Fast Bit-Vector Satisfiability

Peisen Yao
The Hong Kong University of Science and Technology, China
pyao@connect.ust.hk

Qingkai Shi
The Hong Kong University of Science and Technology, China
qshiaa@cse.ust.hk

Heqing Huang
The Hong Kong University of Science and Technology, China
hhuangaz@cse.ust.hk

Charles Zhang
The Hong Kong University of Science and Technology, China
charlesz@cse.ust.hk

ABSTRACT

SMT solving is often a major source of cost in a broad range of techniques such as the symbolic program analysis. Thus, speeding up SMT solving is still an urgent requirement. A dominant approach, which is known as the eager SMT solving, is to reduce a first-order formula to a pure Boolean formula, which is handed to an expensive SAT solver to determine the satisfiability. We observe that the SAT solver can utilize the knowledge in the first-order formula to boost its solving efficiency. Unfortunately, despite much progress, it is still not clear how to make use of the knowledge in an eager SMT solver. This paper addresses the problem by introducing a new and fast method, which utilizes the interval and data-dependence information learned from the first-order formulas.

We have implemented the approach as a tool called Trident and evaluated it on three symbolic analyzers (Angr, Qsym, and Pinpoint). The experimental results, based on seven million SMT solving instances generated for thirty real-world software systems, show that Trident significantly reduces the total solving time from 2.9× to 7.9× over three state-of-the-art SMT solvers (Z3, CVC4, and Boolector), without sacrificing the number of solved instances. We also demonstrate that Trident achieves the end-to-end speedups for three program analysis clients by 1.9×, 1.6×, and 2.4×, respectively.

CCS CONCEPTS

• **Theory of computation** → **Automated reasoning**; **Program analysis**.

KEYWORDS

Satisfiability modulo theory, SAT solving, program analysis

ACM Reference Format:
Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast Bit-Vector Satisfiability. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397378

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397378

1 INTRODUCTION

Satisfiability Modulo Theories (SMT) solving has undergone steep development during the last decade, enabling a wide range of practical applications, such as test case generation [20, 40], static bug finding [77, 86], and program repair [62]. SMT solvers decide the satisfiability of formulas over first-order theories. Among the many theories within the SMT-LIB initiative [3], the theory of bit-vector is of crucial importance for software analysis, due to its capability of faithfully and precisely modeling the bit-level behavior of the machine instructions [86].

Existing SMT solving algorithms can be classified as being lazy or eager [15, 51]. The latter is the predominant approach to solve bit-vector constraints [43]. As illustrated in Figure 1(a), an eager bit-vector solver first simplifies the input formula with the word-level simplification rules [1, 17], and then translates a bit-vector formula into an equisatisfiable Boolean formula via bit-blasting, which is finally solved by an SAT solver. In practice, the word-level simplification and bit-blasting phases are efficient. However, the SAT solving phase is often the performance bottleneck due to its NP-completeness [43, 50].

While a significant effort has been investigated in the lazy SMT solving [9, 22, 42, 75], to the best of our knowledge, there is little progress on improving the SAT solving algorithms for an eager bit-vector solver. Most existing effort either focuses on enhancing the word-level preprocessing phase via different simplifications and semi-decision procedures [34, 36, 44, 61], or aims to devise better encoders for bit-blasting [45, 57]. These methods attempt to reduce the overhead of SAT solving by translating the bit-vector formulas into smaller Boolean formulas. However, the SAT solver itself still uses the default strategy irrespective of the word-level problem characteristics, thus losing many optimization opportunities.

In this paper, we present an approach that significantly improves the bit-vector SMT solver by leveraging the word-level information, just as shown in Figure 1(b). Our key insight is that the word-level information inferred *before* bit-blasting can be preserved to boost the SAT solving phase. Specifically, a conventional SAT solver often works as follows: given a Boolean formula, the SAT solver checks its satisfiability by choosing a variable in the formula, assigning a truth value to it, simplifying the formula based on the assignment, and then recursively checking if the simplified formula is satisfiable.

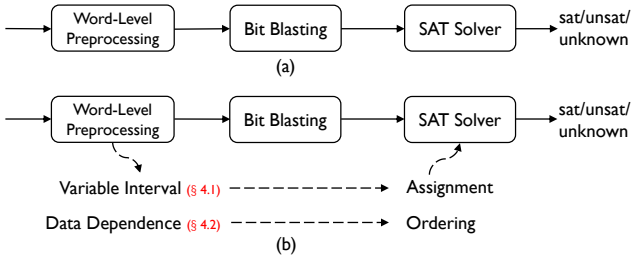


Figure 1: (a) Conventional workflow of eager SMT solving. (b) Our workflow of eager SMT solving.

If the search fails, the same recursive check is performed assuming the opposite truth value. Clearly, the performance of the SAT solver crucially depends on the *branching heuristic* [12, 47, 54, 59, 85], which concerns both the order in which Boolean variables are chosen for assignment and the values assigned to them. Therefore, we can utilize two categories of word-level information — data dependence and variable interval — to guide the decision order and the assignments, respectively.

We have implemented our approach as a tool called Trident on top of the Z3 SMT solver [29]. We evaluate Trident on three realistic symbolic analysis platforms (Angr [78], Qsym [88], and Pinpoint [77]), which generate a large set of, nearly 7.4 million, bit-vector constraints from thirty real-world software systems. The experimental results show that, compared to Z3, Trident achieves 3.4× to 6.3× speedup (with 4.9× on average). Trident also outperforms two other state-of-the-art SMT solvers, CVC4 and Boolector, achieving 7.9× and 2.9× speedup, respectively. Armed with our new bit-vector solver, the three symbolic analysis platforms, Angr, Qsym, and Pinpoint, achieves 1.9×, 1.6×, and 2.4× speedups, respectively. To sum up, we make the following main contributions in the paper:

- We introduce an approach to scaling eager bit-vector solvers, which leverages interval and data-dependence information to guide the branching heuristic in the SAT solving phase.
- We implement the proposed approach and apply our solver to three symbolic analysis applications, including control-flow recovery, test case generation, and static bug hunting.
- We conduct a thorough evaluation, confirming the effectiveness of our approach. We also provide a new set of publicly-available benchmarks with nearly seven million SMT solving instances, which can help better evaluate SMT solvers.

2 PRELIMINARIES

This section introduces the basic concepts and terminologies used in the paper.

2.1 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some first-order theories. Examples include the theories of linear arithmetics, bit-vectors, arrays, lists, and strings.

A bit-vector is a fixed sequence of bits. The theory of quantifier-free bit-vectors (QF_BV) is a many-sorted first-order theory. The length of a bit-vector is usually referred to as bit-width, and bit-vectors with different widths correspond to different sorts. The functions in the theory include $+, -, \times, \div, \&, |, \oplus, \ll, \gg$, *concat*, and *extract*, interpreted as addition, minus, multiplication, division, bit-wise and, bit-wise or, bit-wise exclusive or, left-shift, right-shift, concatenation, and extraction, respectively. The predicates include $=$, $<$, and $\leq$, which are interpreted as "equal to", "less than", and "less than or equal to", respectively. For the ease of presentation, we assume that all bit-vector variables represent unsigned integers. Deciding the satisfiability of a quantifier-free bit-vector formula is NP-complete, or more exactly NEXPTIME-complete [50].

2.2 Eager Bit-Vector Solving

Existing SMT solving algorithms can be classified as being *lazy* or being *eager*. Given a first-order formula, a lazy approach [5, 17, 75] usually iteratively refines an over-abstraction of the formula by combining a SAT solver and a theory solver. First, it abstracts the first-order formula by replacing each atomic predicate[1] with a distinct Boolean variable, producing a formula called the *Boolean skeleton*. Then, it uses the SAT solver to iteratively enumerate models of the Boolean skeleton, and the theory solver to check these models for satisfiability. In contrast, an eager approach [36, 48] translates the first-order formula, in a single satisfiability-preserving step, into an equisatisfiable Boolean formula, which is delegated to a SAT solver for determining the satisfiability, as shown in Figure 1.

Example 2.1. Consider the bit-vector formula $\phi \equiv x \geq 3 \wedge (x = 0 \vee y = 4)$, where x and y are two bit vectors. A lazy SMT solver will abstract ϕ as a Boolean skeleton $b_1 \wedge (b_2 \vee b_3)$, where the Boolean variables b_1, b_2, and b_3 represent $x \geq 3, x = 0$, and $y = 4$, respectively. Suppose that its SAT solver first suggests a model $(b_1 = 1, b_2 = 1, b_3 = 0)$. The Boolean model is mapped back as conjunctions of atomic predicates $x \geq 3 \wedge x = 0 \wedge y \neq 4$, which is then checked by the theory solver. If the conjunctions are satisfiable, so is the original bit-vector formula. Clearly, the conjunctions are unsatisfiable, meaning that the Boolean model is spurious. Then, its SAT solver attempts to suggest another model of the Boolean skeleton. In the worst case, the SAT solver enumerates all models of the Boolean skeleton, which is expensive. In contrast, the eager approach translates ϕ into an equisatisfiable Boolean formula and only calls a SAT solver once to determine the satisfiability.

In the eager approach to SMT solving, the typical method for translating a bit-vector formula into an equisatisfiable Boolean formula is *bit-blasting*. The procedure encodes a bit-vector variable using a sequence of auxiliary Boolean variables, each of which represents a bit of the bit-vector variable. Bit-vector functions such as addition and multiplication are modeled using Boolean connectives in a way that mimics the hardware circuits of these functions [15, 51]. Note that in order to encode these functions more compactly, the bit-blaster also introduces a sequence of auxiliary Boolean variables to represent each *bit-vector term*, i.e., functions

[1] An atomic predicate is a Boolean-type expression without Boolean connectives such as $x + y = 3$.

Algorithm 1: Eager bit-vector solving

```
   Input: A bit-vector formula φ.
   Output: satisfiable or unsatisfiable.
1  φ' ← apply word-level simplifications to φ;
2  BF(φ') ← apply bit-blasting to φ';
3  BV(·) ← the set of Boolean variables in BF(φ');
4  return CDCL_SAT(BF(φ'));
5  Function CDCL_SAT(φ):
6      while true do
7          if there is an unassigned Boolean variable then
8              select and assign a variable from BV(·);
9              while BCP() == CONFLICT do
10                 (C, level) = conflict_analysis();
11                 if level < 0 then
12                     return unsatisfiable;
13                 else
14                     backtrack(level);
15                     add_clause(C);
16         else
17             return satisfiable;
```

applied to a set of bit-vector variables.[2] In the remainder of the paper, we denote by $BF(\phi)$ the Boolean formula encoding a bit-vector formula ϕ. We denote a n-bits bit-vector variable or term t as $t = [b_{n-1}, b_{n-2}, \ldots, b_0]$, where $BV(t) = \{b_{n-1}, b_{n-2}, \ldots, b_0\}$ is the set of Boolean variables encoding t. Since we assume that bit-vectors are unsigned, we have

$$t = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_0 \times 2^0 , \qquad (1)$$

which means that once the values of Boolean variables $BV(t)$ are fixed, we can obtain the value of t, and vice versa.

Example 2.2. Let x and y be two 8-bits bit-vectors such that $x = [a_7, \ldots, a_0]$ and $y = [b_7, \ldots, b_0]$. Consider a term $x \mid y$, which is the "bit-wise or" of the variables x and y. To encode the term, the bit-blaster first introduces eight Boolean variables, $c_0, c_1, \ldots, c_7$, to represent the calculation result, i.e., $x \mid y = [c_7, \ldots, c_0]$, and then translates the term into the following Boolean formula that encodes the semantic of "bit-wise or":

$$\bigwedge_{i=0}^{7} ((a_i \vee b_i) \leftrightarrow c_i)$$

One purpose for introducing the auxiliary variables c_i is that we can reuse the translation of the term $x \mid y$ when the term is used many times in a constraint like $x \mid y + z > 10 \wedge x \mid y - z < 5$.

2.3 SAT Solving and Branching Heuristic

To determine the satisfiability of a Boolean formula generated by bit blasting, a Conflict-Driven Clause-Learning (CDCL) SAT solver will be employed [11, 80]. The function `CDCL_SAT` of Algorithm 1 presents the basic CDCL search loop. At each step, the branching heuristic picks an unassigned variable and assigns it a truth value of 1 or 0 (Line 8). The picked variable is called the decision variable. Then, the solver uses a method called Boolean Constraint Propagation (BCP) (Line 9) to simplify the formula, by leveraging the current assignment and its logical consequences. If the propagation leads to a falsified clause, a conflict occurs, indicating that a previous decision is not appropriate. The level[3] of that decision is identified by the conflict analysis (Line 10), following which the solver recovers from the conflict by backtracking, undoing the offending decision, and trying some other assignments. A clause learned from the conflict is also added to the original formula (Line 15), to prevent the search from repeating the mistake. The loop repeats until all clauses are satisfied, or some conflict cannot be resolved by backtracking and, thus, the formula is unsatisfiable (Line 12).

Example 2.3. Consider the Boolean formula $(a \vee b) \wedge (\neg a \vee b)$. If we pick the Boolean variable a as the decision variable and assign 1 to it, the BCP can infer than b should also be 1, because otherwise the clause $(\neg a \vee b)$ would be falsified.

Example 2.4. Consider the Boolean formula $(a) \wedge (\neg a \vee \neg b) \wedge (b \vee c)$. Suppose that we first choose a and assign 1 to it. The BCP can infer that b should be 0 and further deduce that c should be 1. All clauses are satisfied now. In total, only one decision is made. However, if we first choose c and assign 0 to it, the BCP can infer that b should be 1 and further infer that a should be 0. The current assignment $(c = 0, b = 1, a = 0)$ results in a falsified clause (a). The SAT solver has to perform backtracking, undoing the first decision and continuing a new round of search.

CDCL SAT solvers crucially depend on the branching heuristic for their performance. A state-of-the-art branching heuristics Variable State Independent Decaying Sum (VSIDS) [59] maintains a score for each Boolean variable throughout the search. It initializes the score based on the globally statistical information, such as the number of clauses in which a variable appears. At each step, VSIDS selects the variable with the highest score as the next decision variable. The scores are adjusted periodically by aggregating a variable's effects in the previous conflicts.

3 MOTIVATION

In this section, we use several examples to motivate our approach. We show that the bit-vector level information can be leveraged to partially determine the assignments (§ 3.1) and the order of decision variables (§ 3.2).

3.1 Assignment Restriction

In program analysis, to achieve the bit-level precision, an integer variable in a program is usually modeled as a bit-vector, of which the length is the bit width of the variable's type. The constraints in a program analysis consist of the operations and the relations among the bit-vector variables. In theory, a 32-bit unsigned bit-vector variable represents a large range of values, i.e., from 0 to $2^{32} - 1$. Besides, the number of variables in a path constraint can be huge. As the consequence, such a vast search space stresses the capability of the constraint solver, causing significant performance issues.

[2] More exactly, a term is inductively defined as either a variable or a function applied to any number of other terms. We differentiate between variable and term for the sake of presentation.

[3] The decision level of a variable is the number of decision variables occurring before the variable.

Fortunately, in practice, we find that the feasible solution space of the variables can be small. More specifically, we observe that, in real-world programs, not all statements are complex, e.g., containing non-linear computation and library function calls. Most variables are only involved in simple statements, such as linear assignments and linear branch conditions. By inspecting the constraints encoded for these simple statements, we can soundly approximate the solution space of the bit-vector variables and, thus, reduce the search space of SAT solving.

Example 3.1. Suppose that $x = [a_7, \ldots, a_0]$ and $y = [b_7, \ldots, b_0]$ encode two 8-bits unsigned integer variables and the formula $\phi \equiv y = x + 1 \wedge y < 5 \wedge x * x < 60$ is a constraint generated by a program analyzer. After bit-blasting the formula, we obtain a Boolean formula $BF(\phi)$. If we are able to infer that $x < 4$ from the linear constraint $y = x + 1 \wedge y < 5$ in ϕ, then we have that $a_7, a_6, \ldots, a_2$ must be 0, because otherwise x must be greater than or equal to 4. Therefore, we reduce the search space of SAT solving by fixing the values of six Boolean variables, i.e., $a_7, a_6, \ldots, a_2$.

Remark 1. Although the SAT solver can leverage Boolean Constraint Propagation (BCP) and backtracking to establish and exploit the correlations of Boolean variables automatically, it is unaware of the restriction about the variables before the search. If having an oracle that narrows down the search space of some bit-vector variables, we can reduce the number of Boolean variables to decide, thereby improving the SAT solving performance by reducing unneeded constraint propagation and backtracking.

3.2 Variable Ordering

Consider a formula with a set N of bit-vector variables, each having a value range of size k. In the worst case, the SAT solver needs to explore $k^{|N|}$ possible assignments. Like many other search problems, the order in which we select Boolean variables for assignments has an enormous effect on the performance of SAT solving. Specifically, we observe that the data dependence relations in a constraint can be leveraged. The insight here is that the presence of data dependence means that the values of a subset of variables deterministically derive the values of other variables.

Example 3.2. Let us consider the constraint ϕ in Example 3.1. According to the discussion in Section 2.2, before SAT solving, the constraint ϕ is translated to a Boolean formula consisting of Boolean variables, $BV(x)$, $BV(y)$, $BV(x + 1)$, and $BV(x * x)$. In the SAT solving phase, the SAT solver will select and assign values to these Boolean variables in some order. In the example, once $BV(x)$ are assigned, the values of $BV(x + 1)$, $BV(y)$, and $BV(x * x)$, can be derived automatically due to the data dependence relations. Thus, giving higher priority to the Boolean variables in $BV(x)$ than those in $BV(y)$, $BV(x + 1)$, and $BV(x * x)$ can accelerate the speed of cutting the search space.

Remark 2. Modern SAT solvers have sophisticated scoring schemes to decide the decision order. However, we find that state-of-the-art schemes like VSIDS are not sufficiently accurate at the beginning of the search. This is because they typically initialize the scores via statistical and syntactical information, which may introduce bias. Therefore, it would be beneficial to assist the solver in the initial phase by scheduling the decision order according to the semantic information such as the data dependence relations.

4 APPROACH

The branching heuristic in SAT solving concerns the order in which the Boolean variables are chosen for assignment and the values assigned to them. As shown in Figure 1(b), our techniques aim to guide it by leveraging two abstract domains over bit-vector variables: the non-relational interval domain and the relational data-dependence domain.

First, we utilize the interval information to fix the values of Boolean variables introduced during bit-blasting, thereby reducing the search space of SAT solving. After that, for variables that cannot be fixed yet, the solver still needs to decide their assignments. Second, to further accelerate SAT solving, we exploit the data-dependence information for guiding the decision order of Boolean variables.

The additional overhead of inferring the information is low: the runtime of the word-level preprocessing phase increases by about 15%. Such a price is acceptable because we can pay a small up-front cost to avoid a large amount of work in the later SAT solving phase. In what follows, we detail the two strategies for improving the performance of SAT solving.

4.1 Interval-Guided Variable Assignments

We first present the strategy of using the interval analysis to reduce the search space. Our observation is that in real-world programs, not all statements are complex. Thus, a constraint generated by a program analyzer usually contains many simple sub-constraints, which can be used to infer a sound interval of some bit-vector variables or terms, t, thereby restricting the values of their corresponding Boolean variables $BV(t)$.

However, given a bit-vector formula, acquiring the precise interval information of its variables is non-trivial. Specifically, computing the most precise interval of a variable is NP-hard, which can be reduced to Max-SMT problems [53]. For example, to obtain the maximum value of a bit-vector variable x subject to a formula $\phi(x, \ldots)$, we can solve the Max-SMT problem:

$$\begin{cases} \text{maximize} & x \\ \text{subject to} & \phi(x, \ldots)\,. \end{cases} \tag{2}$$

This is impractical because solving the optimization problem can be even harder than checking the satisfiability of the formula.

4.1.1 Interval Analysis. To balance the precision and performance, we employ a lightweight interval analysis, which takes a bit-vector formula as input, and determines a sound approximation, i.e., interval, of the numeric values for the bit-vector variables in a formula [37]. The details of the interval analysis are omitted, as it is direct and not our key contribution. For instance, given the constraint $z = x + y$, where $x \in [l_x, u_x]$, $y \in [l_y, u_y]$. According to the rules in the previous work [37], we have $z \in [l_x + l_y, u_x + u_y]$. In addition, given a constraint $z = x + y \wedge z = u - w$, we need to compute the intervals for $z = x + y$ and $z = u - w$, respectively, and then conjunct the two intervals to get an interval for the variable z.

Conventionally, let $R_1 = [l_1, u_1]$ and $R_2 = [l_2, u_2]$ be two intervals. The logical conjunction and disjunction operations are modeled with the meet (denoted $\sqcap$) and join (denoted $\sqcup$) operations, respectively:

$$R_1 \sqcap R_2 = \begin{cases} \bot & \text{, if } \max(l_1, l_2) > \min(u_1, u_2) \\ [\max(l_1, l_2), \min(u_1, u_2)] & \text{, otherwise} \end{cases}$$
$$R_1 \sqcup R_2 = [\min(l_1, l_2), \max(u_1, u_2)] \,. \tag{3}$$

For example, applying the join operator to the intervals, $R_1 = [1, 3]$ and $R_2 = [8, 11]$, produces a new interval $R' = [1, 11]$. Apparently, such a join operation will lose precision as it introduces a set of false values $\{4, 5, 6, 7\}$ within R'. In the remainder of this subsection, we focus on how to mitigate such precision loss so that the interval analysis can non-trivially improve the performance of SAT solving.

Disjunctive Domain with Lazy Join. One solution to the problem of precision loss is enriching the analysis with disjunction [68, 73]. For instance, instead of computing the new interval R', we record the interval as $R_1 \cup R_2$ in the above example to avoid the precision loss. Unfortunately, the number of intervals to maintain may grow extremely large, which is exponential in the number of disjunctions in the formula.

To restore the precision loss caused by joins but to avoid the expensive disjunctive abstraction, we employ a selective merging heuristic. The heuristic is responsible for determining whether two intervals should be merged with join or kept separately. The basic idea is that, if a join does not introduce much precision loss (i.e., yields a similar set of abstract states as taking their union), we can perform a join. To measure the precision loss quantitatively, we define a *dissimilarity ratio* σ ranging from 0 to 1. If the ratio is smaller than a threshold, we perform a join. Next, we discuss the method for computing the ratio.

First, we need to quantify the dissimilarity between the two intervals. Conventionally, the Hausdorff distance is a common measure of the discrepancy between two polyhedra A and B, defined as

$$H(X, Y) = \max_{x \in X}\{\min_{y \in Y}\{d(x, y)\}\}$$
$$\text{Hausdorff}(A, B) = \max\{H(A, B), H(B, A)\} \,, \tag{4}$$

where $d(x, y)$ is the Euclidean distance between two points x and y in Euclidean space.

The Hausdorff distance between two general polyhedra is hard to compute [6]. Fortunately, in our setting, A and B are both intervals. Suppose that we have $A = [l_1, u_1]$ and $B = [l_2, u_2]$. It can be proved that:

$$\text{Hausdorff}(A, B) = \max(|l_1 - l_2|, |u_1 - u_2|) \,. \tag{5}$$

Then, we can measure the proposition of dissimilarities in the new interval introduced after join. Specifically, we define the dissimilarity ratio as

$$\begin{aligned} \sigma &= \text{Hausdorff}(A, B) \div |A \sqcup B| \\ &= \max(|l_1 - l_2|, |u_1 - u_2|) \div (\max(u_1, u_2) - \min(l_1, l_2)) \,. \end{aligned} \tag{6}$$

Example 4.1. Consider the example in Figure 2. Suppose that we focus on merging intervals of the variable x. Comparing Figure 2(a) and Figure 2(b) where R_{1x} and R_{2x} are disjoint, we tend to perform a join for the intervals in Figure 2(a) because fewer false positives would be introduced. Comparing Figure 2(c) and Figure 2(d) where R_{1x} and R_{2x} overlap, we tend to perform a join for the intervals in Figure 2(c) because the two intervals have a larger overlap.

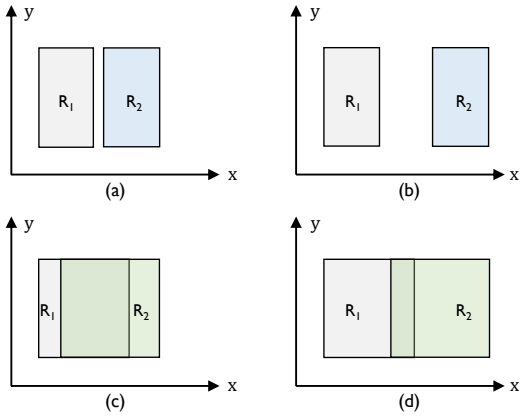


Figure 2: An example of selective merge with join operator.

Example 4.2. Consider the bit-vector formula $\phi \equiv (\varphi_1 \vee \varphi_2) \wedge (\varphi_3 \vee \varphi_4)$, where $\varphi_1, \varphi_2, \varphi_3$, and φ_4 are all conjunctions of atomic predicates. Suppose that we can infer from $\varphi_1, \varphi_2, \varphi_3$, and φ_4 that $R_1 = [3, 10], R_2 = [3, 10], R_3 = [3, 4]$, and $R_4 = [8, 11]$, respectively. We will merge the two intervals R_1 and R_2 with join because the dissimilarity ratio is 0. The merged interval R' is $R_1 \sqcup R_2 = [3, 10]$. In contrast, we tend not to merge R_3 and R_4 because by *Eq.* (6), the dissimilarity ratio is $7 \div 8$, close to 1. Intuitively, the merged interval $[3, 11]$ would introduce three false positives, i.e., $\{5, 6, 7\}$. Later, we need to propagate the interval R' to R_3 and R_4, i.e., compute $R' \sqcap R_3$ and $R' \sqcap R_4$, respectively.

4.1.2 Reducing Search Space. After obtaining the sound intervals of bit-vector variables, we then guide the branching heuristic by fixing the values for a subset of Boolean variables. For example, let x be a bit-vector variable representing a n-bits unsigned integer such that $x = [b_{n-1}, \ldots, b_0]$. Recalling *Eq.* (1), once the values of $b_{n-1}, b_{n-2}, \ldots, b_0$ are fixed, we can obtain the value of x, and vice versa. Therefore, if the interval analysis infers that $x \in [l, u]$, where $l > 0$ or $u < 2^n - 1$, then we fix some Boolean variables from $BV(x)$, prior to the main CDCL search loop. If the interval $[l, u]$ is precise enough, we can produce a dramatic size reduction of the space of the truth assignments searched in by the SAT solver.

An alternative use of the intervals might be adding them as additional constraints to the original formula. However, specifying the constrains can increase the formula size dramatically, when the number of variables is large and each variable has many disjunctive intervals. Consequently, the additional constraints may eventually increase the burden of SAT solving.

4.2 Dependence-Guided Variable Ordering

The interval analysis does not provide a panacea: there exist Boolean variables whose values cannot be fixed if the interval information is not precise enough. When solving the Boolean formula generated by bit-blasting, how to effectively handle those variables remains a problem. In particular, as demonstrated in Algorithm 1, the order

to assign Boolean variables decides the search direction and, thus, is crucial for the performance of SAT solving.

To address the challenge, we now describe the strategy that leverages the word-level data-dependence information to guide the decision order of Boolean variables after bit-blasting (§ 4.2.1). We also discuss the combination of our strategy and the VSIDS branching heuristic (§ 4.2.2).

4.2.1 Ordering with Data Dependence. As discussed in § 2.2, in addition to encoding a bit-vector variable with a sequence of Booleans variables, the bit-blasting procedure also introduces auxiliary Boolean variables to represent each bit-vector term, i.e., the outcome of calculations over the bit-vector variables. Thus, we need to schedule the decision order for Boolean variables encoding both the bit-vector variables and the terms.

Our key observation is that for a formula with a set N of variables and terms, the values of a subset S are often sufficient to determine the values of all variables. Intuitively, this is because program variables and statements usually have some data-dependence relations. For instance, it is common in real-world programs that fresh variables are created for naming expressions, i.e., the calculation results of other variables. Such assignments naturally introduce data dependence.

In the SAT solving phase (Algorithm 1), once the Boolean variables in $BV(S)$ are assigned, the BCP can infer the values of other variables in $BV(N \setminus S)$ automatically. Intuitively, at the beginning of the search, the assignments to $BV(S)$ can cause longer chains of constraint propagation, because the values of other variables, as well as the values of terms over the variables, are the deterministic consequence of that choice.

Therefore, our basic idea is to give higher initial scores to the variables in $BV(S)$, which has two benefits. First, if the decisions made within $BV(S)$ do not lead to conflicts, the BCP can infer more truth values for other variables, accelerating the finding of a satisfying model. Second, if some "inappropriate" decisions lead to conflicts, the SAT solver will use the conflicts information to refine the assignments. The refinement can further influence the values of other clauses that are dependent on $BV(S)$.

We now discuss how to identify the subset S and how to prioritize the elements within S. We start by constructing a data-dependence graph for all bit-vector variables and terms. The graph is split into a set of independent sub-graphs with a formula slicing algorithm [83], such that nodes in each sub-graph must have some data dependence. For example, as shown in Figure 3, a node in the graph represents a variable or a term, and an edge represents a data-dependence relation. Then, we schedule the decision order as follows.

Rule 1. *Given two nodes v_1 and v_2 such that v_2 is data-dependent on v_1, we have $BV(v_2) \preceq BV(v_1)$, meaning that the Boolean variables in $BV(v_1)$ has a higher priority than those in $BV(v_2)$.*

Some clients of SMT solvers, such as symbolic execution, may eliminate intermediate program variables that name bit-vector terms, such that the formulas mention only input variables. In such cases, our approach can still leverage the data dependence to guide the decision order.

Example 4.3. Consider the constraint in Figure 3. Suppose that the variables x and y are the intermediate variables while the others

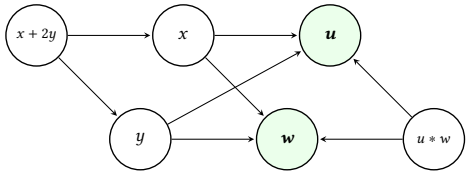


Figure 3: Data-dependence graph for the constraint $\phi \equiv x = u + w \wedge y = 2 * u - w \wedge x + 2 * y < 10 \wedge u * w < 60$.

are input variables of a program. A symbolic executor may generate a constraint where the variables x and y are replaced by $u + w$ and $2 * u - w$, respectively, so that the constraint only contains program input variables:

$$\phi \equiv u + w + 2 * (2 * u - w) < 10 \wedge u * w < 60.$$

We denote the terms, $u+w$, $2*u-w$, $u*w$, and $u+w+2*(2*u-w)$, as, t_1, t_2, t_3, and t_4, respectively. As illustrated in Example 2.2, the bit-blaster needs to introduce Boolean variables to encode each element in u, w, t_1, t_2, t_3, and t_4. Since all the terms are data-dependent on u and w, we have

$$BV(t_1), BV(t_2), BV(t_3), BV(t_4) \preceq BV(u), BV(w).$$

Meanwhile, we have $BV(t_4) \preceq BV(t_1)$ and $BV(t_4) \preceq BV(t_2)$ because t_4 is data-dependent on t_1 and t_2.

In addition to data dependence, control dependence relations encoded in a constraint also can be leveraged to order the nodes that do not have data dependence relations. For instance, for a simple program if (c) { v = a; } else { v = b; }, where the value of the variable v is control-dependent on the condition c, if the value of c is known, we then only need to compute the value of either a or b. Thus, when solving a constraint encoding this control-dependence relation, it is preferable to have $BV(a), BV(b) \preceq BV(c)$. In practice, such a control-dependence relation is usually encoded as an *ite* (if-then-else) constraint: $\phi \equiv v = ite(c, a, b)$, or its equivalent forms. Given such a constraint, we have the following rule.

Rule 2. *Given a formula in the form of $v = ite(c, a, b)$ or its other equivalent forms, we have $BV^*(a), BV^*(b) \preceq BV^*(c)$, where we use $BV^*(t)$ to represent the Boolean variables in $BV(t)$ and all other variables the term t data-depends on.*

4.2.2 In Combination with VSIDS. In principle, our strategy can replace the default VSIDS heuristic, which maintains a score for each variable and updates the scores periodically (§ 2.3). However, in practice, relying exclusively on the dependence-based scheme is not practical because it cannot dynamically refine the scores. The previous study [12] shows that branching strategies with dynamic re-ordering are usually more effective than static ones.

Therefore, we combine our scoring scheme with VSIDS for the decision-making. Recall that VSIDS initializes the variable scores based on only statistical information, such as the number of clauses in which a variable appears. The previous work [56, 85] has shown that guiding VSIDS in initializing the scores can notably increase the efficiency of the solving process. Hence, we initialize the scores by combining the dependence and the statistical information, and then update the scores by following VSIDS.

In general, the impact of the data-dependence information decreases over time because VSIDS favors the most recently detected conflict clauses, which will eventually dominate when the search progresses. Fortunately, in practice, VSIDS can make more informed decisions when the search progresses, because it can gather more information about the search history to make a sophisticated choice.

To summarize, our design aims to guide the SAT solver at the beginning of the search, when VSIDS is not yet well-formed enough, and allows VSIDS to override the initial decisions when it has a deeper understanding of the formula.

5 EVALUATION

We implement Trident on top of the Z3 SMT solver. Specifically, Trident uses Z3 for parsing formulas in the SMT-LIB v2.6 format, performing word-level simplifications, and conducting the bit-blasting. Finally, the translated Boolean formulas are handed to our customized SAT solver, which leverage our interval and data-dependence analyses to guide its branching heuristics. Our evaluation is designed to answer the following research questions:

- **RQ1**: How effective are the two guidance strategies of Trident (§ 5.2)?
- **RQ2**: Is Trident faster to solve constraints compared to other state-of-the-art SMT solvers (§ 5.3)?
- **RQ3**: Can Trident improve the scalability of existing symbolic analysis tools (§ 5.4)?

5.1 Experimental Setup

Subjects. We use three realistic symbolic analysis tools to generate bit-vector constraints for evaluating our approach. Specifically, Angr [78][4] is a binary analysis platform, Qsym [88][5] is a symbolic execution engine for hybrid fuzzing, and Pinpoint [77] is a path-sensitive static bug finder. We target subjects that cover different scales of programs, whose sizes range from a few thousand lines to multi-million lines, cover a wide range of applications, such as networking libraries and database engines, and cover both standard benchmarks and open-source projects. In total, as listed in Table 2, we collect thirty programs in three groups:

- Angr is configured to analyze ten programs from Coreutils,[6] a commonly-used dataset in symbolic execution.
- The ten programs run by Qsym are taken from the tool's original paper [88], including three programs in the LAVA-M dataset [30] and seven open-source projects.
- We apply Pinpoint to analyze ten industrial-sized C/C++ programs, which are the monthly trending projects on GitHub that we can set up.

To answer RQ1 and RQ2, we run the tools with their default solvers to dump SMT queries as the SMT-LIB v2.6 format, and then conduct the experiments on these queries. The number of total queries generated by Angr, Qsym, and Pinpoint is 2,123,211, 1,502,958, and 3,806,989, respectively. Among the queries, 5,236,735 instances are satisfiable, and 2,196,423 ones are unsatisfiable. We believe such a large number of instances are sufficient to evaluate the performance of SMT solving.

[4] https://github.com/angr/angr
[5] https://github.com/sslab-gatech/qsym
[6] https://www.gnu.org/software/coreutils

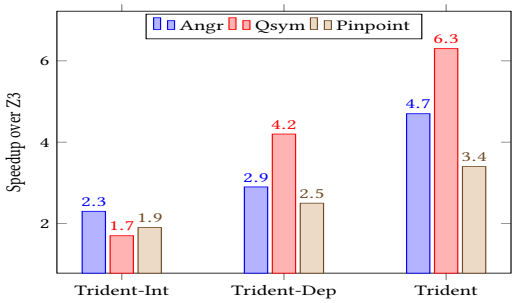


Figure 4: Relative performance impact of the strategies.

Table 1: Number of unsolved queries.

Group	Z3	Trident-Int	Trident-Dep	Trident
Angr	61	29	27	23
Qsym	133	121	32	11
Pinpoint	57	45	21	21

To answer RQ3, we substitute Trident for Z3, which is the SMT solver used by Angr, Qsym, and Pinpoint. Since Trident retains the Z3 API, the program analyzers using Z3 can directly benefit from our work. In the experiment, we examine whether the superior performance of Trident translates into end-to-end benefits for the symbolic analysis tools.

Environment. All solvers are compiled with gcc 7.4.0 using the flags `-O3 -m64 -march=native`. We configure each solver to use a per-query timeout of 30 seconds, following the prior works [66, 71] on accelerating SMT solving in symbolic analysis. All solvers are compiled with gcc 7.4.0 using the flags `-O3 -m64 -march=native`. We configure each solver to use a per-query timeout of 15 seconds.

5.2 RQ1: Effectiveness of the Guidance Strategies

First, we investigate the effectiveness of the strategies in our solver by comparing its four configurations. Specifically, we compare Trident to the original Z3 solver,[7] Trident-Int, and Trident-Dep, two configurations with each of the interval-guided assignment or dependence-guided ordering strategies turned on.

Figure 4 shows the results in terms of solving time. The data for each benchmark group is normalized to the runtime of Z3. A number larger than 1.0 is a speedup. Table 1 compares the number of unsolved queries within the given time limit.

We can observe that both of the two strategies in Trident contribute to its performance. The data dependence-based strategy has better effects than that of interval-based strategy. It would be hard to infer precise interval information for some queries, rendering the results less effective in reducing the search space. In the Qsym group, data-dependence analysis is the most effective. We find that queries from Qsym tend to have many variables that have strong

[7] We set the SAT solver of Z3 to use the VSIDS branching heuristic and the Luby restart strategy [55].

data-dependence relations. Overall, the speedups range from 3.4× to 6.3×.

Additionally, using the two optimizing strategies, Trident is able to solve 38, 122, and 36 more queries than Z3 in the Angr, Qsym, and Pinpoint groups, respectively.

> Trident leverages the interval and data-dependence information to reduce the solving time of Z3 by 3.4× to 6.3×, as well as increase the number of solved queries.

5.3 RQ2: Comparison to Other SMT Solvers

In addition to comparing with Z3, on which Trident is built, we also examine the practicality of Trident by comparing it against two state-of-the-art SMT solvers, namely, CVC4 v1.7 and Boolector v2.4.1, In particular, Boolector won the first place in the QF_BV track of SMT-COMP 2019,[8] and CVC4 also won many champions in other tracks.[9]

Figure 5 plots the cumulative runtime of the solvers. Table 2 shows the detailed comparison results. For each program, we report the total SMT queries, the number of unsolved queries, and the time cost of the solvers. The last column of Table 2 denotes the speedup achieved by Trident over the fastest baseline in CVC4 and Boolector.

We can notice that, for most of the programs, Trident obtains 2.0× to 3.1× increases in performance against the baseline. For the Angr, Qsym, and Pinpoint groups, Trident is, on average 3.0×, 2.3×, and 3.1× faster than the baseline, respectively. The largest improvements are seen for `ffmpeg` and `v8` in the Pinpoint group, with speedups of 4.2×.

There are four programs for which the speedups are smaller than 2.0×, including `nm`, `tcpdump` and `jhead` in the Qsym group, and `mysql` in the Pinpoint group. However, we have seen other solvers much slower when handling queries from several programs. When solving queries from `who`, `size`, `tcpdump`, and `jhead`, CVC4 is more than 11× slower than Trident. In the cases of `gluster`, `ffmpeg`, and `v8`, Boolector is 4× slower than Trident. To sum up, Trident is, on average 7.9× and 2.9× faster than CVC4 and Boolector, respectively.

There are a total of eleven programs for which the three SMT solvers finish all queries: six of them are in the Angr group, and five of them are in the Qsym group. In total, Trident solves 6, 2, and 40 more queries than the baseline for the Angr, Qsym, and Pinpoint groups, respectively. Overall, we conclude that Trident can solve as many queries as the other two solvers.

> Compared with the fastest solver between CVC4 and Boolector, Trident improves the SMT solving speed by 1.6× to 4.2×, without lessening the number of solved queries within the time limit.

[8] https://smt-comp.github.io/
[9] https://cvc4.github.io/awards.html

5.4 RQ3: Improving the Scalability of Symbolic Analysis Tools

Finally, we evaluate the usefulness of Trident for three program analysis clients. Specifically, we configure Angr for running control-flow recovery analysis [78], Qsym for generating test cases, and Pinpoint for detecting null pointer dereference bugs. For each client, we measure the speedups achieved by the three configurations of our solver, following the settings of § 5.2.

Results of Angr. Figure 6 shows the reduction of analysis time for control-flow recovery, which is a fundamental step for binary analysis [24, 74, 78]. On average, Trident-Int, Trident-Dep, and Trident demonstrate 1.5×, 1.6×, and 1.9× speedups, respectively. The overall analysis speedups are smaller than the pure SMT solving speedups, because, in addition to the symbolic execution engine, Angr consists of other components that can be time-consuming, such as the backward slicing [78].

Results of Qsym. Figure 7 summarizes the speedup of input generation in 24 hours of testing. To sum up, on average, Trident-Int, Trident-Dep, and Trident increase the speed by 1.2×, 1.5×, and 1.6× speedups, respectively. As a case study, Figure 8 shows the cumulative branch coverage for `objdump` and `readelf`. The speedups are smaller than the pure SMT solving speedups because Qsym is used for hybrid fuzzing [82, 88], where a significant proposition of the time is spent on program execution and symbolic emulation. Overall, the results show that Trident significantly improves the scalability of Qsym, a state-of-the-art symbolic execution engine for test case generation.

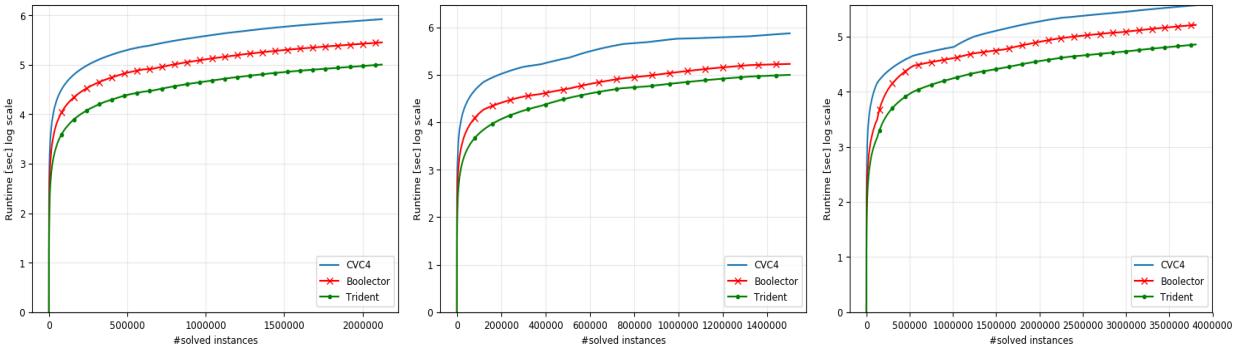
Results of Pinpoint. In Table 3, we report speedups on the four largest programs, which are representative of speedups on the remaining programs. On average, Trident-Int, Trident-Dep, and Trident demonstrate about 1.5×, 1.7×, and 2.4× speedups, respectively. We remark that the time reduction is non-trivial for a static bug finder, which is enough to make an originally impractical analysis usable in practice. For example, originally, Pinpoint cannot finish analyzing `v8` within 11.4 hours. In contrast, with the optimizations offered by Trident, Pinpoint completes the analysis within 5 hours, meaning that it can run in nightly mode, i.e., perform the full analysis of a large codebase from scratch, run by the continuous integration system over night [13].

> On average, Trident increases the end-to-end analysis speed of three clients in Angr, Qsym, and Pinpoint by 1.9×, 1.6×, and 2.4×, respectively.

5.5 Threats to Validity

The internal validity mainly depends on the correctness of our implementation. To reduce this threat, at least four developers review the source code of Trident. Furthermore, we have performed extensive testing of Trident on thousands of applications, during which it has solved billions of SMT queries. For these queries, we compare the solving results given by Trident and several existing SMT solvers for validating the correctness of our implementation.

The threats to external validity lie in our test suits. To reduce the threat resulting from benchmarks, we validate our approach



(a) Queries from Angr (b) Queries from Qsym (c) Queries from Pinpoint

Figure 5: Results of CVC4, Boolector, and Trident on all solved instances.

Table 2: Results of the SMT solvers over the test suites. For each solver, we record the number of unsolved queries and the total time (in minutes) taken. For Trident, the column "Speedup" denotes the speedup over the *fastest* solver between CVC4 and Boolector.

Group	Program	Queries	CVC4		Boolector		Trident		
			TO	Time	TO	Time	TO	Time	Speedup
Angr	fmt	363,552	8	2598	3	862	3	**279**	3.1×
	logname	189,243	0	1167	0	366	0	**119**	3.1×
	mkfifo	146,998	0	1120	0	342	0	**110**	3.3×
	nice	307,243	62	2214	8	736	9	**238**	3.1×
	nohup	140,296	0	859	0	267	0	**87**	3.1×
	env	253,595	21	1708	5	585	0	**189**	3.1×
	head	97,438	0	642	0	212	0	**68**	2.7×
	mv	173,122	20	1060	13	335	11	**109**	3.1×
	nl	145,582	0	779	0	191	0	**70**	2.7×
	nproc	306,142	0	1850	0	586	0	**191**	3.1×
Qsym	uniq	133,587	0	1211	0	243	0	**121**	2.0×
	md5sum	92,692	0	188	0	188	0	**59**	3.2×
	who	121,647	0	1001	9	259	0	**90**	2.9×
	readelf	113,812	11	726	0	172	6	**77**	2.2×
	nm	80,496	0	327	0	71	0	**40**	1.8×
	objdump	178,555	0	1053	0	209	0	**88**	2.4×
	size	129,580	19	919	1	160	5	**71**	2.3×
	tcpdump	236,256	0	3536	2	521	0	**292**	1.8×
	djpeg	235,357	0	970	1	525	0	**226**	2.3×
	jhead	180,976	0	1383	0	118	0	**76**	1.6×
Pinpoint	gluster	465,054	1	211	28	411	0	**65**	3.2×
	libicu	226,418	2	226	0	134	0	**50**	2.7×
	imgmagick	111,122	4	89	2	54	1	**22**	2.5×
	openssl	356,173	28	219	6	155	1	**44**	3.5×
	python	124,685	3	72	0	53	0	**19**	2.8×
	gcc	132,550	13	108	0	47	0	**23**	2.0×
	ffmpeg	419,306	91	416	2	402	1	**95**	4.2×
	v8	645,970	100	674	7	616	4	**146**	4.2×
	mysql	468,879	28	198	6	133	6	**70**	1.9×
	wine	856,832	150	677	10	719	8	**187**	3.6×
Total TO			561		103		55		

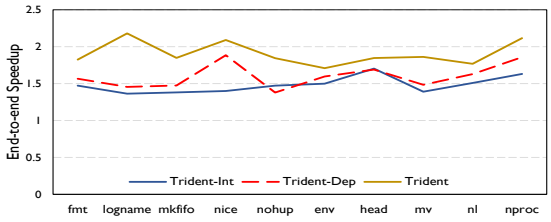


Figure 6: Speedup of control-flow recovery analysis in Angr.

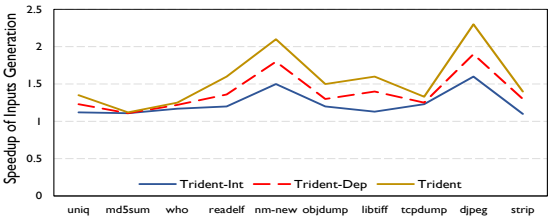


Figure 7: Speedup of test case generation using Qsym.

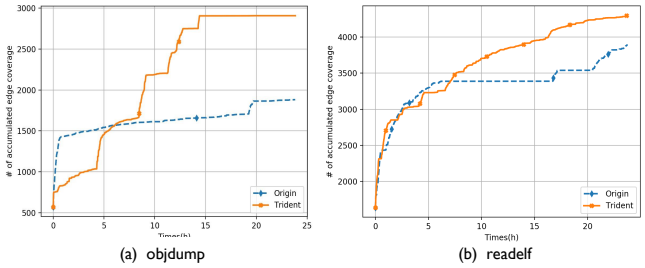


Figure 8: Number of covered branches in 24 hours.

Table 3: Reducing the end-to-end bug finding time (in minutes) of Pinpoint.

Program	KLoC	Z3	Trident-Int	Trident-Dep	Trident
ffmpeg	1001	315	207 (1.5×)	188 (1.7×)	118 (2.7×)
v8	1201	687	424 (1.6×)	359 (1.9×)	278 (2.5×)
mysql	2030	164	122 (1.3×)	112 (1.5×)	97 (1.7×)
wine	4108	526	302 (1.7×)	329 (1.6×)	210 (2.5×)

over three different symbolic analysis platforms and on thirty real-world subjects varying in scale and functionality. However, the queries from these platforms are not necessarily representative of other tools. Another threat to validity is whether our strategies can be generalized to other eager SMT solvers. In the future, we will further apply our approach to other solvers and more symbolic analysis frameworks.

6 RELATED WORK

We discuss related work in three groups: symbolic execution (§ 6.1), bit-vector constraint solving (§ 6.2) and branching heuristics (§ 6.3).

6.1 Accelerating SMT Solving in Symbolic Execution

There is a large body of work on accelerating constraint solving for symbolic execution. Different forms of caches such as satisfying model [20] and unsatisfiable core [60] help avoid calling the solver when possible. Klee employs a method called "constraint independence" to eliminate sub-formulas that are irrelevant to the current branch expression. Our approach can guide the SAT solver to better handle variables that are not independent, by levering their data-dependence relations. Symbolic executors can perform different algebraic simplifications of expressions to make the constraints more solver-friendly [21, 67, 76]. Semantics-preserving program transformations [19, 31, 84] also help to generate simpler and fewer SMT queries. These simplifications and transformations operate at the level of the symbolic execution engines, and, thus, are orthogonal to our work that operates at the level of SMT solvers. The idea of employing a portfolio of SMT solvers for parallel solving has been explored [66], whereby each solver runs independently, and the result of the fastest solver is taken. Our solver can be combined with existing solvers in a portfolio.

6.2 Decision Procedures for Bit-Vectors

The majority of the state-of-the-art SMT solvers [2, 25, 29, 33, 63] solve bit-vector constraints via reduction to SAT, some employing specialized procedures for equality reasoning [17] and linear modulo arithmetic [36]. A number of methods alternative to bit-blasting have been developed, such as the Canonizer-based approaches [4, 28], reduction to Effectively Propositional Logic (EPR) [49], the model-constructing satisfiability calculus (mcSAT) [89], the stochastic local search (SLS) algorithms [35, 64, 65], and the abstraction-based methods [18]. Most of these approaches operate over restricted subsets of bit-vector theory. We tried the SLS-based solvers in Boolector and Z3, but found that they are not competitive with the solvers examined in our evaluation.

Most existing work on enhancing the eager approach focuses on improving the word-level preprocessing and bit-blasting phases [34, 45, 48, 57, 70, 81]. Unconstrained term propagation [16, 17] identifies variables and terms that are irrelevant for determining satisfiability, thereby reducing the problem size. Nadel [61] proposes an algorithm that generates word-level rewriting rules at runtime for a given problem. Singh and Solar-Lezama [81] generates word-level simplifiers using program synthesis. Inala et al. [45] generate domain-specific encoding schema for bit-blasting with synthesis and machine learning techniques. These optimizations for word-level preprocessing and bit-blasting are orthogonal to our approach.

Interval Constraint Propagation (ICP) is a sound but incomplete numerical method for constraint solving [7, 8, 14, 23, 32, 38]. Most existing ICP techniques work only for real or integer formulas. Janota and Wintersteiger [46] propose a method for inferring interval information of bit-vectors from a system of simple inequalities. In each of the inequality, only one variable is permitted and there are no multiplications. Dustmann et al. [32] present a semi-decision procedure for bit-vectors, which tracks the possible values for each symbolic variable, so as to quickly solve certain types of queries before bit-blasting. Compared with the previous work, our analysis is not restricted to certain classes of queries. Besides, it attempts to

reduce the search space of SAT solving after bit-blasting, instead of directly deciding the satisfiability.

6.3 Branching Heuristics

There is a huge literature on SAT branching heuristics, such as MOM [69], Jeroslow-Wang [47], VSIDS [59], Conflict History-Based (CHB) [54], and so on. Different variants of VSIDS have also been proposed, such as BerkMin's strategy [41], exponential VSIDS (EVSIDS) [10], variable move-to-front (VMTF) [72], clause-move-to-front (CMTF) [39], and average conflict index decision score (ACIDS) [12]. However, these branching heuristics are designed for general SAT problems.

The dependence information has been used to restrict the set of decision variables for SAT solving [26, 27, 52, 58]. Given a Boolean formula with a set N of variables, they restrict the branching heuristic to focus on a subset S that is sufficient to determine the truth values of all variables. In contrast, we leverage the data-dependence relations to initialize the scores, without restricting the branching only to the subset. Besides, we combine the data- and control-dependence information to further schedule the order of variables within the subset S.

Previous work shows that structure information can be utilized to improve the branching heuristic for solving SAT queries from bounded model checking (BMC). Shtrichman [79] predetermines the variable ordering by traversing the variable dependency graph. Wang et al. [85] identify important variables from previous unsatisfiable BMC instances and apply them to solve the current instance. Yin et al. [87] give a higher priority to the transition variables from the transition system. These approaches target pure Boolean formulas and leverage the domain-specific knowledge of circuits, instead of word-level structure and information.

The theory-aware approaches [9, 22, 42, 75] for DPLL(T) lazy SMT solving attempt to make the SAT branching heuristic aware of the T-semantic of the literals. Z3Str3 [9] biases the search towards branches that contain easier string constraints. Goldwasser et al. [42] guide the branching by leveraging the T-implications between linear arithmetic constraints. Being aware of the implication relations, they can choose unassigned atomic predicates that are consistent with the current partial assignment. A recent work [22] takes advantage of the control-flow information to reduce the redundant search space. These methods differ from our method across two key technical dimensions. First, they work under the context of lazy SMT solving, while our approach attempts to accelerate eager SMT solving. Given a first-order formula, our reasoning centers around the bit-blasted and equisatisfiable Boolean formula, but not around the approximated Boolean skeleton. Second, they exploit correlations between atomic predicates locally, while we analyze globally word-level information such as the variable interval.

7 CONCLUSION

This paper presents an approach to accelerating eager bit-vector solvers, which infers the interval and data-dependence information of the bit-vector formula to guide the SAT solver after bit-blasting. We have evaluated the proposed techniques over queries from three realistic symbolic analysis platforms, demonstrating the advantages of our approach.

ACKNOWLEDGMENTS

We thank Yushan Zhang, Yikun Hu, as well as the anonymous reviewers for their insightful comments. This work is partially funded by an MSRA grant, the Hong Kong GRF16230716, GRF16206517, ITS/215/16FP, ITS/440/18FP grants, and the NSFC Project No. 61902329. Qingkai Shi is the corresponding author.

REFERENCES

[1] Athanasios Avgerinos. 2014. *Exploiting Trade-offs in Symbolic Execution for Identifying Security Bugs.* Ph.D. Dissertation.
[2] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177.
[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The satisfiability modulo theories library (SMT-LIB). www. *SMT-LIB. org* 15 (2010), 18–52.
[4] Clark W Barrett, David L Dill, and Jeremy R Levitt. 1998. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Annual Design Automation Conference* (San Francisco, California, USA) *(DAC '98)*. ACM, New York, NY, USA, 522–527. https://doi.org/10.1145/277044.277186
[5] Clark W. Barrett, David L. Dill, and Aaron Stump. 2002. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*. Springer-Verlag, Berlin, Heidelberg, 236–249. https://doi.org/10.5555/647771.734410
[6] Michael Bartovn, Iddo Hanniel, Gershon Elber, and Myung-Soo Kim. 2010. Precise Hausdorff Distance Computation between Polygonal Meshes. *Comput. Aided Geom. Des.* 27, 8 (Nov. 2010), 580–591. https://doi.org/10.1016/j.cagd.2010.04.004
[7] Jason Belt, Robby, and Xianghua Deng. 2009. Sireum/Topi LDP: A Lightweight Semi-Decision Procedure for Optimizing Symbolic Execution-Based Analyses. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1595696.1595762
[8] Frédéric Benhamou and Laurent Granvilliers. 2006. Continuous and interval constraints. In *Foundations of Artificial Intelligence*. Vol. 2. Elsevier, 571–603.
[9] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A String Solver with Theory-Aware Heuristics. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design* (Vienna, Austria) *(FMCAD '17)*. FMCAD Inc, Austin, Texas, 55–59. https://doi.org/10.5555/3168451.3168468
[10] Armin Biere. 2008. Adaptive Restart Strategies for Conflict Driven SAT Solvers. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing* (Guangzhou, China) *(SAT'08)*. Springer-Verlag, Berlin, Heidelberg, 28–33.
[11] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, NLD. https://doi.org/10.5555/1550723
[12] Armin Biere and Andreas Fröhlich. 2015. Evaluating CDCL Variable Scoring Schemes. In *Theory and Applications of Satisfiability Testing – SAT 2015*, Marijn Heule and Sean Weaver (Eds.). Springer International Publishing, Cham, 405–422. https://doi.org/10.1007/978-3-319-24318-4_29
[13] Sam Blackshear, Nikos Gorogiannis, Peter W. OHearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276514
[14] Mateus Borges, Marcelo d'Amorim, Saswat Anand, David Bushnell, and Corina S. Pasareanu. 2012. Symbolic Execution with Interval Solving and Meta-Heuristic Search. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, USA, 111–120. https://doi.org/10.1109/ICST.2012.91
[15] Aaron R Bradley and Zohar Manna. 2007. *The calculus of computation: decision procedures with applications to verification.* Springer Science & Business Media.
[16] Robert Brummayer. 2009. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays.* Ph.D. Dissertation. Informatik, Johannes Kepler University.
[17] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. 2007. A lazy and layered SMT (BV) solver for hard industrial verification problems. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 547–560.
[18] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. 2007. Deciding Bit-Vector Arithmetic with Abstraction. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Braga, Portugal) *(TACAS'07)*. Springer-Verlag, Berlin, Heidelberg, 358–372.
[19] Cristian Cadar. 2015. Targeted Program Transformations for Symbolic Execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*

(Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 906–909. https://doi.org/10.1145/2786805.2803205

[20] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224.

[21] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2006. EXE: automatically generating inputs of death. (2006), 322–335. https://doi.org/10.1145/1180405.1180445

[22] Jianhui Chen and Fei He. 2018. Control Flow-guided SMT Solving for Program Verification. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 351–361. https://doi.org/10.1145/3238147.3238218

[23] Zakaria Chihani, Bruno Marre, François Bobot, and Sébastien Bardin. 2017. Sharpening constraint programming approaches for bit-vector theory. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 3–20.

[24] Cristina Cifuentes and Mike Van Emmerik. 1999. Recovery of Jump Table Case Statements from Binary Code. In *Proceedings of the 7th International Workshop on Program Comprehension (IWPC '99)*. IEEE Computer Society, USA, 192. https://doi.org/10.5555/520033.858247

[25] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver.. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Rome, Italy) *(TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7

[26] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. 2001. Benefits of Bounded Model Checking at an Industrial Setting. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*. Springer-Verlag, London, UK, UK, 436–453.

[27] James M. Crawford and Andrew B. Baker. 1994. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 2)* (Seattle, Washington, USA) *(AAAI'94)*. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1092–1097. http://dl.acm.org/citation.cfm?id=199480.199540

[28] David Cyrluk, Oliver Möller, and Harald Rueß. 1997. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*. Springer-Verlag, London, UK, UK, 60–71. http://dl.acm.org/citation.cfm?id=647766.733602

[29] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[30] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121. https://doi.org/10.1109/SP.2016.15

[31] Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. 2015. Studying the Influence of Standard Compiler Optimizations on Symbolic Execution. In *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE) (ISSRE '15)*. IEEE Computer Society, USA, 205–215. https://doi.org/10.1109/ISSRE.2015.7381814

[32] Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. 2018. PARTI: A Multi-interval Theory Solver for Symbolic Execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 430–440. https://doi.org/10.1145/3238147.3238179

[33] Bruno Dutertre. 2014. Yices2.2. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

[34] Anders Franzén. 2010. *Efficient solving of the satisfiability modulo bit-vectors problem and some extensions to SMT*. Ph.D. Dissertation. University of Trento.

[35] Andreas Fröhlich, Armin Biere, Christoph M Wintersteiger, and Youssef Hamadi. 2015. Stochastic Local Search for Satisfiability Modulo Theories.. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) *(AAAI'15)*. AAAI Press, 1136–1143. http://dl.acm.org/citation.cfm?id=2887007.2887165

[36] Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 519–531.

[37] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2015. Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss. *ACM Trans. Program. Lang. Syst.* 37, 1, Article 1 (Jan. 2015), 35 pages. https://doi.org/10.1145/2651360

[38] Sicun Gao, Malay Ganai, Franjo Ivančić, Aarti Gupta, Sriram Sankaranarayanan, and Edmund M. Clarke. 2010. Integrating ICP and LRA Solvers for Deciding Nonlinear Real Arithmetic Problems. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design* (Lugano, Switzerland) *(FMCAD '10)*. FMCAD Inc, Austin, Texas, 81–90.

[39] Roman Gershman and Ofer Strichman. 2005. HAIFASAT: A New Robust SAT Solver. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing* (Haifa, Israel) *(HVC'05)*. Springer-Verlag, Berlin, Heidelberg, 76–89. https://doi.org/10.1007/11678779_6

[40] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[41] E. Goldberg and Y. Novikov. 2002. BerkMin: A Fast and Robust Sat-Solver. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '02)*. IEEE Computer Society, USA, 142. https://doi.org/10.5555/882452.874512

[42] Dan Goldwasser, Ofer Strichman, and Shai Fine. 2008. A Theory-Based Decision Heuristic for DPLL(T). In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design* (Portland, Oregon) *(FMCAD '08)*. IEEE Press, Article 13, 8 pages.

[43] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. 2014. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 680–695. https://doi.org/10.1007/978-3-319-08867-9_45

[44] Trevor Alexander Hansen. 2012. *A constraint solver and its application to machine code test generation*. Ph.D. Dissertation.

[45] Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. 2016. Synthesis of Domain Specific CNF Encoders for Bit-Vector Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2016*, Nadia Creignou and Daniel Le Berre (Eds.). Springer International Publishing, Cham, 302–320. https://doi.org/10.1007/978-3-319-40970-2_19

[46] Mikolás Janota and Christoph M Wintersteiger. 2016. On Intervals and Bounds in Bit-vector Arithmetic.. In *SMT@ IJCAR*. 81–84.

[47] Robert G Jeroslow and Jinchang Wang. 1990. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence* 1, 1-4 (1990), 167–187.

[48] Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. 2009. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 668–674. https://doi.org/10.1007/978-3-642-02658-4_53

[49] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. 2013. bv2Epr: A tool for polynomially translating quantifier-free bit-vector formulas into EPR. In *Proceedings of the 24th International Conference on Automated Deduction* (Lake Placid, NY) *(CADE'13)*. Springer-Verlag, Berlin, Heidelberg, 443–449. https://doi.org/10.1007/978-3-642-38574-2_32

[50] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. 2016. Complexity of Fixed-Size Bit-Vector Logics. *Theor. Comp. Sys.* 59, 2 (Aug. 2016), 323–376. https://doi.org/10.1007/s00224-015-9653-1

[51] Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures: An Algorithmic Point of View* (1 ed.). Springer Publishing Company, Incorporated.

[52] Chu Min Li. 2000. Integrating Equivalency Reasoning into Davis-Putnam Procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, 291–296. https://doi.org/10.5555/647288.760210

[53] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic optimization with SMT solvers. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/2535838.2535857

[54] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. 2016. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. AAAI Press, 3434–3440. http://dl.acm.org/citation.cfm?id=3016100.3016385

[55] Michael Luby, Alistair Sinclair, and David Zuckerman. 1993. Optimal Speedup of Las Vegas Algorithms. *Inf. Process. Lett.* 47, 4 (Sept. 1993), 173–180. https://doi.org/10.1016/0020-0190(93)90029-9

[56] Zoltán Ádám Mann and Pál András Papp. 2017. Guiding SAT solving by formula partitioning. *International Journal on Artificial Intelligence Tools* 26, 04 (2017), 1750011.

[57] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. 2012. Automated Reencoding of Boolean Formulas. In *Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing* (Haifa, Israel) *(HVC'12)*. Springer-Verlag, Berlin, Heidelberg, 102–117. https://doi.org/10.1007/978-3-642-39611-3_14

[58] Fabio Massacci and Laura Marraro. 2000. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning* 24, 1-2 (2000), 165–203.

[59] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the*

38th Annual Design Automation Conference (Las Vegas, Nevada, USA) *(DAC '01)*. ACM, New York, NY, USA, 530–535. https://doi.org/10.1145/378239.379017

[60] Jan Mrázek, Martin Jonáš, Vladimír Štill, Henrich Lauko, and Jiří Barnat. 2017. Optimizing and Caching SMT Queries in SymDIVINE. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*. Springer-Verlag, Berlin, Heidelberg, 390–393. https://doi.org/10.1007/978-3-662-54580-5_29

[61] Alexander Nadel. 2014. Bit-Vector Rewriting with Automatic Rule Generation. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 663–679. https://doi.org/10.1007/978-3-319-08867-9_44

[62] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890

[63] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014 (published 2015). Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), 53–58.

[64] Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. Propagation based local search for bit-precise reasoning. *Formal Methods in System Design* 51, 3 (2017), 608–636. https://doi.org/10.1007/s10703-017-0295-6

[65] Aina Niemetz, Mathias Preiner, Armin Biere, and Andreas Fröhlich. 2015. Improving local search for bit-vector logics in SMT with path propagation. *Proceedings of DIFTS* (2015), 1–10.

[66] Hristina Palikareva and Cristian Cadar. 2013. Multi-Solver Support in Symbolic Execution. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (Saint Petersburg, Russia) *(CAV 2013)*. Springer-Verlag, Berlin, Heidelberg, 53–68.

[67] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating Array Constraints in Symbolic Execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/3092703.3092728

[68] Corneliu Popeea and Wei-Ngan Chin. 2006. Inferring Disjunctive Postconditions. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues* (Tokyo, Japan) *(ASIAN'06)*. Springer-Verlag, Berlin, Heidelberg, 331–345. https://doi.org/10.5555/1782734.1782760

[69] Daniele Pretolani. 1996. Efficiency and stability of hypergraph SAT algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 26 (1996), 479–498.

[70] Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Andres Nötzli, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2018. Rewrites for SMT solvers using syntax-guided enumeration. In *SMT Workshop*.

[71] Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling up DPLL (T) string solvers using context-dependent simplification. In *International Conference on Computer Aided Verification*. Springer, 453–474.

[72] Lawrence Ryan. 2004. *Efficient algorithms for clause-learning SAT solvers*. Ph.D. Dissertation. Theses (School of Computing Science)/Simon Fraser University.

[73] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. 2006. Static Analysis in Disjunctive Numerical Domains. In *Proceedings of the 13th International Conference on Static Analysis* (Seoul, Korea) *(SAS'06)*. Springer-Verlag, Berlin, Heidelberg, 3–17. https://doi.org/10.1007/11823230_2

[74] B. Schwarz, S. Debray, and G. Andrews. 2002. Disassembly of Executable Code Revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE 02) (WCRE '02)*. IEEE Computer Society, USA, 45. https://doi.org/10.5555/882506.885138

[75] Roberto Sebastiani. 2007. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation* 3 (2007), 141–224.

[76] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[77] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 693–706. https://doi.org/10.1145/3192366.3192418

[78] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[79] Ofer Shtrichman. 2000. Tuning SAT checkers for bounded model checking. In *International Conference on Computer Aided Verification (CAV '02)*. Springer, 480–494.

[80] João P. Marques Silva and Karem A. Sakallah. 1996. GRASP&Mdash;a New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design* (San Jose, California, USA) *(ICCAD '96)*. IEEE Computer Society, Washington, DC, USA, 220–227. http://dl.acm.org/citation.cfm?id=244522.244560

[81] Rohit Singh and Armando Solar-Lezama. 2016. Swapper: A Framework for Automatic Generation of Formula Simplifiers Based on Conditional Rewrite Rules. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) *(FMCAD '16)*. FMCAD Inc, Austin, TX, 185–192. http://dl.acm.org/citation.cfm?id=3077629.3077661

[82] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[83] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. ACM, New York, NY, USA, Article 58, 11 pages. https://doi.org/10.1145/2393596.2393665

[84] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2013. Overify: Optimizing Programs for Fast Verification. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*.

[85] Chao Wang, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. 2004. Refining the SAT Decision Ordering for Bounded Model Checking. In *Proceedings of the 41st Annual Design Automation Conference* (San Diego, CA, USA) *(DAC '04)*. ACM, New York, NY, USA, 535–538. https://doi.org/10.1145/996566.996713

[86] Yichen Xie and Alex Aiken. 2005. Scalable error detection using boolean satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. ACM, New York, NY, USA, 351–363. https://doi.org/10.1145/1040305.1040334

[87] Liangze Yin, Fei He, and Ming Gu. 2013. Optimizing the SAT Decision Ordering of Bounded Model Checking by Structural Information. In *Proceedings of the 2013 International Symposium on Theoretical Aspects of Software Engineering (TASE '13)*. IEEE Computer Society, USA, 23–26. https://doi.org/10.1109/TASE.2013.11

[88] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, Berkeley, CA, USA, 745–761. http://dl.acm.org/citation.cfm?id=3277203.3277260

[89] Aleksandar Zeljić, Christoph M Wintersteiger, and Philipp Rümmer. 2016. Deciding bit-vector formulas with mcSAT. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 249–266.