Solving Engines (RQ1.3). Next, we compare the constraint solving strategies in Table IV. We use the four basic invariant templates and their property strengthening and disjunctive completion (with two clauses in disjunctive template) variants as a case study. Figure 4 summarizes the key metrics obtained from our evaluation, which present several performance trends.

First, in the bit-blasting category, the QBF and SAT-based solvers underperform compared to the solvers in the quantifier instantiation category (e.g., QI-Z3) and the iterative synthesis category (e.g., IS-Yices). For example, BB-caqe solves 23 instances using the interval template with disjunctive completion, which is fewer than most of the quantifier instantiation-based solvers. We find the the main performance bottleneck of the SAT-based solvers is the quantifier elimination algorithm, which can be much more expensive than SAT solving. However, we believe that there is significant room for improvement in the QBF-based solvers, as demonstrated by the competitive performance of the BDD-based solver Q3B [58], which uses several simplifications and approximations specially designed for quantified bit-vectors.

Second, although the quantifier instantiation approach demonstrates favorable performance on numerous instances, it is worth noting that we have observed certain limitations. For instance, in several formulas, QI-Z3 returns an "unknown" result within a relatively short time, rather than attempting to solve until reaching the time limit. Despite the decidability of the theory of quantified bit-vectors, modern SMT solvers may exhibit such behavior due to sophisticated heuristics and implementation details. We are actively engaging with the developers to investigate the underlying causes, such as potential incompleteness bugs within the solver.

Third, in general, there is no definitive winner in terms of solving performance among the solvers. Certain solvers exhibit superior performance on certain benchmark instances. Overall, several solvers produce similar results (QI-Z3, QI-Yices2) for most of the templates. Thus, a promising direction is to automatically select the constraint solving engine according to problem's features.

B. *Comparison with Existing Invariant Generation Techniques (RQ2)*

In this section, we evaluate the effectiveness of EAGLE by comparing it against three different approaches in Table V, namely Spacer, Eldarica, and CVC5Sy. We use the results of the configuration of EAGLE that verifies most instances in the previous experiments, i.e, i.e., EAGLE-Zone-PS-IS-Yices2 (zone template augmented with property strengthening, using IS-Yices2 as the constraint solver). Table VII compares the number of verified and timeout instances. Figure 5 illustrates the distribution of verified instances among the tools. We summarize the key findings below.

First, EAGLE is capable of uniquely verifying 2, 46, 43 instances that cannot be verified by Eldarica, Spacer, and CVC5Sy, respectively. This demonstrates that the template-based approach used in EAGLE is effective and complements the existing invariant inference approaches of these tools.

Fig. 5: Venn diagram illustrating the distribution of verified instances among the four tools.

TABLE VII: Comparison of EAGLE-Zone-IS-QI-Yices2 and the other three invariant generation tools ("Failed" refers to parsing error, out of memory, crash, and so forth).

Prover	Verified	Failed	Timeout	Avg. time
CVC5sy	158	82	0	2.86
Eldarica	190	8	42	62.90
Spacer	92	14	134	182.10
EAGLE	109	87	44	97.08

Second, Eldarica has the highest number of instances verified among the four tools. Its good performance follows from the highly optimized features such as intepolation for bit-vectors [55].

Third, on average, CVC5Sy outperforms the other three tools in efficiency. Notably, nearly all benchmarks are either verified or prematurely terminated (designated as "unknown") within a short time, which is related to the specific implementation of the tool and is worth further investigation. CVC5sy combines several novel bounded term enumeration strategies for smart and fast enumeration of invariants. However, it is worth noting that the trade-off for its fast enumeration strategy is a potential sacrifice in completeness [39].

Overall, we believe that these techniques are complementary in terms of verification capability, and there is much room in improving the accuracy and efficiency of EAGLE. We discuss a few possible directions in § V-C.

C. Discussion

Summary of Evaluation. From our experiments, we have the following observations:

- Among the four basic templates, we have the counter-intuitive observation that the weakest template of intervals outperforms other templates. This may be due to the fact that overflows in other templates are not well-handled and simply over-approximated to $\top$.
- Among the template precision-enhancing strategies, we find that our simple property strengthening performs quite well in the experiments, while the disjunctive completion may not pay off. Since disjunction mostly timeout, this suggests that an important improvement is to devise better algorithms for disjunction.
- Among different constraint solving engines, we observe that the quantifier instantiation and the iterative syntheis approaches perform better than the bit-blasting approach. However, there is no single best constraint solver.
- Among the comparison with other invariant generation methods for bit-vectors, we find the effectiveness of EAGLE which uniformally handle 40+ instances compared to Spacer and CVC5Sy. Note that Eldarica performs better than ours, but it is highly optimized for bit-vector, while our approach is coarse and has not incorporated optimization yet.

Thus, our results suggest the effectiveness and efficiency of the template-based approach for verifying bit-vector programs.

Future Work. We discuss a few directions for advancing the template-based verification of bit-vector programs.

Optimizing Invariant Templates. Optimizing the design of templates is crucial because it directly impacts the precision of the obtained invariants in the template-based approach. First, the superior performance of interval templates suggests that a major enhancement would be to refine the treatment of the wrap-around behavior in templates other than intervals. Second, beyond the simple property strengthening optimization (§ IV-A), it would be promising to design new strategies for "refining" an existing template, e.g., borrowing the idea of iterative strengthening or weakening [62, 67]. Finally, in alignment with previous work, we require fixing a pre-defined template, which may limit expressiveness. Thus, automatically searching through the space of candidate templates is important for improving the usability of the approach.

Speeding up Constraint Solving. There are several avenues to explore for accelerating constraint solving. First, numerous developments have been made concerning the quantifier instantiation approach [49, 56–58]. Although designed for general quantified constraints, they have shown promising performance in our experiments. Nonetheless, they also exhibit certain undesirable behaviors such as returning an "unknown" result in a relatively short time. Therefore, to enhance the stability of the quantifier instantiation heuristics [70], further improvements should be pursued for bit-vector arithmetic. Second, regarding the bit-blasting approach, it is important to devise new simplifications and reductions, which can be applied before [71, 72], during [73], and after the bit-blasting phase. Third, for the iterative synthesis approach, optimizing both the learner and verifier is crucial. This can be achieved by developing better strategies for generating candidates (e.g., employing constrained sampling heuristics) and generalizing from counterexamples (e.g., using variants of model-based projection). Finally, a possible direction is to identify prominent features that influence the performance and choose the best constraint solving engine that match the features of a program [74].

Combination with other Verification Techniques. The template-based invariant generation approach can be effectively

combined with other verification techniques to address the limitations posed by the expressiveness of the templates. First, k-induction is a powerful methodology readily applicable to bit-vector constraints.[8] Several attempts have been made to enhance k-induction through the use of auxiliary invariants [46, 75, 76], which can be generated using various techniques, including the template-based approach. Second, there have been a few studies that combine the template-based approach and predicate abstraction [77, 78], which could be adapted to bit-vector programs. Third, a recent work [10] combines recurrence analysis and template-based approach, building on real arithmetic. A promising direction to explore is to adapt this technique for bit-vector arithmetic. One possible approach involves translating bit-vector constraints into linear constraints using the techniques in [79], and subsequently applying the technique in [10].

Modeling other Types of Programs. In addition to modeling machine integers, the bit-vector theory can also be utilized to model other types of language constructs. First, it is well-known that we can use bit-vectors to encode floating-point numbers and bounded-length strings [80]. Second, to handle uninterpreted functions and arrays, we may apply the techniques proposed in [43, 49] to eliminate these constructs and subsequently use the established methods for bit-vector arithmetic. By employing these techniques, we can extend the applicability of the template-based approach to a broader range of programs and verification scenarios.

VI. Related Work

Template-based Invariant Generation. This approach begins with a template with unknown quantities and finds invariants by solving constraints on the unknowns. Most existing methods operate over integer or real arithmetic, either for generating linear invariants [5–7, 18–21, 81, 82] or non-linear invariants [22–26, 26–30]. To date, there has been limited research on extending the template-based approach to bit-vector programs. Besides, existing studies rely on domain-specific reductions (such as Farkas' Lemma for linear arithmetic [7] and Ackermann's reduction for uninterpreted functions [43]) to simplify the constraints. However, these reductions are not applicable to bit-vectors. Consequently, solving quantified bit-vector constraints becomes necessary. In this work, we perform a comprehensive review of existing techniques for solving such constraints. Furthermore, we present the first quantitative study on the performance of different constraint solving strategies.

Bit-Precise Invariant Generation. Several SMT-based model checking algorithms have been extended to support bit-vector arithmetic, such as IC3/PDR [9, 34–37] and CEGAR [8, 33, 55]. These algorithms often rely on interpolant generation, e.g., for approximating image computation and generating lemmas for refinement. However, interpolation for bit-vectors is not well-supported in many modern SMT solvers. The abstract interpretation community has proposed various abstract domains for bit-vector arithmetic, such as the strided intervals [83] and the wrapped intervals [84]. Unfortunately, abstract interpretation is often incomplete. SyGuS has also been used to generate inductive invariants using either a user-supplied grammar [39] or an automatically generated one [48]. However, existing SyGuS-based algorithms mainly focus on integer or real arithmetic.

Complete Abstract Interpretation. The template-based invariant generation has a connection with the problem of making abstract interpretation complete [31, 85–87]. This problem was initially proposed by Cousot and Cousot [88]. Giacobazzi et. al. [31] provide a constructive characterization of complete abstract interpretation in the general case. Reps et. al. [3] introduce the problem of *symbolic abstraction*, which computes the best (most precise) abstraction of a formula in a given abstract domain [3, 89, 90]. Under certain conditions, symbolic abstraction can produce the most precise inductive invariants, thereby facilitating relative completeness. Existing symbolic abstraction algorithms, however, are not goal-directed. In comparison, the template-based approach may not necessarily compute the most precise invariants, as it only needs to compute a sufficient one for completing the proof.

Empirical Evaluation of Program Analyzers. There have been several empirical evaluations of different program analysis techniques, such as taint analysis [91–93], k-induction [94], and IC3/PDR [95]. However, to the best of our knowledge, there is a lack of systematic evaluation specifically focusing on the template-based invariant generation for bit-vector programs. In this work, we present empirical evidence on the effectiveness of the adaption and study the precision and performance trade-offs in this setting.

VII. Conclusion

The template-based approach to invariant generation has been widely studied for integer and real arithmetic. In this work, we present the first comprehensive investigation into its effectiveness when applied to bit-vector programs. We examine the performance of different templates and constraint solving strategies, as well as the effectiveness of the template-based approach compared to alternative techniques. Our work presents quantitative evidence on the promises of template-based verification for bit-vector programs and suggests potential avenues for future research.

Acknowledgment

We thank the anonymous reviewers for the valuable feedback. The authors are supported by the National Key R&D Program of China (under Grant No. 2022YFB4501903), the National Natural Science Foundation of China (under Grant No. 62172271, 62272400, and 62132014), and the Qizhen Scholar Foundation of Zhejiang University. Hongfei Fu is the corresponding author.

References

[1] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.

[8] Usually, implementing k-induction only requires a decision procedure for checking the satisfiability of quantifier-free formulas.

[2] P. Cousot and N. Halbwachs, “Automatic discovery of linear restraints among variables of a program,” in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL’78)*.
[3] T. Reps, M. Sagiv, and G. Yorsh, “Symbolic implementation of the best transformer,” in *Verification, Model Checking, and Abstract Interpretation (VMCAI’04)*.
[4] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, “Symbolic optimization with smt solvers,” in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’14)*.
[5] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, “Linear invariant generation using non-linear constraint solving,” in *International Conference on Computer Aided Verification (CAV’03)*.
[6] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, “Constraint-based linear-relations analysis,” in *International Static Analysis Symposium (SAS’04)*.
[7] ——, “Scalable analysis of linear systems using mathematical programming,” in *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’05)*.
[8] H. Hojjat and P. Rümmer, “The eldarica horn solver,” in *2018 Formal Methods in Computer Aided Design (FMCAD’18)*.
[9] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “IC3 modulo theories via implicit predicate abstraction,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’14)*.
[10] J. Breck, J. Cyphert, Z. Kincaid, and T. W. Reps, “Templates and recurrences: better together,” in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI’20)*.
[11] Z. Kincaid, J. Cyphert, J. Breck, and T. W. Reps, “Non-linear reasoning for invariant synthesis,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 2018.
[12] L. Kovács and T. Jebelean, “Automated generation of loop invariants by recurrence solving in theorema,” in *Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC’04)*.
[13] Z. Kincaid, J. Breck, A. F. Boroujeni, and T. W. Reps, “Compositional recurrence analysis revisited,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’17)*.
[14] X. Si, A. Naik, H. Dai, M. Naik, and L. Song, “Code2inv: A deep learning framework for program verification,” in *International Conference on Computer Aided Verification (CAV’20)*.
[15] P. Garg, C. Löding, P. Madhusudan, and D. Neider, “Ice: A robust framework for learning invariants,” in *International Conference on Computer Aided Verification (CAV’14)*.
[16] R. Sharma, A. V. Nori, and A. Aiken, “Interpolants as classifiers,” in *International Conference on Computer Aided Verification (CAV’12)*.
[17] R. Xu, F. He, and B.-Y. Wang, “Interval counterexamples for loop invariant learning,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE’20)*.
[18] S. d. Oliveira, S. Bensalem, and V. Prevosto, “Synthesizing invariants by solving solvable loops,” in *International Symposium on Automated Technology for Verification and Analysis (ATVA’17)*.
[19] K. Chatterjee, P. Novotny, and D. Zikelic, “Stochastic invariants for probabilistic termination,” in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL’17)*.
[20] J.-P. Katoen, A. K. McIver, L. A. Meinicke, and C. C. Morgan, “Linear-invariant generation for probabilistic programs,” in *International Static Analysis Symposium (SAS’10)*.
[21] A. Gupta and A. Rybalchenko, “Invgen: An efficient invariant generator,” in *International Conference on Computer Aided Verification (CAV’09)*.
[22] D. Kapur, “Automatically generating loop invariants using quantifier elimination,” in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.
[23] K. Chatterjee, H. Fu, A. K. Goharshady, and E. K. Goharshady, “Polynomial invariant generation for non-deterministic recursive programs,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’20)*.
[24] Y.-F. Chen, C.-D. Hong, B.-Y. Wang, and L. Zhang, “Counterexample-guided polynomial loop invariant generation by Lagrange interpolation,” in *Proceedings of the International Conference on Computer Aided Verification (CAV’15)*.
[25] E. Hrushovski, J. Ouaknine, A. Pouly, and J. Worrell, “Polynomial invariants for affine programs,” in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS’18)*.
[26] A. Humenberger, M. Jaroschek, and L. Kovács, “Automated generation of non-linear loop invariants utilizing hypergeometric sequences,” in *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation (ISSAC’17)*.
[27] Y. Feng, L. Zhang, D. N. Jansen, N. Zhan, and B. Xia, “Finding polynomial loop invariants for probabilistic programs,” in *International Symposium on Automated Technology for Verification and Analysis (ATVA’17)*.
[28] S. d. Oliveira, S. Bensalem, and V. Prevosto, “Polynomial invariants by linear algebra,” in *International Symposium on Automated Technology for Verification and Analysis (ATVA’16)*.
[29] A. Adjé, P.-L. Garoche, and V. Magron, “Property-based polynomial invariant generation using sums-of-squares optimization,” in *International Static Analysis Symposium (SAS’15)*.
[30] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, “Non-linear loop invariant generation using gröbner bases,” in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL’04)*.
[31] R. Giacobazzi and F. Ranzato, “Completeness in abstract interpretation: A domain perspective,” in *International Conference on Algebraic Methodology and Software Technology*. Springer, 1997.
[32] R. Giacobazzi, F. Ranzato, and F. Scozzari, “Making abstract interpretations complete,” *Journal of the ACM (JACM’00)*.
[33] P. Backeman, P. Rümmer, and A. Zeljić, “Interpolating bit-vector formulas using uninterpreted predicates and presburger arithmetic,” *Formal methods in system design (FMSD’21)*.
[34] Y.-S. Ho, A. Mishchenko, and R. Brayton, “Property directed reachability with word-level abstraction,” in *2017 Formal Methods in Computer Aided Design (FMCAD’17)*.
[35] H. Zhang, A. Gupta, and S. Malik, “Syntax-guided synthesis for lemma generation in hardware model checking,” in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’21)*.
[36] A. Goel and K. Sakallah, “Model checking of verilog RTL using IC3 with syntax-guided abstraction,” in *NASA Formal Methods Symposium (NFM’19)*.
[37] S. Lee and K. A. Sakallah, “Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction,” in *International Conference on Computer Aided Verification (CAV’14)*.
[38] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen, “Counterexample guided inductive synthesis modulo theories,” in *Computer Aided Verification: 30th International Conference (CAV’18)*.
[39] A. Reynolds, H. Barbosa, A. Nötzli, C. Barrett, and C. Tinelli, “cvc4sy: smart and fast term enumeration for syntax-guided synthesis,” in *Computer Aided Verification: 31st International Conference (CAV’19)*.
[40] H.-Y. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter, “Bit-precise procedure-modular termination analysis,” *ACM Transactions on Programming Languages and Systems (TOPLAS’17)*.
[41] S. Falke, D. Kapur, and C. Sinz, “Termination analysis of imperative programs using bitvector arithmetic,” in *Verified Software: Theories, Tools, Experiments (VSTTE’12)*.
[42] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger, “Ranking function synthesis for bit-vector relations,” *Formal methods in system design (FMSD’13)*.
[43] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, “Invariant synthesis for combined theories,” in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’07)*.
[44] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio, “SMT-based array invariant generation,” in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’13)*.
[45] J. Katoen, A. McIver, L. Meinicke, and C. C. Morgan, “Linear-invariant generation for probabilistic programs: - automated support for proof-based methods,” in *Static Analysis: 17th International Symposium (SAS’10)*.
[46] M. Brain, S. Joshi, D. Kroening, and P. Schrammel, “Safety verification and refutation by k-invariants and k-induction,” in *Static Analysis: 22nd International Symposium (SAS’15)*.
[47] S. Gulwani, S. Srivastava, and R. Venkatesan, “Program analysis as constraint solving,” in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’08)*.
[48] G. Fedyukovich, S. J. Kaufman, and R. Bodik, “Sampling invariants from frequency distributions,” in *2017 Formal Methods in Computer Aided Design (FMCAD’17)*.
[49] C. M. Wintersteiger, Y. Hamadi, and L. De Moura, “Efficiently solving

quantified bit-vector formulas," *Form. Methods Syst. Des. (FMSD'13)*.
[50] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*.
[51] A. R. Bradley, "Understanding IC3," in *International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*.
[52] M. Heizmann, J. Hoenicke, and A. Podelski, "Software model checking for people who love automata," in *International Conference on Computer Aided Verification (CAV'13)*.
[53] S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta, "Static analysis in disjunctive numerical domains," in *Proceedings of the 13th International Conference on Static Analysis (SAS'06)*.
[54] H. Peleg, S. Shoham, and E. Yahav, "D3: Data-driven disjunctive abstraction," in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'16)*.
[55] P. Backeman, P. Rummer, and A. Zeljic, "Bit-vector interpolation and quantifier elimination by lazy reduction," in *2018 Formal Methods in Computer Aided Design (FMCAD'18)*.
[56] M. Preiner, A. Niemetz, and A. Biere, "Counterexample-guided model synthesis," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*.
[57] A. Niemetz, M. Preiner, A. Reynolds, C. Barrett, and C. Tinelli, "Solving quantified bit-vectors using invertibility conditions," in *International Conference on Computer Aided Verification (CAV'18)*.
[58] M. Jonáš and J. Strejček, "Solving quantified bit-vector formulas using decision diagrams," in *International Conference on Theory and Applications of Satisfiability Testing (TACAS'16)*.
[59] S. Kong, A. Solar-Lezama, and S. Gao, "Delta-decision procedures for exists-forall problems over the reals," in *Computer Aided Verification: 30th International Conference (CAV'18)*.
[60] A. Miné, "A new numerical abstract domain based on difference-bound matrices," in *Proceedings of the Second Symposium on Programs As Data Objects*, 2001.
[61] A. Miné, "The octagon abstract domain," *Higher Order Symbol. Comput.*, 2006.
[62] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA'13)*.
[63] S. Padhi, R. Sharma, and T. Millstein, "Loopinvgen: A loop invariant generator based on precondition inference," *arXiv preprint arXiv:1707.02029*, 2017.
[64] A. Niemetz and M. Preiner, "Bitwuzla at the SMT-COMP 2020," *arXiv preprint arXiv:2006.01621*, 2020.
[65] S. Graham-Lengrand, "Yices-QS 2022, an extension of yices for quantified satisfiability," 2022.
[66] B. Dutertre, "Solving exists/forall problems with Yices," in *Workshop on satisfiability modulo theories*, 2015.
[67] S. Lin, J. Sun, H. Xiao, Y. Liu, D. Sanán, and H. Hansen, "Fib: squeezing loop invariants by interpolation between forward/backward predicate transformers," in *International Conference on Automated Software Engineering (ASE'17)*.
[68] D. Riley and G. Fedyukovich, "Multi-phase invariant synthesis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22)*.
[69] R. Sharma, A. V. Nori, and A. Aiken, "Bias-variance tradeoffs in program analysis," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*.
[70] K. R. M. Leino and C. Pit-Claudel, "Trigger selection strategies to stabilize program verifiers," in *Computer Aided Verification: 28th International Conference (CAV'16)*.
[71] A. Nadel, "Bit-vector rewriting with automatic rule generation," in *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'14)*.
[72] M. Jonáš and J. Strejček, "On simplification of formulas with unconstrained variables and quantifiers," in *International Conference on Theory and Applications of Satisfiability Testing (SAT'17)*.
[73] J. P. Inala, R. Singh, and A. Solar-Lezama, "Synthesis of domain specific cnf encoders for bit-vector solvers," in *Theory and Applications of Satisfiability Testing (SAT'16)*.
[74] M. N. Mansur, B. Mariano, M. Christakis, J. A. Navas, and V. Wüstholz, "Automatically tailoring abstract interpretation to custom usage scenarios," in *Computer Aided Verification: 33rd International Conference (CAV'21)*.
[75] D. Beyer, M. Dangl, and P. Wendler, "Boosting k-induction with continuously-refined invariants," in *Computer Aided Verification: 27th International Conference (CAV'15)*.
[76] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The jkind model checker," in *Computer Aided Verification: 30th International Conference (CAV'18)*.
[77] S. Gulwani, S. Srivastava, and R. Venkatesan, "Constraint-based invariant inference over predicate abstraction," in *Verification, Model Checking, and Abstract Interpretation, 10th International Conference (VMCAI'09)*.
[78] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*.
[79] T. Okudono and A. King, "Mind the gap: Bit-vector interpolation recast over linear integer arithmetic," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'20)*.
[80] R. Axelsson, K. Heljanko, and M. Lange, "Analyzing context-free grammars using an incremental SAT solver," in *Automata, Languages and Programming: 35th International Colloquium (ICALP'08)*.
[81] Y. Ji, H. Fu, B. Fang, and H. Chen, "Affine loop invariant generation via matrix algebra," in *Computer Aided Verification - 34th International Conference (CAV'22)*, S. Shoham and Y. Vizel, Eds.
[82] H. Liu, H. Fu, Z. Yu, J. Song, and G. Li, "Scalable linear invariant generation with Farkas' lemma," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, 2022.
[83] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Compiler Construction: 13th International Conference (CC'04)*.
[84] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Interval analysis and machine arithmetic: Why signedness ignorance is bliss," *ACM Transactions on Programming Languages and Systems (TOPLAS'15)*.
[85] R. Bruni, R. Giacobazzi, R. Gori, I. Garcia-Contreras, and D. Pavlovic, "Abstract extensionality: on the properties of incomplete abstract interpretations," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, 2019.
[86] F. Bonchi, P. Ganty, R. Giacobazzi, and D. Pavlovic, "Sound up-to techniques and complete abstract domains," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*.
[87] R. Bruni, R. Giacobazzi, R. Gori, and F. Ranzato, "A logic for locally complete abstract interpretations," in *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'21)*.
[88] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'79)*.
[89] A. Thakur and T. Reps, "A method for symbolic computation of abstract operations," in *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*.
[90] P. Yao, Q. Shi, H. Huang, and C. Zhang, "Program analysis via efficient symbolic abstraction," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, 2021.
[91] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*.
[92] L. Luo, F. Pauck, G. Piskachev, M. Benz, I. Pashchenko, M. Mory, E. Bodden, B. Hermann, and F. Massacci, "Taintbench: Automatic real-world malware benchmarking of Android taint analyses," *Empirical Software Engineering (EMSE'22)*.
[93] L. Luo, E. Bodden, and J. Spath, "A qualitative analysis of Android taint-analysis results," in *International Conference on Automated Software Engineering (ASE'19)*.
[94] O. M. Alhawi, H. Rocha, M. R. Gadelha, L. C. Cordeiro, and E. Batista, "Verification and refutation of c programs based on k-induction and invariant inference," *International Journal on Software Tools for Technology Transfer (STTT'21)*.
[95] D. Beyer and M. Dangl, "Software verification with PDR: Implementation and empirical evaluation of the state of the art," *arXiv preprint arXiv:1908.06271*, 2019.