Counterexample-Guided Bit-Precision Selection[⋆]

Shaobo He and Zvonimir Rakamarić

School of Computing, University of Utah
Salt Lake City, UT, USA
{shaobo,zvonimir}@cs.utah.edu

Abstract. Static program verifiers based on *satisfiability modulo theories* (SMT) solvers often trade precision for scalability to be able to handle large programs. A popular trade-off is to model bitwise operations, which are expensive for SMT solving, using uninterpreted functions over integers. Such an over-approximation improves scalability, but can introduce undesirable false alarms in the presence of bitwise operations that are common in, for example, low-level systems software. In this paper, we present our approach to diagnose the spurious counterexamples caused by this trade-off, and leverage the learned information to lazily and gradually refine the precision of reasoning about bitwise operations in the whole program. Our main insight is to employ a simple and fast type analysis to transform both a counterexample and program into their more precise versions that block the diagnosed spurious counterexample. We implement our approach in the SMACK software verifier, and evaluate it on the benchmark suite from the International Competition on Software Verification (SV-COMP). The evaluation shows that we significantly reduce the number of false alarms while maintaining scalability.

1 Introduction

Advances in *satisfiability modulo theories* (SMT) solving [3] have significantly enhanced the potential of program verifiers and checkers to reason about large-scale software systems. For instance, SLAM [2] has helped developers to find important bugs in Windows device drivers. The Linux Driver Verification project [19] that uses BLAST [13] and CPAchecker [5] has reported a large number of bugs in Linux drivers. SAGE [12] has been regularly finding security-critical bugs in large Microsoft applications such as media players.

A major obstacle that still often prevents software developers from adopting program verifiers is a high rate of false alarms. A recent survey conducted inside Microsoft shows that most developers are willing to accept only up to 5% false alarm rate [7], which is much smaller than what most state-of-the-art program analyzers can achieve. There are several reasons for such low tolerance to false alarms. First, false alarms can take a long time to triage, therefore significantly impeding developers' productivity. Second, trivial false alarms compromise developers' confidence in using program verifiers. Finally, if a threshold is set for reporting alarms, true bugs can be masked in the presence of many false alarms.

[⋆] This work was supported in part by NSF awards CNS 1527526 and CCF 1552975.

In SMT-based program verifiers, false alarms usually arise from a trade-off between the efficiency of the underlying theory solvers and complete modeling of program semantics. For example, low-level programming languages such as C contain bitwise operations. While the commonly used SMT theory of integers is scalable, it cannot be used to efficiently and precisely model such program constructs. Hence, in practice, verifiers often rely on uninterpreted functions over unbounded integers to over-approximate bitwise operations. This design choice aims to improve scalability at the expense of occasionally losing precision — it does not miss bugs, but can introduce false alarms. On the other hand, it is not difficult for program verifiers to model these behaviors precisely using the theory of bit-vectors instead of integers. However, scalability suffers since the theory of bit-vectors is typically much slower. Moreover, manually deciding which theory to use reduces the usability of program verifiers since users have to determine the necessity for bit-precision, which may not be obvious even for medium-sized programs. For example, bit-field manipulations in C typically compile to bitwise operations. Users of a program verifier that operates on a compiler intermediate representation (or even binary) may not be aware of such details, and would fail to enable the theory of bit-vectors even though a verifier maybe supports it.

In this paper, we propose an automatic counterexample-guided abstraction refinement (CEGAR) [8] approach to gradually on-demand (i.e., lazily) improve bit-precision of SMT-based verifiers. Our approach is based on the observation that the precision of only a subset of bitwise operations is relevant for proving program assertions. Therefore, enabling bit-precision everywhere is an overkill that degrades scalability. We start with a program that uses uninterpreted integer functions to model bitwise operations, iteratively convert spurious bit-imprecise counterexamples to precise ones using type unification, and then propagate the learned type information to the input program until either the program verifies or a real counterexample is found. Our goal is to focus on the bitwise operations that affect the correctness of user provided assertions. Our main contribution is to employ a simple and fast type analysis to assign precise bit-vector types to imprecise uninterpreted bitwise operations and propagate the learned type information throughout the program. We implement our approach as an extension of the SMACK software verification toolchain [21, 23]. We perform an empirical evaluation on benchmarks used in the International Competition on Software Verification (SV-COMP) [25], and show that it automatically removes a large proportion of false alarms while maintaining scalability.

2 Background

In this section, we describe the SMACK software verification toolchain and the simple intermediate verification language that our approach takes as input.

2.1 SMACK Software Verification Toolchain

SMACK is an SMT-based static assertion checker that targets languages compilable to the LLVM *intermediate representation* (IR) [17]. Currently, SMACK

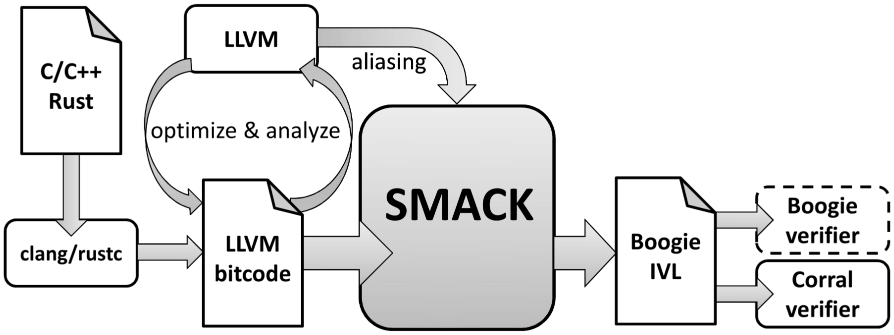

Fig. 1. SMACK software verification toolchain

mainly targets C programs, and adding support for C++ and Rust is work in progress. By default, SMACK verifies user-provided assertions up to a selected loop/recursion bound. SMACK can also automatically generate assertions to check domain-specific properties such as memory safety and signed integer overflows. Fig. 1 shows the SMACK verification toolchain. We first obtain LLVM IR code (bitcode) of the input program using a specific compiler front-end (e.g., clang/clang++ for C/C++). The main SMACK module then translates LLVM IR into the Boogie *intermediate verification language* (IVL) [20, 10] by encoding the semantics of LLVM IR instructions into Boogie. Finally, SMACK integrates multiple Boogie verifiers, and the generated Boogie program is verified using a chosen back-end verifier. In this work, we use Corral [16] as the Boogie verifier because it is scalable and can produce precise error traces; Corral internally invokes SMT solver Z3 [9]. Although we instantiate our approach using SMACK, Corral, and Z3, any verifier combination would suffice that operates on a statically-typed input IVL and produces precise counterexamples.

During the translation, SMACK performs analysis and optimization of the input LLVM IR program to simplify the downstream verification process. One such analysis is the data structure analysis (DSA) [18] provided by LLVM. SMACK uses it as a precise alias analysis to split the input program heap into distinct regions such that pointers referring to two regions can never alias [22]. Each region is translated into a separate memory map (i.e., array) in the Boogie program, which often greatly improves scalability since the number of updates of each individual map is reduced. In situations where the DSA-based alias analysis is imprecise, typically due to low-level pointer manipulations, smaller number of memory maps with more updates each are generated — this can lead to a significant performance penalty as we observe in our empirical evaluation.

By default, SMACK uses Boogie integer type to represent both LLVM pointer type and integer types with certain bit widths. Bitwise operations are over-approximated using uninterpreted functions and thus results of such operations are arbitrary integers. We refer to such a setup that uses the theory of integers as the integer mode of SMACK. In addition to using imprecise bitwise operations,

$$
\begin{array}{rcl}
\Gamma_{int} & ::= & int1 \mid int8 \mid int16 \mid int32 \mid int64 \\
\Gamma_{bv} & ::= & bv1 \mid bv8 \mid bv16 \mid bv32 \mid bv64 \\
\Gamma_{scalar} & ::= & \texttt{bool} \mid \texttt{ref} \mid \Gamma_{int} \mid \Gamma_{bv} \\
\Gamma & ::= & \Gamma_{scalar} \mid [\texttt{ref}]\Gamma_{scalar} \\
x & \in & \texttt{Var} \\
lit & ::= & true \mid false \mid intlit \mid bvlit \\
pred & ::= & == \mid != \\
binop_{ia} & ::= & \textsf{add} \mid \textsf{sub} \mid \textsf{mul} \mid \textsf{udiv} \mid \textsf{sdiv} \mid \textsf{urem} \mid \textsf{srem} \\
binop_{bw} & ::= & \textsf{and} \mid \textsf{or} \mid \textsf{lshr} \mid \textsf{ashr} \mid \textsf{shl} \mid \textsf{xor} \\
binop & ::= & binop_{ia} \mid binop_{bw} \mid compop \mid castop \\
e & ::= & x \mid lit \mid e_1 \; pred \; e_2 \mid uop(e) \mid binop(e_1, e_2) \mid \\
 & & load(x, e_1) \mid store(x, e_1, e_2) \\
cmd & ::= & x := e \mid \textbf{assert} \; e \mid \textbf{assume} \; e \mid \textbf{call} \; x := p(e_i)
\end{array}
$$

Fig. 2. Subset of Boogie IVL that SMACK emits. We only show the syntax up to Boogie commands.

the integer mode does not capture either the wrap-around behavior of unsigned integer overflows or casts between signed and unsigned numbers, which can result in both false alarms and missed bugs. We consider this to be an orthogonal issue that can be handled by, for example, injecting overflow checks into the program, and we only consider false alarms resulting from the over-approximation of bitwise operations. The precision with respect to bitwise operations can be tuned up by enabling the bit-vector mode of SMACK where LLVM scalar types are translated to fixed-size bit-vector types in Boogie. However, our experience in applying SMACK on real-world programs indicates that the bit-vector mode is much less scalable than the integer mode.

2.2 Simple Intermediate Verification Language

A Boogie program generated by SMACK consists of a set of global variables, procedures, and functions. Each procedure contains a set of basic blocks, each of which consists of a series of commands. SMACK translates most LLVM IR instructions to Boogie commands that are either assignments or procedure calls. The left-hand side of such an assignment is a variable corresponding to the result of the instruction while the right-hand side is either another variable or application of a Boogie function representing the operation of the instruction. The syntax of Boogie programs generated by SMACK is shown in Fig. 2. Pointer type `ref` is a synonym of integer type *int*32 or *int*64 depending on the architecture. Integer type Γ_{int} is a synonym of the Boogie integer type `int` which is only used

```
#include "smack.h"
#include <stdlib.h>

typedef struct
{
  int x;
} S;

int f(int* p)
{
  return *p & 0xf;
}

int main()
{
  S* s = (S*)malloc(sizeof(S));
  unsigned y = __nondet_int();

  if (__nondet_int()) {
    s->x = foo(&s->x);
  }
  else {
    assume(y < 4U);
    y >>= 2U;
  }

  if (s->x >= 16)
    assert(!y);
}
```

```
procedure main() returns(r:int32) {
  var p0, p1, p5, p7: ref;
  var i2, i3, i6, i8, i11: int32;
  var i4, i9: int1;

  $bb0:
  call p0 := malloc(4);
  p1 := bitcast.ref.ref(p0);
  call i2 := __nondet_int();
  call i3 := __nondet_int();
  i4 := ne.i32(i3, 0);
  goto $bb1, $bb2;

  $bb1:
  assume (i4 == 1); p5 := p1;
  call i6 := f(p5); p7 := p1;
  M.0 := store.i32(M.0, p7, i6);
  i8 := i2; goto $bb3;

  $bb2:
  assume !((i4 == 1));
  ...
  i8 := i11; goto $bb3;
  ... }
procedure f($p:ref) returns(r:int32) {
  var i0, i1: int32;
  i0 := load.i32(M.0, $p);
  i1 := and.i32(i0, 15);
  r := i1; return; }
```

Fig. 3. C program with bitwise operations and part of its Boogie IVL translation

in the integer mode. Instead, Boogie bit-vector type Γ_{bv} only shows up in the programs generated by SMACK in the bit-vector mode.

SMACK translation decorates each function name with types of its arguments and result. For example, function `add.i32` expects the types of its operands to be *int*32 and returns an integer value of the same type. There will be an incarnation of each binary function (*binop*) for the integer mode and the bit-vector mode, respectively. Function `add.bv32` is the counterpart of `add.i32` in the bit-vector mode which takes two 32 bit bit-vectors as arguments and returns their sum. Functions encoding bitwise operations ($binop_{bw}$) are uninterpreted in the integer mode while implemented precisely as wrappers to Z3 built-in bit-vector functions in the bit-vector mode. Fig. 3 shows a C program and a code snippet of its Boogie IVL translation.

An important feature of Corral that we leverage is that it can generate error trace programs. An error trace program produced by Corral represents an error path that starts from the program entry and ends with an assertion failure. It is a regular Boogie IVL program and follows the syntax defined in Fig. 2. The main difference from the input program is that procedure bodies in the error trace program follow a single control flow path. Hence, verifying the error trace program and its transformation only requires unrolling depth one and is generally much faster than verifying the entire program.

```
Function cexg($P$):
    result, $trace$ = verify($P$)
    if $result \in \{true,\ timeout\}$ then
        return result
    else
        if $binop_{bw}$ exists and remains uninterpreted then
            $trace'$, $EC^t$ = transform($trace$)
            result = verify($trace'$)
            if $result \in \{false,\ timeout\}$ then
                return result, $trace'$
            else
                $P'$ = update($EC^t$, $P$)
                return cexg($P'$)
            end
        else
            return result, $trace$
        end
    end
end
```

Fig. 4. Pseudocode of our approach

3 Approach

In this section, we present our counterexample-guided approach to reduce the number of false alarms due to imprecise modeling of bitwise operations. It is an iterative algorithm that keeps refining the program to verify based on the feedback provided by the already refined error trace programs.

Fig. 4 presents the pseudocode of our approach. We define it as a tail-recursive function cexg which consists of the following steps. First, the Boogie verifier (function verify) is called to verify the input program P. If the program verifies or the verification is inconclusive (i.e., timeout), function cexg exits with the result returned by the verifier. If a bug is reported and the error trace program representing it is generated, then we check if the error trace program contains any uninterpreted functions corresponding to bitwise operations. If not, this counterexample is considered feasible and presented to the user. Otherwise, our approach transforms the error trace program to a more precise version with respect to bitwise operations (function transform). Then, the verifier is invoked to verify it. If the verification result is false or timeout, the error trace is feasible after the refinement or its feasibility cannot be determined given the time limit. In both cases, function cexg terminates and returns a more precise error trace or timeout status, respectively. On the other hand, when the verification succeeds, we know that the counterexample is spurious. Hence, the input program P is updated to a new program P' using function update, which removes the spurious counterexample according to the type equivalence classes EC^t learned from the error trace. Finally, function cexg recurs with P' being the input program. In the rest of this section, we describe the transformation function transform and the update function update in details.

$$\begin{array}{ll}
t_{m\text{-}p5} = t_{m\text{-}p1} & t_{m\text{-}p7} = t_{m\text{-}p1} \\
t_{m\text{-}p5} = t_{f\text{-}\$p} & t_{m\text{-}i6} = t_{f\text{-}r} \\
t_{m\text{-}i8} = t_{m\text{-}i2} & t_{m\text{-}i6} = t_{M.0\text{-}val} \\
t_{f\text{-}i0} = t_{f\text{-}load.i32(M.0,\$p)} & \\
t_{f\text{-}load.i32(M.0,\$p)} = t_{M.0\text{-}val} & \\
t_{f\text{-}i1} = t_{f\text{-}and.i32(i0,15)} & \\
t_{f\text{-}and.i32(i0,15)} = t_{f\text{-}i0} = BV & \\
\hline
\multicolumn{2}{c}{t_{m\text{-}p1}, t_{m\text{-}p5}, t_{m\text{-}p7}, t_{f\text{-}\$p}} \\
\hdashline
\multicolumn{2}{c}{t_{m\text{-}i8}, t_{m\text{-}i2}} \\
\hdashline
\multicolumn{2}{c}{t_{m\text{-}i6}, t_{f\text{-}r}, t_{M.0\text{-}val}, t_{f\text{-}i0}, t_{f\text{-}i1}} \\
\multicolumn{2}{c}{t_{f\text{-}load.i32(M.0,\$p)}, t_{f\text{-}and.i32(i0,15)}, BV} \\
\hdashline
\end{array}$$

```
M.0 : [ref]bv8;
procedure main() returns(r:int32) {
  var $i6 : bv32;
  ...
  $bb1:
  assume (i4 == 1); p5 := p1;
  call i6 := f(p5); p7 := p1;
  M.0 := store.bytes.32(M.0, p7, i6);
  i8 := i2; goto $bb3;
  ... }
procedure f($p:ref) returns(r:bv32){
  var i0, i1: bv32;
  i0 := load.bytes.32(M.0, $p);
  i1 := and.bv32(i0, 15bv32);
  r := i1; return; }
```

Fig. 5. The left part shows the type constraints and solution for block `$bb1` of procedure `main` and the body of procedure `f`. Procedure name `main` is abbreviated as `m` in the type variables. Equivalence classes are separated by dotted lines. The right part is the snippet of the transformed program according to the solution.

3.1 Program Transformation

The transformation implemented by the function transform is based on a simple type analysis. The basic idea is that a new base type representing bit-vectors is created and the types of variables or expressions involved in uninterpreted bitwise operations are assigned to this type. The transformed program returned by function transform is thus more precise than the input program because any uninterpreted bitwise operations are replaced with their precise bit-vector counterparts during the transformation. To discover which types should be updated to bit-vectors, we generate type constraints in terms of equalities and the result of solving these constraints gives us an over-approximation of such types. Finally, we simply rewrite the program to install the type updates. Fig. 5 demonstrates the three phases of transforming an error trace program of the program in Fig. 3 when the condition of the first if statement is true.

Generating Type Constraints Recall that Boogie programs generated by SMACK leverage integer types of LLVM IR that specify the bit-width of each type. For the purpose of generating type constraints, two base types are introduced into our type system: integer *int* and bit-vector *bv*. The integer bit-width information contained in the original LLVM IR types is sufficient to rewrite integer subtypes to more precise corresponding bit-vector subtypes. For example, the type of the local variable `i1` of procedure `f` in Fig. 3 is *int*32. If the type gets lifted to *bv*, we can easily rewrite it as *bv*32 since its original type name contains the bit-width of integers that it represents.

Fig. 6 formalizes the rules used to generate type constraints. The basic idea of type constraint generation is that whenever an uninterpreted bitwise operation is observed, we change the types of the operands as well as the expression into *bv*. On the other hand, we simply equate the types of operands and results

$$\frac{\Gamma \vdash \mathit{binop}_{bw}(e_1, e_2) : t \qquad \Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{t_1 = t_2 = t = BV} \qquad \frac{\Gamma \vdash \mathit{op}(\overrightarrow{e_i}) : t \qquad \Gamma \vdash e_i : t_i}{t_i = t}$$

$$\frac{\Gamma \vdash \mathit{load}(M, e) : t \qquad \Gamma \vdash M : t_1 \to t_2}{t_2 = t} \qquad \frac{P \vdash x := e \qquad \Gamma \vdash x : t_1 \qquad \Gamma \vdash e : t_2}{t_1 = t_2}$$

$$\frac{\Gamma \vdash \mathit{store}(M, e_1, e_2) : t_1 \to t_2 \qquad \Gamma \vdash M : t_3 \to t_4 \qquad \Gamma \vdash e_2 : t_5}{t_2 = t_4 = t_5}$$

$$\frac{\Gamma \vdash e_1 \ \mathit{pred} \ e_2 : \mathbf{bool} \qquad \Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{t_1 = t_2}$$

$$\frac{P \vdash \mathit{call} \ x := p(\overrightarrow{e_i}) \qquad \Gamma \vdash e_i : t_{ai} \qquad \Gamma \vdash x : t \qquad \Gamma \vdash p : t_{pi} \to t_r}{t_{ai} = t_{pi} \qquad t = t_r}$$

Fig. 6. Type rules for generating type constraints. We introduce a type variable for each expression (except literals) and a special type constant BV. Symbol binop_{bw} refers to uninterpreted integer functions that over-approximate bitwise operations. Symbol *op* refers to the union of *uop* and *binop* (excluding binop_{bw}) from Fig. 2.

for other operations. For the example in Fig. 3, the types of variable `i0` and expression `and.i32(i0, 15)` in procedure `f` are assigned to *bv* since `and.i32` is an uninterpreted function over integers used to model bitwise AND operation. On the other hand, the type of variable `i3` is just equal to that of the inequality expression `ne.i32(i3, 0)` in procedure `main`.

The type of *load* expression or the value argument e_2 in the *store* expression is consistent with the type of the map argument M's range. The type of map domain is forced to be *int* by not generating type constraints associated with it. We place such a restriction because we observed that combining the theory of bit-vectors and theory of arrays is generally slow, especially for the case where bit-vector types are map domain types. However, we do not restrict the type of pointers to *int*. For example, the updated type of variable `p7` in Fig. 3 could be *bv*. Therefore, an expression of type *bv* is cast to *int* during the rewriting stage when it is used as the pointer argument of *load* and *store* expressions. On the other hand, we do not introduce casts from *int* to *bv* for two reasons. First, integer to bit-vector cast is an extremely expensive operation for Z3. Second, the input integer value of such a cast must be constrained since the resulting bit-vector has only a finite set of values, thereby increasing the complexity of modeling and reasoning about the program. The cast operations *castop* in Fig. 2 include functions representing LLVM cast instructions (e.g., `zext`, `trunc`, `ptrtoint`). We keep the source type and the destination type as the same base type and thus equate them. The same rule also applies to the comparison operations *compop* in our language that correspond to LLVM comparison instructions.

Solving Type Constraints Since all the type constraints generated are equivalence relations, we use a simple unification algorithm that unifies constraints and produces a number of disjoint sets that consist of type variables that are equivalent. We call these sets *equivalence classes*, and all type variables in an equivalence class have the same base type. If an equivalence class contains BV, then all the type variables in this class have type *bv*. On the other hand, a variable or expression keeps its original type if BV is not in the equivalence class where its type variable is.

Rewriting Programs Once the generated type constraints are solved, we recursively rewrite the input program based on the computed solution. The declared types of variables are changed to fixed-size bit-vector types if their corresponding identifier expressions have base type *bv*. For map variables, only their range types can be lifted. Integer constants are simply replaced with their bit-vector counterparts if the expressions where they are used have type *bv*.

For function applications, recall that there is an integer version and a bit-vector version for each function in the Boogie program generated by SMACK. Therefore, we simply replace the integer version with the bit-vector version if the type is decided to be *bv* and recur to its arguments. Since we would like to keep the type of map domains as integer, casts from *bv* to *int* (implemented as Z3 built-in function `bv2int`) are added to expressions which have type *bv* and are used to index maps. For this reason, the pointer argument of functions *load* and *store* may be encapsulated with function `bv2int`.

Note that the concrete type of a *load* expression or the value argument of a *store* expression can be different from the range type of the map passed as the first argument, although their base types are the same. For example, the type of expression `load.i32(M.0, $p)` in Fig. 3 is $int32$ while the type of `M.0` is [`ref`]$int8$. This often happens when the alias analysis employed by SMACK is imprecise and results in a large number of multi-type accesses to a single map. The range type of such map is $int8$, which indicates that type unsafe accesses may occur and thus accesses should be byte-level. In the integer mode, we assume that elements stored in a map do not overlap. Functions *load* and *store* are just wrappers around map selection and update expressions, respectively. For example, the body of function `load.i32(M, p)` in Fig. 3 is simply $\mathtt{M}[\mathtt{p}]$. If the range types of such maps get updated to *bv*, then all the accesses (*load* and *store*) are rewritten to byte-level versions that use bit-vector extraction and concatenation operations. For example, the body of function `load.bytes.32(M, p)` in Fig. 5 is $\mathtt{M}[\mathtt{p}+3]++\mathtt{M}[\mathtt{p}+2]++\mathtt{M}[\mathtt{p}+1]++\mathtt{M}[\mathtt{p}]$, where $++$ is the bit-vector concatenation operator. For the case where a map holds only elements of a single type, map accesses are simply replaced with their bit-vector counterparts that are also wrappers around the map selection or update expressions.

3.2 Program Update

If the precise error trace program $trace'$ verifies (see Fig. 4), we know that over-approximation of bitwise operations produces a spurious counterexample and updating the types of relevant variables invalidates it. Then, as a simple solution to prevent such spurious counterexamples, it is sufficient to enable the bit-vector mode. However, we observed that such approach severely limits scalability. Instead, rather than changing the bit-precision of the whole program, we perform a restricted transformation of parts of P (implemented as function update) such that it only prevents the error trace represented by $trace$ from reappearing. Such a transformation contains all the type updates performed on $trace$ that are necessary to block the spurious counterexample. On the other hand, we do not change the precision of bitwise operations that are not related to the trace expressions whose types get changed to bit-vectors. Therefore, our approach has the potential to outperform the bit-vector mode of SMACK or even applying the transformation described in Sec. 3.1 to the entire program.

Function update first generates and solves the type constraints of the entire program to obtain equivalence classes EC^p. Then, it propagates the type constraint solution of $trace, EC^t$ to EC^p and thereafter rewrites P. The type constraints of the whole program P are generated slightly differently than of the error trace program $trace$. Recall that for the type constraint generation of $trace$, types of an uninterpreted bitwise operation and its operands are equated to each other and to the type constant BV, according to the first rule in Fig. 6. In contrast, if an uninterpreted bitwise operation is encountered during the type constraint generation of the whole program, types of this expression and its arguments are only equated.[1] Hence, the type constraint solution for the whole program does not contain BV and only indicates which expressions have the same base type.

Propagating the type updates of $trace$ to P works as follows: for each equivalence class $ec^p \in EC^p$ of the whole program, if it intersects with an equivalence class $ec^t \in EC^t$ of the error trace program and $BV \in ec^t$, then BV is added to ec^p. We show next that the propagation ensures the type updates from int to bv in $trace$ are also contained in P. Note that for a type variable tv, if $tv \in ec^p$ of P and $tv \in ec^t$ of $trace$ then $ec^t \subseteq ec^p$. The type constraints generated for the whole program subsume those produced for any error trace programs because the sequence of commands in error trace programs is always a subset of the commands in the whole program. Therefore, to solve the type constraints of the whole program, our unification algorithm could unify type variables that belong to both the entire program and the error trace program, resulting in the same set of equivalence classes. Then, we start with these equivalence classes and continue the unification algorithm, which either expands a class or merges two classes. Both operations produce an equivalence class that is a super set of the original one. Hence, lifted types in the scope of the entire program include those in the scope of an error trace program.

Invoking the verifier on the updated program P' cannot produce $trace$ again. If it would, then the new error trace program $trace''$ would have all the necessary

[1] The solution to the type constraints of the whole program is the same for each iteration of function cexg and can thus be cached. To simplify the presentation, we recompute it in the paper at each iteration.

types being *bv* since P' already contains the type updates of $trace$, and $trace''$ would verify just as $trace'$ does. Moreover, function update updates at least several types in P in each iteration of cexg, which ensures that our iterative approach makes progress. Assume that in iteration i of cexg, update is called if there are uninterpreted bitwise operations in the error trace program $trace_i$ that cause the counterexample to be spurious. In iteration $i+1$, if update is invoked again it improves the precision of some other uninterpreted bitwise operations since those in $trace_i$ and all previous iterations are already precisely modeled. In other words, the number of iterations of our approach is bounded by the number of uninterpreted bitwise operations in the input program, which contributes to the termination guarantee of our approach formalized by the following theorem.

Theorem 1. *Function* cexg *in Fig. 4 terminates if function* verify *terminates.*

Proof. If each invocation of the verifier finishes or exceeds the time limit, then the only potentially non-terminating path in cexg is to recur, which calls function update. Each time update is called, at least one uninterpreted bitwise operation becomes precisely modeled. Since there is only a finite number of uninterpreted bitwise operations in the input program, the number of calls to update as well as cexg is also finite.

4 Empirical Evaluation

We empirically evaluate our approach using benchmarks from SV-COMP [25], which contain several categories representing different aspects of software systems. We leverage BenchExec [6] as the core of our benchmarking infrastructure for reliable and precise performance measurements. Experiments are performed on machines with two Intel Xeon E5-2630 processors and 64 GB DDR4 RAM running Ubuntu 14.04, which are a part of the Emulab infrastructure [26, 11]. As in SV-COMP, we set time limit to 900 seconds and memory limit to 15 GB for each benchmark. We implemented functions transform and update of our approach as a standalone tool, which we invoke from the SMACK toolchain.[2]

We created 4 different configurations of SMACK: *baseline*, *nobv*, *allbv*, and *cexg*. Configuration *baseline* corresponds to the SMACK baseline version (release v1.8.1) that uses heuristics for SV-COMP; the baseline has been carefully optimized for SV-COMP benchmarks using manually crafted filters that identify benchmarks that require bit-precise reasoning, and subsequently enabling the theory of bit-vectors on such benchmarks. Configuration *nobv* uses imprecise reasoning in the theory of integers (i.e., bitwise operations are encoded as uninterpreted functions), while configuration *allbv* uses precise reasoning in the theory of bit-vectors. Configuration *cexg* implements our counterexample-guided bit-precision selection approach. Whenever the theory of bit-vectors is used, we tune Corral by enabling several well-known options that improve its performance

[2] We made the tool publicly available at `https://github.com/shaobo-he/TraceTransformer`.

in the presence of bit-vectors. We run every configuration on all SV-COMP benchmarks[3] except categories ReachSafety-Float, ConcurrencySafety-Main, MemSafety-LinkedLists, and Termination .[4]

4.1 Used Metrics

We measure the performance of each configuration using the SV-COMP scoring schema, which assigns scores to verification results as follows:

a) +2 when the verifier correctly verifies a program,
b) +1 when it reports a true bug,
c) −32 when the verifier misses a bug,
d) −16 when it reports a false bug (i.e., false alarm), and
e) 0 when the verifier either times out or crashes.

We calculate the weighted score of a meta-category (e.g., ReachSafety) by again following the SV-COMP rules: normalized score of each subcategory is summed up and multiplied with the average number of benchmarks in that category.

Since a verifier is severely punished for reporting incorrect results, we add *correct result ratio* as an additional metric to complement the weighted score. We define it as the average percentage of correctly labeled verification tasks of all subcategories in a (meta-)category. We introduce the *timeout ratio* and *false alarm ratio* metrics to measure the scalability and precision of the configurations, respectively. Similar to the correct result ratio, the timeout ratio is the weighted percentage of timeouts in a category, while the false alarm ratio is the weighted percentage of false alarms.

We use score-based quantile functions [4] to visualize the overall performance of a configuration. Fig. 7 shows an example of such functions for subcategory ReachSafety-BitVectors. The horizontal axis x represents the accumulated score of n fastest correct verification tasks and all of those incorrectly labeled as false. Therefore, the x coordinate of the leftmost point is the number of all false alarms multiplied by −16 and of the rightmost point is the total score. The vertical axis y is the largest runtime of n fastest correct verification tasks.

4.2 Results

Table 1 shows the performance of each configuration over all the used metrics. As expected, configuration *baseline* yields the best performance overall in terms

[3] Benchmark `sleep_true-no-overflow_false-valid-deref.i` from category Systems_BusyBox_Overflows is removed because an invalid dereference leads to a signed integer overflow error, which is not specified by the SV-COMP rules.

[4] Bit-vector mode must always be enabled for reasoning about floating-points, and it is also not consistent with SMACK's support for Pthreads. Our tool currently does not fully support SMACK's encoding of memory safety properties. Finally, SMACK currently cannot verify termination.

Table 1. Experimental results for all the SV-COMP categories of interest

Category	Weighted Score				Timeout Ratio			
	baseline	*nobv*	*allbv*	*cexg*	*baseline*	*nobv*	*allbv*	*cexg*
ReachSafety	3498	1155	2905	2675	16.6	17.5	29.5	17.7
MemSafety	375	321	179	375	10.9	10.9	57.6	10.9
Overflows	472	141	450	469	4.2	4.2	7.7	4.2
SoftwareSystems	2731	2133	503	2529	46.2	41.8	85.1	44.0

Category	Correct Result Ratio				False Alarm Ratio			
	baseline	*nobv*	*allbv*	*cexg*	*baseline*	*nobv*	*allbv*	*cexg*
ReachSafety	81.2	76.3	66.8	79.2	0.00	3.33	0.00	0.22
MemSafety	88.6	87.5	42.2	88.6	0.00	1.10	0.00	0.00
Overflows	92.3	85.8	91.7	91.4	0.00	5.56	0.00	0.00
SoftwareSystems	50.2	54.6	8.8	53.4	0.04	1.79	0.00	0.85

of both scalability and precision since it has been manually fine-tuned over the years on these benchmarks. It has the highest weighted score for all the top-level categories and it solved the largest weighted percentage of benchmarks correctly in 3 out of the 4 categories total. Configuration *nobv* times out the least, but it also produces the largest number of false alarms. Note that the scoring system is crafted to mimic users' disfavor of false alarms by deducting a large number of points (16) when a false alarm is generated. Therefore, although *nobv* managed to correctly solve the largest percentage of benchmarks in the SoftwareSystems category, its score is less than both *baseline* and *cexg* as a result of its high false alarm rate. In contrast, configuration *allbv* does not generate any false alarms, but at the expense of solving the smallest percentage of verification tasks — in particular in the SoftwareSystems category that contains benchmarks from real-world large software systems such as Linux drivers. Moreover, *allbv* also times out much more frequently than other configurations in the MemorySafety category with mostly small or medium benchmarks. Hence, we conclude that always using the theory of bit-vectors is not practical on real-world benchmarks. Configuration *cexg*, which implements our approach, successfully eliminates most of the false alarms seen in *nobv* without placing a significant burden on scalability.

We focus next on the results for benchmarks that could actually benefit from our approach, meaning those benchmarks where uninterpreted functions appear in counterexamples. (Note that our approach has basically no influence on other benchmarks.) For each of these benchmarks, our approach either produces a correct result or it times out (on the transformed error trace program or the updated version of the whole program). Table 2 shows the experimental data we gathered for subcategories that contain such benchmarks. The results show that our approach outperforms configurations *nobv* and *allbv* in terms of precision and scalability, respectively. It avoids all the false bugs reported by *nobv* and times out less than *allbv*. In general, our approach does

Table 2. Experimental results for benchmarks that could potentially benefit from our approach. #B is the total number of such benchmarks; #TOC is the number of timeouts in *cexg*; #TET is the number of true (i.e., confirmed) error traces; #FET is the number of false (i.e., infeasible) error traces discovered by our approach; #RR is the average runtime ratio between our approach and baseline; #FAN is the number of false alarms in *nobv*; #TOA is the number of timeouts in *allbv*. R, M, and O in the first column stand for categories ReachSafety, MemSafety, and Overflows, respectively. The meta-category prefix (Systems) is omitted for subcategories DeviceDriverLinux64 and BusyBox_Overflows.

Subcategory	**#B**	**#TOC**	**#TET**	**#FET**	**#RR**	**#FAN**	**#TOA**
R-BitVectors	20	0	4	16	1.5	14	1
R-ControlFlow	21	2	19	0	3.6	0	21
R-Loops	2	0	1	1	4.4	0	1
M-Heap	8	0	1	7	1.8	6	0
O-Other	6	0	0	6	2.3	6	0
DeviceDriversLinux64	136	111	22	18	11.6	3	133
BusyBox_Overflows	1	1	0	1	—	1	1

not place a significant runtime overhead on the verification tasks such that the time limit is exceeded. However, as the complexity of benchmarks increases, and especially when they become memory-intensive (i.e., containing numerous reads and writes from dynamically allocated memory), our approach may not scale well. Benchmarks in the two subcategories with high runtime overhead, ReachSafety-ControlFlow and DeviceDriversLinux64, contain many more memory accesses than those in the other subcategories in Table 2. Nevertheless, our approach still outperforms *allbv* on these subcategories. Moreover, for subcategory DeviceDriversLinux64 where error traces are usually long and tedious to debug, it successfully rejected 18 spurious counterexamples, 2 of which lead to more precise ones.

4.3 Discussion

We identify memory accesses as the main culprit that sometimes limits the scalability of our approach because type updates are much less contained in their presence. The range type of a memory map is updated to bit-vector if it is involved in a bitwise operation in an error trace, thereby changing the types of all the elements in this map to bit-vectors. This leads to large parts of the program being unnecessarily converted into using the more expensive theory of bit-vectors. Moreover, the imprecision of the alias analysis used for memory splitting can also cause the types of pointer variables to be changed to bit-vectors even though they are not involved in any bitwise operations (due to false aliasing with an integer variable that is involved in a bitwise operation). In this case, we add expensive type cast operations to load and store expressions. Finally, bit-vector extraction and concatenation operations are needed for byte-level accesses to maps containing elements of different types. We conjecture that improving

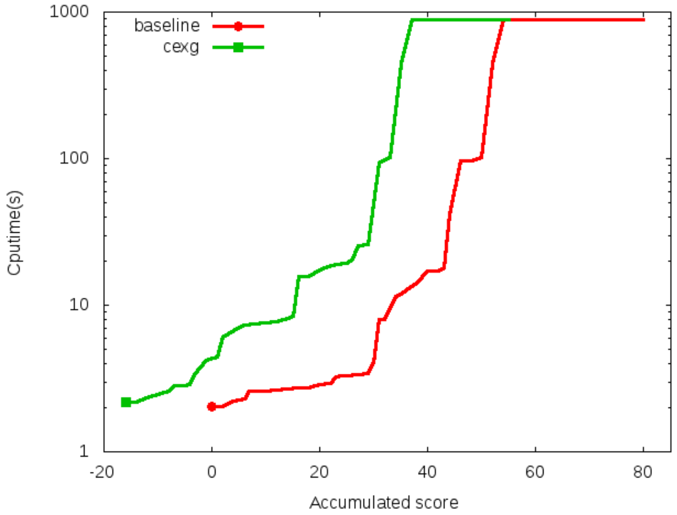

Fig. 7. Quantile functions for subcategory ReachSafety-BitVectors for configurations *baseline* and *cexg*. The vertical axis uses a logarithmic scale.

the precision of the alias analysis used in memory splitting would result in better scalability of our approach, but this is beyond the scope of this paper.

Another issue that can limit the scalability of our approach is that the type constraint generation can lead to overly aggressive type changes since all the uninterpreted functions in an error trace are considered, some of which may not affect the correctness of program assertions. Although optimizations are possible as discussed in Sec. 6, the current setup suffices to give correct diagnose when a false alarm arises as a result of over-approximations of bitwise operations. Furthermore, the type analysis is context-insensitive which can also be optimized (e.g., by forking stateless procedures). In the rest of this section, we present two detailed case studies that highlight the strengths and limitations of our approach.

ReachSafety-BitVectors Subcategory ReachSafety-BitVectors contains benchmarks that require precise modeling of bitwise operations and unsigned integers (e.g., to model the wrap-around behavior of unsigned integer overflows). Therefore, configuration *nobv* yields a high false alarm rate on this subcategory, reporting 15 false alarms out of 50 verification tasks in total. Moreover, even for the buggy benchmarks the produced error traces are likely to be infeasible. On the other hand, our approach removed 14 of these false alarms. (The only false alarm that is not removed by our approach is due to lack of precise modeling of casts between signed and unsigned integers.) In addition, it also discovered 2 infeasible error traces due to the imprecise modeling of bitwise operations, and it automatically refined them into their precise counterparts. With respect to scalability, our approach did not cause any verification tasks to time out and it also placed little run time overhead as demonstrated by Table. 2. It even outperformed the baseline version on some benchmarks. Fig. 7 presents the quantile functions of *cexg* and *baseline*, and we can observe that the two are close in terms of performance and scalability. We find these results to be particularly encouraging because our approach completely automatically achieves almost the same level of precision and performance as the highly-optimized (albeit manually) baseline version.

SoftwareSystems This category includes large and complicated real-world benchmarks, and hence the results obtained by our approach are mixed.

BusyBox Benchmarks Benchmarks ported from the BusyBox 1.22.0 Linux utilities are checked for memory safety and signed integer overflows in the Systems_BusyBox_MemSafety and Systems_BusyBox_Overflows subcategories, respectively. Our approach outscored the baseline version in subcategory Systems_BusyBox_Overflows by proving 4 more benchmarks while reporting zero false alarms. In System_BusyBox_MemSafety, although the baseline version outscores our approach, we still report one more correct result. The only remaining false alarm we report is not due to bitwise operations, and is thus out of scope of this work. The SMACK baseline version is not as fine-tuned on the BusyBox benchmarks as it is on the benchmarks from other categories. Hence, the fact that our approach correctly solves more benchmarks proves its potential to enable automatic bit-precision selection that has better performance than manually deciding bit-precision.

Linux Drivers The subcategory Systems_DeviceDriversLinux64 contains large benchmarks extracted from Linux drivers. This category pushes our approach to the limits of its scalability: only 39.4% of the benchmarks that the baseline version correctly reported as false is also correctly reported by our approach, while 81.6% of the benchmarks times out when Corral is invoked on either the transformed error trace programs or updated input programs. Several reasons contribute to such poor scalability of our approach on this subcategory. First, these benchmarks contain numerous bit-level manipulations over the driver flags that are often defined as C bit-fields, and some even employ bitwise operations in pointer arithmetic. Second, the benchmarks are pointer- and memory-intensive due to the heavy usage of the Linux kernel data structures. Finally, the alias analysis used in memory splitting loses precision more in this subcategory than in the other (sub-)categories, likely due to the existence of external calls and inline assembly.

5 Related Work

The Boogie-to-Boogie transformation used in our approach was inspired by Microsoft's Static Driver Verifier (SDV) [15], which also leverages type analysis to re-type the program and generate a mixed integer and bit-vector program which may precisely model the semantics of bitwise operations. However, SDV eagerly lifts the types of the whole program while our approach lazily updates them based on the feedback obtained from error trace programs. SDV's memory model as well as rare appearance of bitwise operations in the SDV's benchmark suite makes type changes fairly controlled even at the level of the entire programs. On the other hand, even though in our approach the transformation function can be applied to the whole program, verification of the transformed program does not scale on our more complex benchmarks. The proposed selective type updates guided by counterexamples allow us to alleviate this limitation. Moreover, the transformation used in our approach is guaranteed to improve the precision of modeling bitwise operations whereas SDV's may not because it restricts the types of certain expressions as integers in order to maintain scalability.

Our approach is also similar to the framework of *counterexample-guided abstraction refinement* (CEGAR) [8] used in SLAM [2]. Both systems validate the counterexamples produced in the verification process and leverage them to refine the input program. They are different in two aspects. First, the root cause of the spurious witness traces in SLAM is the predicate abstraction on the input program while for this paper it is using uninterpreted functions over integers to model bitwise operations. Second, although CEGAR guarantees progress, it does not necessarily terminate since the input program may contain infinite states. In contrast, the number of iterations in our approach is bounded by that of bitwise operations and thus it eventually terminates given that the verifier terminates when invoked. Furthermore, our approach also guarantees to make progresses in each iteration as discussed in Sec. 3.2.

Others have explored the idea of counterexample-guided abstraction refinement as well. For example, Lahiri et al. [14] present a greedy CEGAR technique that is used to refine memory maps of a program. Babić et al. [1] introduce a technique called *structural abstraction* that iteratively refines a program by analyzing counterexamples and replacing uninterpreted functions with inlined procedures. CAMPY [24] tackles the problem of verifying if a program satisfies a complexity bound expressed in undecidable non-linear theories by selectively inferring necessary axioms in terms of grounded theorems of such non-linear theories and fitting them into decidable theories. This framework could potentially be used as an alternative method to refine the uninterpreted functions that over-approximate bitwise operations. However, for certain programs, especially those in the ReachSafety-BitVectors subcategory of the SV-COMP benchmark suite, simple axioms over integers and uninterpreted functions are usually not expressive enough to complete the proofs, while complex axioms (e.g., enumerations of input-output mappings of bit-vector functions) may significantly reduce the performance.

6 Conclusions and Future Work

Based on our experience with performing software verification on real-world low-level programs, we identified the need for precise reasoning about bitwise operations as an important issue that is crippling many contemporary software verifiers. On one end of the spectrum, verifiers opt for exclusively bit-precise reasoning, which often becomes a performance and/or scalability bottleneck. On the other end, verifiers opt for imprecise modeling using integers and uninterpreted functions, which often leads to a large number of false alarms. We propose an approach that attempts to strike a balance between the two extremes — it starts with imprecise modeling and gradually increases precision on-demand driven by spurious counterexamples. We implemented the approach by leveraging the SMACK toolchain, and performed an extensive empirical evaluation on the SV-COMP benchmarks. Our results show that it reduces the number of false alarms while maintaining scalability, which makes it competitive with a highly manually optimized baseline on small- to medium-size benchmarks.

As future work, we would like to improve the scalability of our approach on large-scale memory-intensive benchmarks by exploring two possible directions. The first direction is to refine the memory model such that the type updates are more controlled. For example, we could leverage the strict aliasing rules of C/C++ to further split memory regions (maps) by element types since aliasing of pointers pointing to elements with different types is undefined behavior. In this way, we could probably avoid the case where pointers become bit-vectors due to the overly conservative alias analysis. The second direction is to increase the granularity of the refinement. Currently, all the uninterpreted bitwise operations in an error trace program are considered, while in fact some of them may not contribute to the false alarm. Therefore, we could greedily start from one of the uninterpreted bitwise operations and perform the transformation, while leaving the others as uninterpreted. If the false alarm disappears, we propagate the type updates caused by only a subset of the uninterpreted bitwise operations in the error trace program. As an alternative to this greedy technique, we could analyze the model returned by Z3 to identify relevant bitwise operations.

References

1. Babić, D., Hu, A.J.: Structural abstraction of software verification conditions. In: Proceedings of the International Conference on Computer Aided Verification (CAV). pp. 371–383 (2007)
2. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static driver verification with under 4% false alarms. In: Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD). pp. 35–42 (2010)
3. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, chap. 26, pp. 825–885. IOS Press (2009)
4. Beyer, D.: Second competition on software verification. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 594–609 (2013)
5. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proceedings of the International Conference on Computer Aided Verification (CAV). pp. 184–190 (2011)
6. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Proceedings of the International Symposium on Model Checking Software (SPIN). pp. 160–178 (2015)

7. Christakis, M., Bird, C.: What developers want and need from program analysis: An empirical study. In: Proceedings of the International Conference on Automated Software Engineering (ASE). pp. 332–343 (2016)
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings of the International Conference on Computer Aided Verification (CAV). pp. 154–169 (2000)
9. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340 (2008)
10. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005)
11. Emulab network emulation testbed. `http://www.emulab.net`
12. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium (NDSS). pp. 151–166 (2008)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the Symposium on Principles of Programming Languages (POPL). pp. 58–70 (2002)
14. Lahiri, S.K., Qadeer, S., Rakamarić, Z.: Static and precise detection of concurrency errors in systems code using SMT solvers. In: Proceedings of the International Conference on Computer Aided Verification (CAV). pp. 509–524 (2009)
15. Lal, A., Qadeer, S.: Powering the Static Driver Verifier using Corral. In: Proceedings of the International Symposium on Foundations of Software Engineering (FSE). pp. 202–212 (2014)
16. Lal, A., Qadeer, S., Lahiri, S.: Corral: A solver for reachability modulo theories. In: Proceedings of the International Conference on Computer Aided Verification (CAV) (2012)
17. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). pp. 75–86 (2004)
18. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI). pp. 278–289 (2007)
19. Linux driver verification project. `https://forge.ispras.ru/projects/ldv`
20. Leino, K.R.M.: This is Boogie 2 (2008)
21. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: Proceedings of the International Conference on Computer Aided Verification (CAV). pp. 106–113 (2014)
22. Rakamarić, Z., Hu, A.J.: A scalable memory model for low-level code. In: Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009). pp. 290–304 (2009)
23. SMACK software verifier and verification toolchain. `http://smackers.github.io`
24. Srikanth, A., Sahin, B., Harris, W.R.: Complexity verification using guided theorem enumeration. In: Proceedings of the Symposium on Principles of Programming Languages (POPL). pp. 639–652 (2017)
25. International competition on software verification (SV-COMP). `https://sv-comp.sosy-lab.org`
26. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. SIGOPS Oper. Syst. Rev. 36(SI), 255–270 (2002)