# BLITZ: Compositional Bounded Model Checking for Real-World Programs

Chia Yuan Cho[†§] Vijay D'Silva[§] Dawn Song[§]
[§]University of California, Berkeley, USA
[†]DSO National Laboratories, Singapore
{chiayuan, vijayd, dawnsong}@cs.berkeley.edu

***Abstract*—Bounded Model Checking (BMC) for software is a precise bug-finding technique that builds upon the efficiency of modern SAT and SMT solvers. BMC currently does not scale to large programs because the size of the generated formulae exceeds the capacity of existing solvers. We present a new, compositional and property-sensitive algorithm that enables BMC to automatically find bugs in large programs. A novel feature of our technique is to decompose the behaviour of a program into a sequence of BMC instances and use a combination of satisfying assignments and unsatisfiability proofs to propagate information across instances. A second novelty is to use the control- and data-flow of the program as well as information from proofs to prune the set of variables and procedures considered and hence, generate smaller instances. Our tool BLITZ outperforms existing tools and scales to programs with over 100,000 lines of code. BLITZ automatically and efficiently discovers bugs in widely deployed software including new vulnerabilities in Internet infrastructure software.**

## I. INTRODUCTION

Software Bounded Model Checking (BMC) is a powerful technique for finding bugs in bounded program executions. The technique constructs a formula, called a BMC *instance*, that encodes the behaviour of a program up to a user-specified bound. BMC instances can capture bit-level operations and the memory model, and so generate precise information about bugs. However, existing BMC tools exhaust memory or time on programs containing a few thousand lines of code [11].

In this paper, we ask: *is it possible solve a large instance by only solving instances that fit in memory*. Specifically, we seek to design a *compositional* BMC algorithm. An analysis technique is compositional if it can decompose a large problem into smaller problems, and solve only a small problem at a time. For example, Hoare logic is compositional because it allows for proving a statement about a program by proving statements about its parts. Several static analysis techniques are compositional because they propagate information through a program one block at a time.

Decomposing BMC instances is challenging because dependencies between constraints are lost if the formula is decomposed. Imagine a procedure `bar()` which calls a procedure `foo(int x)` with argument `2`. If the assertion is never violated, no formula that includes the behaviour of both `foo` and `bar` will be satisfiable. If we decompose the problem and reason about `bar()` and `foo(int x)` separately, the assertion may be violated because there is no constraint on the input variable `x`. More generally, decomposing a BMC instance may render an unsatisfiable formula satisfiable because the context in which the code executes is not taken into account. We do not wish to report that an assertion *may be violated* because we lose the appealing property that BMC reports precise information about bugs.

### *Compositional Bounded Model Checking*

We use two ideas to address the problems sketched above. First, we consider *preconditions for violation*, which are conditions under which a program is guaranteed to violate an assertion. Computing such preconditions precisely would involve the expensive step of quantifier elimination while over-approximating preconditions leads to spurious bug reports. We compute the *underapproximate* preconditions for a violation, and weaken them iteratively. We use information from proofs generated by SAT solvers to *expand* the set of error states that must lead to a violation. As a result, we can decompose a BMC instance, consider multiple inputs to a piece of code, and preserve the accuracy of BMC with respect to bugs.

Our second idea is to incrementally generate BMC instances using information from proofs generated by SAT solvers. Solvers reason about semantic relationships between variables when they construct proofs, so proofs provide *relevance heuristics* [14] for tuning program analyses. By constructing BMC instances relevant to variables and operations appearing in a proof, we dramatically reduce the number of instances that must be considered. Our approach is closely related to the use of Craig interpolants in verification [17] and the formulae we construct also satisfy the interpolation criteria. Consequently, our technique is not only compositional, but like interpolation techniques, is property-driven.

Our tool BLITZ implements a compositional BMC technique that combines underapproximate preconditions with incremental construction of BMC instances. Both steps are guided by unsatisfiability proofs generated by SAT solvers. We have evaluated BLITZ on vulnerability benchmarks containing unmodified real-world programs and found that BLITZ faces no capacity problems and can find bugs in programs of approximately 100KLOC. In addition, we applied BLITZ to check critical Internet infrastructure software from the Internet Systems Consortium and found multiple new vulnerabilities.

### *Content and Contributions*

We present an algorithm and tool for compositional BMC that boosts the capacity of BMC and can discover bugs in real-

world programs with about 100KLOC. We make the following contributions.

1) We present a new BMC algorithm that achieves compositionality by computing underapproximate preconditions, and improves scalability by proof-guided incremental instance construction.
2) We demonstrate that the technique extends the state of the art of BMC by applying our tool BLITZ to real-world benchmarks containing up to about 100KLOC.
3) We deploy BLITZ on critical Internet infrastructure software and find multiple new vulnerabilities.

The paper is organised as follows: We revisit the terminology of BMC in Section III, where we also recall notions from program analysis. Our compositional BMC algorithm is presented in Section IV and evaluated in Section V. We discuss related work in Section VI and conclude in Section VII.

## II. Technique Overview

We present a run of BLITZ on a simple program. The example is intentionally simple to facilitate presentation.

**Goal.** The program in Figure 1 begins execution in `main` and contains calls to four procedures `foo`, `baz`, `bar` and `qux`. The procedures `baz` and `qux` have no side effects and are not shown. The goal is to determine if the assertion at line `18` of `bar` is violated. The standard BMC procedure will take as input a parameter $k$ and construct an instance $\mathcal{M}_k$ in which loops and recursive procedures are unwound $k$ times and procedure calls are inlined. This approach leads to large formulae.

***Formula construction:*** The first novelty of BLITZ is that we only generate formulae for small code fragments rather than the entire program. BLITZ begins by generating a formula for the behaviour of `bar` as shown in Figure 2. Observe that the variables have been labelled with subscripts so that a variable has different indices as its value changes in the lifetime of the program. Technically the formula corresponds to Single Static Assignment (SSA) form. The values of `b` and `c`, being inputs, are unknown and denoted $*$, as is the value of `f`, because the body of `qux` is not considered.

***Underapproximate preconditions:*** The formula for `bar` is satisfiable so a SAT solver can generate a satisfying assignment. This satisfying assignment may not correspond to a feasible execution because the calling context for `bar` does not appear in the formula. The second novelty of BLITZ is to use a satisfying assignment to first generate an unsatisfiable formula and then use a proof of unsatisfiability to obtain a *precondition for assertion violation*. In this case, BLITZ generates the precondition $(b_1 < \texttt{'b'})$, where `'b'` denotes the ASCII value of the character `b`. This means that the assertion is violated if the value of the variable $b_1$ is strictly less than the ASCII value of the character `b`.

There are two differences between the precondition computed by BLITZ and that of standard techniques. It is common to compute weakest preconditions with respect to a correctness property. Instead, we compute preconditions with respect to an assertion violation. The second difference is that we do not necessarily compute weakest preconditions. The Dijkstra weakest precondition with respect to the assertion violation condition $(d < \texttt{'d'})$ is the formula

$$\begin{aligned}(2 \times c_1 = 1) &\implies (c_1 < \texttt{'d'})\\ \wedge\neg(2 \times c_1 \neq 1) &\implies (b_1 + 2 < \texttt{'d'})\end{aligned}$$

which simplifies to

$$\begin{aligned}(2 \times c_1 = 1) &\implies (c_1 < \texttt{'d'})\\ \wedge\neg(2 \times c_1 \neq 1) &\implies (b_1 < \texttt{'b'}).\end{aligned}$$

The precondition generated by BLITZ is an *underapproximation* of the weakest precondition. The practical benefit of our formula is that it is smaller and contains fewer Boolean operations, hence is easier for a solver to manipulate. This simplicity translates to significant performance benefits for larger formulae. The trade-off of this performance benefit is that BLITZ may miss bugs (produce false negatives) because it uses underapproximation. False negatives are mitigated by using a refinement strategy that weakens preconditions.

***Incremental Instance Construction:*** Next, BLITZ has to determine if `bar` can be called in a context satisfying $(b_1 < \texttt{'b'})$. Standard BMC and recent techniques like CORRAL [11] and ALTER [21] consider the callers and callees surrounding a procedure to generate such constraints. This approach would result in `qux` and `baz` being added to the instance.

BLITZ uses data-flow information in the program and only considers contexts that affect the underapproximate precondition. In the example, we can skip `qux` and `baz` and consider the call to `foo` at line `3`. Solving an instance with the body of `foo` generates the precondition $(a_0 < \texttt{'a'})$, which is a sufficient precondition for the assertion violation. Solving this formula provides an input value to trigger the assertion violation. We have found that incrementally generating BMC instances by combining data-flow with underapproximate preconditions significantly reduces the number and size of instances we consider.

***Fine-Grained Decomposition:*** In the example above, we have argued at the level of procedures. Sometimes a procedure (or its unwinding) is too large to fit in memory. BLITZ supports decomposing BMC instances at different levels of granularity up to single statements. At the finest level, we start with a single statement directly preceding an assertion violation and incrementally compute necessary preconditions for single statements. We only work at this level of granularity when required or when configured by the user.

To complete the overview, Figure 2 (b) illustrates the formula constructed if we inline `bar` in `main`. Suppose BLITZ is run with a statement-level granularity, it will generate an underapproximate precondition for individual statements. Since data-flow is taken into account, statements not affecting the approximate preconditions are ignored.

## III. Terminology and Preliminaries

We introduce background and notation about BMC and program analysis. The details of BMC included here are required later for correctness proofs and to clarify the differences between our technique and existing ones.

```
1:  void main(unsigned char a)
2:  {
3:   char b = foo(a);
4:   char c = baz(b); /* some function */
5:   bar(b,c);
6:  }
7:  char foo(unsigned char a)
8:  {
9:   return a + 1;
10: }
```

```
11: void bar(unsigned char b, c)
12: {
13:  char d = b + 2;
14:  char e = c * 2;
15:  char f = qux(d,e); /* some function */
16:  if (e == 1)
17:   d = c;
18:  assert(d >= 'd');
19: }
```

Fig. 1: An example program.

| |
|---|
| $b_1 = *;$ |
| $c_1 = *;$ |
| $d_1 = b_1 + 2;$ |
| $e_1 = c_1 \times 2;$ |
| $f_1 = *;$ |
| $d_2 = (e_1 = 1)\ ?\ c_1\ :\ d_1;$ |
| $d_2 < \texttt{'d'}$ |

(a)

| | |
|---|---|
| $b_1 = a_0 + 1;$ | $a_0 < \texttt{'a'}$ |
| $c_1 = baz(b_1);$ | – |
| $d_1 = b_1 + 2;$ | $b_1 < \texttt{'b'}$ |
| $e_1 = c_1 \times 2;$ | $d_1 < \texttt{'d'}$ |
| $f_1 = qux(d_1, e_1);$ | – |
| $d_2 = (e_1 = 1)\ ?\ c_1\ :\ d_1;$ | $d_1 < \texttt{'d'} \vee (e_1 = 1 \wedge c_1 < \texttt{'d'})$ |
| $d_2 < \texttt{'d'}$ | $d_2 < \texttt{'d'}$ |

(b)

Fig. 2: Formulae generated by BLITZ (a) The BMC formula generated for `bar`. (b) The left column shows the formula generated for the program and the right column shows the preconditions computed if BLITZ is run at the granularity of statements. Statements that are bypassed are indicated by '–'.

### A. Syntax and Semantics of Programs

For simplicity of presentation, the formalisation here focuses on a small subset of C. Our technique and tool both apply to full ANSI C and are not subject to these restrictions.

*Syntax:* Let $Var$ be a set of variables. We use the symbol $\star$ to denote a non-deterministic value. Let $Exp$ be a set containing expressions in C and the $\star$ symbol. Similarly, $BExp$ is a set containing Boolean expressions in C and the $\star$ symbol. A *statement* is an assignment, assumption, assertion, sequential composition of statements or a procedure call. The set of statements $Stmt$ is defined inductively below.

$$\texttt{st} ::= \texttt{x = t} \mid \textbf{assume}\texttt{(b)} \mid \textbf{assert}\texttt{(b)} \mid \texttt{call P()} \mid \texttt{st;st} \mid \textbf{if}\texttt{(b)st } \textbf{else } \texttt{st} \mid \textbf{while}\texttt{(b)st}$$

*Semantics:* The semantics of programs is given by states and state transitions. A *program state* consists of the values of the program counter, global variables, and contents of the stack and the heap. Let $State$ be the set of states.

The semantics of a program is given by a relation. A statement $st$ defines a *transition relation* $T_{st} \subseteq State \times State$ that contains a pair $(r, s)$ if executing $st$ in the state $r$ results in the state $s$. An assignment changes the value of a variable in a state and leaves all other states unchanged. The semantics of **assume**(bexp) is a relation that contains $(s, s)$ if $s$ satisfies bexp. The semantics of **assert**(bexp) is similar except that if $s$ does not satisfy bexp there is a transition $(s, e)$ to an error state $e$. The semantics of sequential composition p;q is the relational composition $T_p \circ T_q$. We write $T_P$ to be the transition relation of a program $P$.

*Error Reachability:* Program properties such as assertion violations can be formulated as reachability of states in a transition system. A state $s$ is *reachable* in a program if there is an execution whose last state is $s$. The *error reachability problem* is to determine if an error state is reachable.

An error reachability technique is *sound* if whenever the technique reports an error, the error is reachable. An error reachability technique is *complete* if whenever the technique reports that the error is not reachable the error is indeed not reachable. Our definitions of soundness and completeness are given with respect to reachability and differ from the notions used in the correctness literature.

### B. Bounded Model Checking

Bounded Model Checking (BMC) is a technique for finding bugs in bounded program executions. It operates by unwinding a program, translating the unwinding into a logical formula, and solving the formula.

*Unwinding:* This material is based on [5]. A $k$-*unwinding* of a program $P$ for a non-negative $k$ is a program $unwind(P, k)$ defined inductively below.

$$\begin{aligned}
unwind(\texttt{x = t}, k) &\hat{=} \texttt{x = t} \\
unwind(\textbf{assume}\texttt{(b)}, k) &\hat{=} \textbf{assume}\texttt{(b)} \\
unwind(\textbf{if}\texttt{(b)p } \textbf{else } \texttt{q}, k) &\hat{=} \textbf{if}\texttt{(b)}\ unwind(\texttt{p}, k) \\
&\qquad \textbf{else}\ unwind(\texttt{q}, k) \\
unwind(\textbf{while}\texttt{(b)p}, 0) &\hat{=} \textbf{if}\texttt{(b) } \textbf{assume}\texttt{(false)} \\
unwind(\textbf{while}\texttt{(b)p}, k+1) &\hat{=} \textbf{if}\texttt{(b)\{ p;} \\
&\qquad unwind(\textbf{while}\texttt{(b)p}, k)\texttt{\}}
\end{aligned}$$

The definition of the unwinding for other C constructs is similar. Recursive procedures are handled by inlining. An *input variable* in a BMC unwinding is one that is never

assigned. Input variables are the source of non-determinism in an unwound program.

*Translation to Logic:* Statements in an unwinding are translated to formulae. An assignment `x=x+1` (where `=` is assignment) cannot be written as a formula $x = x + 1$ (where $=$ represents mathematical equality) because $x$ must have the same value on both sides of the equality in the formula. The statement `x=x+1` is rewritten to `x1 = x0 + 1` to make explicit that `x` may have different values before and after the assignment. This statement is then translated to a formula $x_1 = x_0 + 1$. More generally, a program is in Single Static Assignment (SSA) form if every variable is assigned exactly once in the program. Unwindings are translated to SSA form.

A $k$-unwinding of `P` in SSA form is translated into two logical formulae $\mathcal{B}(\texttt{P})$ and $\mathcal{E}(\texttt{P})$. The *behavioural constraint* $\mathcal{B}(\texttt{P})$ encodes program executions. The *error constraint* $\mathcal{E}(\texttt{P})$ encodes error conditions such as out-of-bounds array accesses, double frees, and assertion violations. If $\mathcal{B}$ implies $\neg\mathcal{E}$, no execution of a $k$-unwinding leads to an error. Conversely, a $k$-unwinding contains an error if $\mathcal{B} \wedge \mathcal{E}$ is satisfiable. The satisfiability condition can be checked using a SAT solver.

The behaviour formula $\mathcal{B}(\texttt{st})$ and error condition $\mathcal{E}(\texttt{st})$ for a statement `st` as defined in [5] are recalled below. We write $form(\texttt{t})$ for the logical expression corresponding to a C expression `t`.

| `st` | $\mathcal{B}(\texttt{st})$ | $\mathcal{E}(\texttt{st})$ |
|---|---|---|
| `x=t` | $x = form(t)$ | `true` |
| **`assume`**`(b)` | $form(b)$ | `true` |
| **`assert`**`(b)` | `true` | $\neg form(b)$ |
| `P;Q` | $\mathcal{B}(\texttt{P}) \wedge \mathcal{B}(\texttt{Q})$ | $\mathcal{E}(\texttt{P}) \wedge \mathcal{E}(\texttt{Q})$ |

All statements except assertions modify the behaviour constraint. The property is only modified by assertions. A BMC instance is *satisfiable* if $\mathcal{B} \wedge \mathcal{E}$ is satisfiable. A BMC instance is generated by traversing the control flow graph (CFG) of $unwind(P, k)$ in topological order starting from the initial vertex $init$. If `x` is an input variable of $unwind(P, k)$, we say that $form(x)$ is a *input variable* of of the BMC instance. We illustrate the construction of a BMC instance below.

**Example 1.** A program $P$ is given on the left below with an unwinding on the right.

```
while(x < y)
  x = x + 1;
assert(x >= y);
```

$\longrightarrow$

```
if(x < y){
 x = x + 1;
 if(x < y){
  x = x + 1;
  if(x < y)
   assume(false);
 }
}
assert(x >= y);
```

The unwinding is translated into the formulae below.

$$\mathcal{B} \mathrel{\hat{=}} x_1 = (x_0 < y_0)\ ?\ (x_0 + 1)\ :\ x_0 \wedge x_2 = (x_0 < y_0 \wedge x_1 < y_0)\ ?\ (x_1 + 1)\ :\ x_1 \wedge \neg(x_0 < y_0 \wedge x_1 < y_0 \wedge x_2 < y_0)$$

$$\mathcal{E} \mathrel{\hat{=}} x_2 < y_0$$

If the arithmetic and comparison operations and the memory model respect the semantics of C, the unwinding contains an assertion violation exactly if $\mathcal{B} \wedge \mathcal{E}$ is satisfiable.

Bit-vector logic is typically used to model numeric variables in C. Let $var(\varphi)$ be the set of variables in a formula $\varphi$. Let $\mathbb{B}$ be the set of values that variables can take. An *assignment* to a formula is a function $\sigma : var(\varphi) \to \mathbb{B}$.

### C. Backward Reachability Computation

A classic approach to reasoning about programs is to approximate the set of reachable states. We recall reachability analysis because BLITZ combines BMC with ideas from reachability analysis.

A *symbolic encoding* is a representation of sets of states by data structures such as logical formulae. Let `P` be a program with a transition relation $T$, a set $Init$ of initial states and a set $Err$ of error states. We write $\bar{x}$ for a sequence of variables. The initial and error states are represented symbolically by the formulae $Init(\bar{x})$ and $Err(\bar{x})$. The transition relation generates a formula $T(\bar{x}, \bar{y})$, in which $\bar{x}$ and $\bar{y}$ are of equal length. The *precondition transformer* defined below maps a formula $S(\bar{x})$ to a formula, called a *precondition* representing the predecessors of states represented by $S(\bar{x})$.

$$pre : Form \to Form \quad pre(S(\bar{x})) \mathrel{\hat{=}} \exists \bar{x}.T(\bar{y}, \bar{x}) \wedge S(\bar{x})$$

The set of *backward reachable states* $BReach(S(\bar{x}))$ contains those that lead to a state in $S(\bar{x})$. Below, we use the notation $f^0$ for the identity function and $f^{i+1}$ for the function mapping $x$ to $f(f^i(x))$. The backward reachable states have the well known characterisation below.

$$BReach(Err(\bar{x})) \iff \bigvee_{i \in \mathbb{N}} pre^i(Err(\bar{x}))$$

The equivalence suggests a naïve approach to computing states that lead to an error. The iterative computation terminates when a predicate that is a fixed point is reached. A *fixed point* of $Reach$ is a predicate $F(\bar{x})$ satisfying the condition:

$$Reach(F(\bar{x})) \iff F(\bar{x})$$

A formula $Q(\bar{x})$ is an *underapproximate precondition* if $Q(\bar{x}) \implies pre(S(\bar{x}))$. The notions for overapproximations are similarly defined.

## IV. Compositional Bounded Model Checking

In this section we present an approach for finding deep bugs by combining BMC with approximate preconditions. An obstacle to scalability of BMC is that the size of a BMC instance grows beyond the capacity of solvers as the unwinding depth is increased. Reachability analysis with quantifier elimination faces similar computational obstacles. We combine BMC instances with underapproximate preconditions to scale BMC without producing false alarms about bugs.

### A. Underapproximating Precondition Computation

We describe the use of SAT solvers to underapproximate preconditions with respect to a BMC instance and an error condition. Suppose we have a procedure P and a condition $\psi$ that should hold after P executes. The formula $\psi$ represents a set of states from which an error state can be reached. The set of predecessors $pre(\psi)$ represents all states that lead to a state in $\psi$ through P. Computing this set is expensive, so we compute an approximation. Program analysers typically over-approximate postconditions and may produce spurious results about bugs. We compute underapproximate preconditions to avoid spurious outcomes.

Our first observation is that the values of input variables in a satisfying assignment to a BMC instance dictate the value of other variables in the formula. Suppose a BMC instance $\mathcal{B}(\mathtt{P}) \wedge \mathcal{E}(\mathtt{P})$ has four variables $w, x, y, z$, with the input variables being $x$ and $y$. A satisfying assignment of the form below can be viewed as a formula and this formula implies the sub-formula that ranges only over input variables (abbreviating `true` and `false` as t and f respectively).

$$(w{:}\mathsf{t}, x{:}\mathsf{f}, y{:}\mathsf{f}, z{:}\mathsf{f}) \rightsquigarrow w \wedge \neg x \wedge \neg y \wedge \neg z \implies \neg x \wedge \neg y$$

Even though the sub-formula over input variables is weaker than the assignment, it is sufficient to constrain the values of all other variables, and hence is a *precondition for an error*. For the example above, the formula

$$\neg x \wedge \neg y \wedge \mathcal{B}(\mathtt{P}) \wedge \mathcal{E}(\mathtt{P})$$

has only one satisfying assignment because the values of $x$ and $y$ constraint the values of all other variables. The lemma below states this observation formally. We treat an assignment $\sigma$ also as a formula and write $inp(\sigma)$ for the sub-formula of $\sigma$ that contains only input variables.

**Lemma 1.** *If $\mathcal{B}(P) \wedge \mathcal{E}(P)$ is satisfied by an assignment $\sigma$, the formula $inp(\sigma) \wedge \mathcal{B}(P) \wedge \mathcal{E}(P)$ has a unique satisfying assignment. Consequently, the formula $inp(\sigma) \wedge \mathcal{B}(P) \wedge \neg\mathcal{E}(P)$ is unsatisfiable.*

*Proof.* Consider the satisfying assignment $\sigma$ and the formula $\mathcal{B}(\mathtt{P})$. The semantics of $inp(\sigma) \wedge \mathcal{B}(\mathtt{P})$ is equivalent to replacing every input variable in $\mathcal{B}(\mathtt{P})$ by its truth value in $\sigma$. After such a replacement, only non-input variables remain, which by definition occur on the left-hand sides of assignment statements. Since the formula is generated by a program in SSA form, each non-input variable is assigned only once, and hence has a unique value for the whole formula. It follows that $inp(\sigma) \wedge \mathcal{B}(\mathtt{P}) \wedge \mathcal{E}(\mathtt{P})$ has a unique satisfying assignment.

For the second part, observe that $\sigma$ satisfies $\mathcal{E}(\mathtt{P})$ and so does not satisfy $\neg\mathcal{E}(\mathtt{P})$. The constraints on input variables are determined by $\mathcal{B}(\mathtt{P})$ so $inp(\sigma) \wedge \mathcal{B}(\mathtt{P})$ must be uniquely satisfied by $\sigma$, so the conjunction $inp(\sigma) \wedge \mathcal{B}(\mathtt{P}) \wedge \neg\mathcal{E}(\mathtt{P})$ is unsatisfiable. □

The practical value of Lemma 1 is that we can derive an unsatisfiable formula from a satisfiable formula. A proof-generating SAT solver can generate a resolution refutation for a formula that is unsatisfiable. The refutation makes explicit what properties of the program are used to prove unsatisfiability and this information has numerous applications. Specifically, if an input variable does not appear in a refutation, the value of that variable does not affect the satisfiability of the restricted formula. We can use information from proofs to derive an underapproximation of a precondition that is more general than restricting an assignment to input variables.

**Algorithm 1:** UNDERAPPROXIMATE PRECONDITION

```
approx-pre (F: a code fragment
    ψ: a postcondition for F
    σ: a satisfying assignment to B(F) ∧ E(F) ∧ ψ
w: a weakening parameter)
    i := 0
    φ' := inp(σ, B(F))
    repeat
        φ := φ'
        (res, Π) := solve(φ ∧ B(F) ∧ E(F) ∧ ¬ψ)
        φ' := generalise(φ, Π)
        i := i + 1
    until φ' = φ or i = w
    return φ
solve (φ: a formula)
    if φ is satisfiable then
        Let σ be a satisfying assignment
        return (sat, σ)
    else
        Let Π be a proof of unsatisfiability
        return (unsat, Π)
```

To return to the example, suppose the constraint over input variables is $\neg x \wedge \neg y$ and the variable $x$ does not occur in the refutation (with or without negation). We know that $x$ is not required to prove unsatisfiability of $\neg x \wedge \neg y \wedge \mathcal{B}(\mathtt{P}) \wedge \neg\mathcal{E}(\mathtt{P})$, so we can conclude that $\neg y \wedge \mathcal{B}(\mathtt{P}) \wedge \neg\mathcal{E}(\mathtt{P})$ is also unsatisfiable. Recall that if $A \wedge B$ is unsatisfiable, then $A$ implies $\neg B$. In this example $\neg y \wedge \mathcal{B}(\mathtt{P})$ implies $\mathcal{E}(\mathtt{P})$, meaning that the set of states represented by $\neg y$ only leads to states that contain the error. Since $\neg x \wedge \neg y$ implies $\neg y$, we have generalised the precondition for the error.

The idea above is used to underapproximate preconditions. The reasoning we have sketched above does not require us to start with an assignment. It can be used to take a set of states that lead to an error and derive a larger set that only contains paths to an error. The restriction to states leading to an error is important to avoid spurious results about bugs.

The algorithm for underapproximate precondition computation is shown in Algorithm 1. We encapsulate interaction with the SAT solver by the method solve which takes as input a formula $\varphi$ and returns a pair $(\mathsf{res}, \theta)$, where res is the result of the satisfiability check. If $\varphi$ is satisfiable, $\mathsf{res} = \mathsf{sat}$ and $\theta$ is a satisfying assignment. Otherwise, $\mathsf{res} = \mathsf{unsat}$ and $\theta$ is a proof of unsatisfiability.

The procedure approx-pre takes as input a program fragment F, a postcondition that F must satisfy in order to reach an error, a satisfying assignment $\sigma$ to the formula $\mathcal{B}(\mathtt{F}) \wedge \mathcal{E}(\mathtt{F}) \wedge \psi$ and a parameter $w$. An *underapproximate precondition* is a formula

**Algorithm 2:** ADJUSTABLE, INCREMENTAL BMC INSTANCE CONSTRUCTION

```
next-inst (F: a code fragment
    ψ: a condition for error
d: a decomposition parameter)
    Frag := prev-code(F, d)
    nObj := ∅
    repeat
        foreach fragment G in Frag do
            Remove G from Frag
            if G defines variables in ψ then
                Add G to nObj
            else
                Frag := Frag ∪ prev-code(G, d)
    until Frag = ∅
    return nObj
prev-code (F: a code fragment, d: a decomposition parameter)
    Frag := ∅
    if d = proc then
        Add callers of F in the call graph to Frag
    else if d = stmt then
        Add statements before F in the CFG to Frag
    return Frag
```

$\phi$ satisfying the condition:

$$\phi \wedge \mathcal{B}(\mathtt{F}) \wedge \mathcal{E}(\mathtt{F}) \implies \psi$$

Every execution through F that starts in a state in $\phi$ leads to a state in $\psi$. The procedure approx-pre returns an underapproximate precondition. The condition is approximate because it may not be the weakest such $\phi$. Since $\psi$ is a condition for reaching an error, the formula $\phi$ is a precondition for reaching an error.

The parameter $w$ is used to weaken an underapproximate precondition. If $w$ is 0, the underapproximate precondition is $inp(\sigma)$. If $w$ is 1, the formula $inp(\sigma)$ is weakened using information from an unsatisfiability proof. We have considered two variants for the procedure generalise. One variant is to weaken $\phi$ by removing all variables that do not occur in the proof $\Pi$. To do this, we first walk the proof backwards to eliminate unnecessary deductions. Further techniques for computing minimal unsatisfiable cores of unsatisfiable formulae can also be used [13].

**Lemma 2.** *The formula* approx-pre$(\mathtt{P}, \psi, \sigma, w)$ *is an underapproximate precondition of* P *with respect to* $\psi$*.*

*Proof.* The invariant of the loop in the algorithm is that $\phi \wedge \mathcal{B}(\mathtt{F}) \wedge \mathcal{E}(\mathtt{F}) \wedge \neg\psi$ is unsatisfiable. This is because the generalisation step only reduces an unsatisfiable formula to a smaller unsatisfiable formula and projects out the portion of $\phi$ in this smaller formula. It follows that $\phi \wedge \mathcal{B}(\mathtt{F}) \wedge \mathcal{E}(\mathtt{F})$ implies $\psi$, so $\phi$ is a precondition of $\psi$. □

### B. Incremental, Data-Flow-Based Instance Construction

If a BMC instance is satisfiable, we use approx-pre to construct an underapproximate precondition. To determine if there is indeed a bug, we need to propagate the precondition to the entry point of the program. The standard approach to propagate facts through a program is to use the Control Flow Graph (CFG). Propagation is wasteful if information is propagated through code not relevant for reaching an error.

We use the term *fragment* for a piece of code that is either a procedure or a configurable number of sequential statements. BMC instances are generated incrementally using a granularity specified by the user. A code fragment F *defines* a variable x if x occurs on the left-hand side of an assignment in F and *uses* the variable if the variable occurs in an expression. In program analysis, a *use-def* graph makes data-flow explicit.

We combine both data- and control-flow information to prune the number of instances that are generated. Given a condition for an error, we only use fragments that define variables in that condition to generate BMC instances. We show in the experiments section that this seemingly simple optimisation has significant impact on the number of instances generated, and consequently on the size of programs that can be analysed.

Instances are incrementally constructed using Algorithm 2. The procedure prev-code takes as input a fragment F and a parameter $d$ which specifies the granularity of fragment. It returns a set of fragments that execute before F. At the granularity of procedures, we return the nodes before F in the call graph of the program. At the granularity of statements, we return statements preceding F in the CFG.

The procedure next-inst takes as input a fragment F that has already been analysed, a formula $\psi$, which is a precondition for F to reach an error, and a decomposition parameter $d$, which specifies the granularity of fragments. It returns a set of fragments that precede F and which define variables in $\psi$. It is implemented by iteratively calling prev-code until only fragments manipulating variables in $\psi$ are obtained.

### C. The BLITZ Algorithm

BLITZ combines underapproximate precondition computation with incremental construction of BMC instances to decompose the problem of finding a bug in a large instance to that of solving a sequence of smaller BMC instances. The advantage is that every instance fits in memory and that a smaller number of instances is considered than generating instances based on control flow. The algorithm is parameterised, so that the user may fine-tune the settings used for decomposition and precondition computation using domain-specific knowledge or data from benchmarks.

The procedure BLITZ in Algorithm 3 takes as input a program P which contains assertions, an unwinding bound $k$ for loops and recursive procedures, a weakening parameter $w$ for precondition approximation, and a decomposition granularity $d$. The algorithm returns an input vector $\bar{v}$ if there exists an execution of P with values specified by $\bar{v}$ that leads to an assertion violation. If no violation is found it may be because no error exists in $k$ unwindings, or because relevant states were missed by the underapproximation.

The main data-structure in BLITZ is a set of *obligations* $Obj$. An *obligation* contains a fragment F and a formula $\psi$

**Algorithm 3:** THE BLITZ ALGORITHM

```
BLITZ (P: a program,
       k: a bound for BMC
       w: a weakening parameter
   d: a decomposition granularity)
    Obj := ∅ //BMC obligations

    foreach fragment F containing an assertion do
        Add (F, E(F)) to Obj
    repeat
        Remove (F, ψ) from Obj
        (res, σ) := solve(B(F) ∧ E(F) ∧ ψ)
        if F is the program entry point then
            return "Violation triggered by input inp(σ)"
        else if res = sat then
            φ := approx-pre(F, ψ, σ, w)
            Obj := Obj ∪ next-inst(F, φ, d)
    until Obj is empty
```

representing a condition for an error. The initial value of $\psi$ is the set of assertion violations possible in a program. BLITZ proceeds by solving the instance generated by each fragment in $Obj$. In addition to the behaviour and error constraint, we also use the precondition $\psi$ as an error constraint.

If a BMC instance is unsatisfiable, it is eliminated from $Obj$ because its behaviour is no longer relevant for finding a bug. Otherwise, the instance could be satisfiable either because the error is reachable or because the context in which the fragment F executes is not taken into account. We first underapproximate the error precondition of F using the method given earlier and then propagate this precondition to other fragments in the program. The subsequent fragments are determined by following the control- and data-flow of the program.

There are two outcomes which make BLITZ terminate. If the entry point of the program is reached and the precondition obtained is satisfiable, a satisfying assignment $\sigma$ defines an input vector $inp(\sigma)$ which can be used to trigger an error. The other outcome is an empty set of obligations. This can occur either if no assertion is violated, or if the underapproximations have lost information about states leading to an error. In the latter case, there is nothing conclusive to report.

The precondition weakening approach proposed in Algorithm 1 can be viewed as an *eager* weakening approach that eagerly weakens each precondition up to a pre-determined bound $w$. We extend BLITZ with a *lazy* approach that avoids having to guess the appropriate weakening bound. The lazy approach starts by weakening each precondition once. If the entry of the program is reached with inconclusive results, BLITZ increments $w$ and backtracks to further weaken a previously computed precondition. The weakening sites are chosen based on data-flow and only applied if a precondition is not the weakest precondition.

**Theorem 1.** *If* BLITZ$(P, k, w, d)$ *returns an input vector, an assertion violation is reachable in* P*.*

## V. IMPLEMENTATION AND EVALUATION

### A. Implementation and Comparison to Other Tools

We implemented BLITZ within the CProver framework, which is the basis of the bounded model checker CBMC, which targets ANSI-C [5]. To extract proofs of unsatisfiability, we use a proof-logging version of the SAT solver MINISAT [7].

Our implementation uses several practical optimisations to improve upon the compositional BMC algorithm we presented. For one, we interleave unwinding and formula generation, to avoid storing large structures in memory. This optimisation influences the architecture of implementation. We try to avoid recomputing information about procedures whenever possible. When a procedure has different callees, we also try to reuse preconditions that were computed earlier, if relevant. This optimisation is a rather weak form of procedure summarisation used in program analysis.

We compare BLITZ to other tools at a conceptual level. Tools with the similar goal of bug-finding and which we compare with are CBMC, ESBMC and CORRAL. Architecturally, CBMC and ESBMC are similar — both tools inline all procedures into a single BMC instance. Like BLITZ, CORRAL combines bounded model checking with nondeterminism to bound the scope of analysis, and is competitive with the state-of-the-art [11]. Unlike BLITZ, CORRAL inlines procedures on demand, instead of propagating preconditions. For solvers, CORRAL and ESBMC use Z3 [6], an efficient SMT solver; CBMC and BLITZ use the CPROVER framework, which directly reduces a BMC instance to SAT and solves it with a SAT solver. Section VI contains a more detailed discussion of the algorithmic content of these tools.

### B. Benchmarks

We evaluate BLITZ on the Bugbench data set to measure its performance in detecting known vulnerabilities. Bugbench is an independent benchmark suite for systematic evaluation of bug detection tools [12]. It aims to be a representative collection of real-world C programs with known bugs. In our evaluation, we used the vulnerability subset of the suite that focuses on memory safety violations, the most critical type of bugs. Our results are in Section V-C.

We also applied BLITZ to production software from the Internet Systems Consortium to search for unknown vulnerabilities. The ISC develops and distributes critical Internet infrastructure software to Internet providers worldwide[1]. We found new vulnerabilities in multiple production releases of ISC programs, currently deployed in the Internet. We discuss these results in Section V-D.

In Section V-E, we evaluate BLITZ under configurations regarding propagation (control vs data-flow) and precondition weakening. We find that data-flow based propagation is key to BLITZ's scalability. We also find that the lazy approach to weakening preconditions works better than the eager approach.

All experiments were performed on Intel Xeon 2.93 GHz machines with 32GB RAM.

[1]ISC software is deployed by over 80% of Internet providers worldwide (http://www.isc.org/).

### C. Evaluation on the Bugbench Vulnerability Benchmarks

The vulnerability benchmarks in Bugbench consist of 7 programs `polymorph`, `ncompress`, `man`, `gzip`, `bc`, `squid` and `cvs`. All benchmarks are labelled with the location and type of vulnerability: stack overflow, global overflow, heap overflow and double free. We list these programs in Table I with the labels B1-8, and indicate their sizes and type of vulnerability they contain.

Into each program, we inserted an assertion that if violated would trigger the vulnerability. We also statically determined the minimum unwinding $k$ required to reach the vulnerability. This value is shown in the table and the same value was used with all tools. We then used CBMC, ESBMC, CORRAL and BLITZ to check for assertion violations. CBMC and ESBMC support slicing, which reduces the size of BMC instances. We ran these tools with slicing turned on. All tools terminate once an assertion violation is found, or report no violation on termination. We used a timeout of 24 hours (86400s).

As listed in the table, an unwinding bound of 1-3 was sufficient to find most vulnerabilities. The only exception was B2, where an unwinding of $k = 2048$ was required to reach a buffer overflow. This was the number of iterations to traverse a buffer of that size. We ran BLITZ with data-flow driven propagation and lazy weakening of preconditions. We also fixed the granularity of each analyzed code fragment in BLITZ at the level of a procedure in all our experiments. Thus, each analyzed BMC instance corresponds to a single procedure.

The evaluation results are in rows B1-8 of Table I. CBMC and ESBMC produced similar results, so we omit ESBMC from Table I to conserve space.

All tools find a vulnerability in benchmarks B1, B3, B4 and B5, with BLITZ requiring the least amount of time. BLITZ is the only tool that does not time out and finds vulnerabilities in B2, B6, B7 and B8. The maximum time required by BLITZ was 35.47 minutes to find a double-free vulnerability in `cvs`. The time and memory required by BLITZ increases with the size of the benchmark and unwinding bound but we did not observe a timeout or memory exhaustion.

The experiments demonstrate the utility of underapproximate preconditions for bug-finding. The underapproximate preconditions did not usually require more than one weakening, with the exception of B6, where BLITZ backtracked to weaken the precondition 11 times. The results suggest that our compositional BMC approach is effective for finding violations on real-world programs of significant sizes.

CBMC and BLITZ have similar performance on the smallest benchmark (B1). Both tools also found the bug in B3. On benchmarks B2, B4, B5 and B6, CBMC runs out of memory memory, and on benchmarks B7 and B8, CBMC exceeded the 24-hr timeout.

CORRAL has performance similar to BLITZ on benchmarks B1, B3, B4 and B5, and outperforms CBMC on benchmark B3. However, CORRAL exceeded the 24-hr timeout on benchmarks B2, B6 and B7, and terminated prematurely without finding the vulnerability on the largest benchmark B8.

### D. Evaluation on ISC Programs

We now evaluate BLITZ on open-source production software from the Internet Systems Consortium (ISC). We use the current production release of `aftr v1.1` and `sntp v4.2.6p5`. The Address Family Translation Router (`aftr`) plays a central role in transitioning the global Internet from IPv4 to IPv6 — it provides backwards compatibility to transport IPv4 traffic over IPv6 carrier infrastructure. The transitioning process was launched on June 6, 2012[2] and is currently ongoing, using the version of `aftr` in this evaluation. The program `sntp` is a reference implementation of the Simple Network Time Protocol that provides clock synchronisation over the Internet.

We focus on finding Null pointer dereference vulnerabilities, which are not represented in Bugbench. To check for Null pointer dereferences, we use the default Null pointer dereference assertions inserted by CPROVER. Since we have no knowledge of any vulnerability in the ISC programs, we used an unwinding depth of $k = 1$ for all the tools.

The results are rows I1-2 of Table I. Again, ESBMC produces similar results to CBMC. CBMC quickly ran out of memory on these programs. CORRAL and BLITZ separately found the same vulnerabilities within minutes. We manually verified the results and have confirmed these vulnerabilities with the ISC, who will be patching their software for redistribution to affected Internet providers worldwide.

We briefly describe the two vulnerabilities in the following paragraphs.

**`aftr`.** A pointer `ss1` is dereferenced after a function call `ss1 = stdio_open()`. The function `stdio_open()` calls `fileno(stdin)` in its body, and returns `NULL` if `fileno(stdin)` returns -1 on stream failure. An attacker with control over the program's environment `stdin` will be able to exploit the Null pointer dereference.

**`sntp`.** A pointer `bcastaddr` is dereferenced after being allocated a buffer through a call to `realloc`. However, `realloc` could return `NULL` when memory allocation fails. An attacker who is able induce memory allocation failure will be able to exploit the Null pointer dereference.

### E. Evaluation of BLITZ Features

We now evaluate BLITZ using different configurations for propagation and precondition weakening. First, we compare the effect of using data-flow for propagating preconditions to that using control-flow on performance. Second, we compare the performance of an eager and a lazy weakening strategy. In eager weakening, preconditions are weakened several times before propagation, while in lazy weakening, all propagation is performed first and preconditions are weakened only if an inconclusive result is obtained at the entry of the program. The timeout for the experiments was 24 hours (86400s).

*Data vs Control-flow Propagation:* Fig. 3 compares the running time between using precondition-guided data-flow propagation and control-flow based propagation. In 6 of 10

[2]The Internet Society declared June 6, 2012 the World IPv6 Launch day.

| L. | Program | KLOC | Vulnerability | $k$ | CBMC | | | CORRAL | | | BLITZ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Time (s) | Mem. (MB) | Bug | Time (s) | Mem. (MB) | Bug | #W | #R | Time (s) | Mem. (MB) | Bug |
| *Known Vulnerabilities — BugBench Benchmark* | | | | | | | | | | | | | | | |
| B1 | poly | 0.7 | Stack Overflow | 1 | 1 | 21 | ✓ | 2 | 10 | ✓ | 1 | 2 | 1 | 18 | ✓ |
| B2 | poly | 0.7 | Global Overflow | 2048 | 2427 | >32768 | ✗ | >86400 | 8828 | ✗ | 1 | 3 | 959 | 10165 | ✓ |
| B3 | ncomp | 1.9 | Stack Overflow | 1 | 12 | 258 | ✓ | 6 | 32 | ✓ | 1 | 3 | 6 | 506 | ✓ |
| B4 | man | 4.7 | Global Overflow | 2 | 1692 | >32768 | ✗ | 16 | 119 | ✓ | 1 | 3 | 8 | 156 | ✓ |
| B5 | gzip | 8.2 | Global Overflow | 1 | 1567 | >32768 | ✗ | 20 | 384 | ✓ | 1 | 3 | 5 | 67 | ✓ |
| B6 | bc | 17.0 | Heap Overflow | 1 | 4733 | >32768 | ✗ | >86400 | 212 | ✗ | 11 | 10 | 258 | 2563 | ✓ |
| B7 | squid | 93.5 | Heap Overflow | 1 | >86400 | 15462 | ✗ | >86400 | 22715 | ✗ | 1 | 4 | 1664 | 9024 | ✓ |
| B8 | cvs | 114.5 | Double Free | 3 | >86400 | 4301 | ✗ | 203 | 56 | ✗ | 1 | 30 | 2128 | 3871 | ✓ |
| *Discovered New Vulnerabilities — ISC Programs* | | | | | | | | | | | | | | | |
| I1 | aftr | 13.3 | Null Ptr Deref. | 1 | 664 | >32768 | ✗ | 85 | 482 | ✓ | 1 | 2 | 74 | 959 | ✓ |
| I2 | sntp | 42.1 | Null Ptr Deref. | 1 | 5479 | >32768 | ✗ | 25 | 136 | ✓ | 2 | 3 | 14 | 227 | ✓ |

TABLE I: Comparison of CBMC, CORRAL and BLITZ on benchmarks. Rows B1-8 contain the evaluation results on known vulnerabilities in the Bugbench benchmark, while rows I1-2 contain the evaluation results on programs from the Internet Systems Consortium (ISC), where BLITZ and CORRAL discovered new vulnerabilities. The "L." column lists the benchmark labels. The "KLOC" column indicates the size of each benchmark program in thousands of lines of code. The "$k$" column lists the (statically determined) minimum number of (loop and recursion) unwindings required to trigger the violation condition in each benchmark. The "Time(s)" column indicates the length of time for which the tool ran with >86400 indicating that the tool exceeded the 24-hour timeout. The "Mem.(MB)" column shows the maximum amount of memory used by the tool in MB. We write >32768 to indicate that the tool exceeded 32GB of memory. The "Bug" column indicates whether the vulnerability was found by the tool on termination. The "#W" column indicates the number of precondition *weakening* passes used in BLITZ, detected automatically using the lazy approach. The "#R" column shows the number of procedure *refinements* in the propagation chain that found the bug.

benchmarks, control-flow based propagation leads to a timeout. Barring one benchmark on which the two techniques perform equally, data-flow based propagation is always superior.

The results suggest data-flow based propagation is key to scalability of BLITZ. Existing tools such as CORRAL [11] and ALTER [21] follow control-flow and inline procedures on demand. Using information in preconditions to construct instances allows BLITZ to sidestep code that is irrelevant for detecting a violation. For example, in one of the largest programs (`squid`), a single call to the procedure `comm_close` would have required the analysis of up to 715 million unique procedure instances if control-flow-based inlining was used. BLITZ however did not analyze the `comm_close` procedure.

*Lazy vs Eager Weakening:* We compare lazy and eager weakening of underapproximate preconditions. A lazy approach starts the analysis using a weakening bound 1. When BLITZ terminates with an inconclusive result, the weakening bound is incremented, and BLITZ backtracks to the first computed precondition on the data-flow path that is not equivalent to the weakest precondition. We configured the eager approach to use 11 passes, since this was the upper bound detected by the lazy approach (see Table I).

Fig. 4 compares the effect of lazy and eager weakening on running time. The lazy approach leads to shorter running times because weakening a precondition twice suffices to discover a bug in most benchmarks.

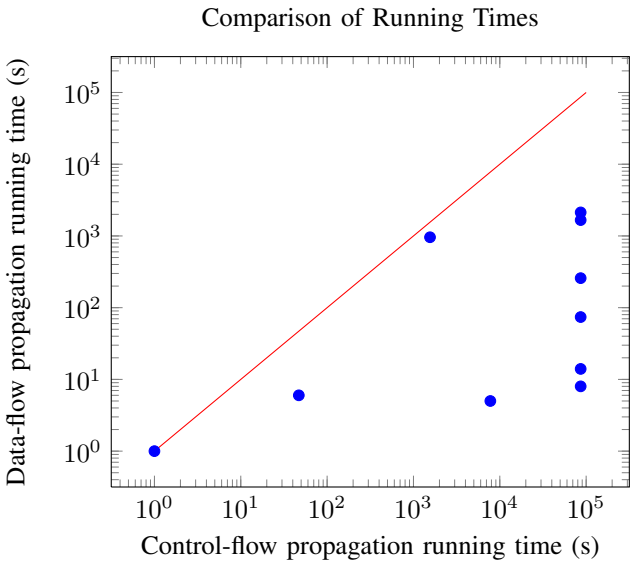


Fig. 3: A comparison of BLITZ running times with data-flow against control-flow propagation, with 24-hr timeout.

## VI. RELATED WORK

This paper lies at the intersection of model checking and program analysis. Of several techniques to automatically compute information about a program, SLAM is notable for its success as a software model checker [3]. SLAM requires repeated calls to a theorem prover to construct abstractions and refines the abstraction using counterexamples. One refinement strategy uses weakest preconditions, but these are computed over single paths, unlike program fragments as in our case.

The compositional approach to system analysis is key for overcoming the complexity of any real-world system. We focused on composition defined with respect to sequential composition and procedure calls, as in Hoare logic [9]. Compositionality is exploited in program analysis [8] and symbolic execution [1], but we are not aware of it being used in BMC to date. The BLITZ algorithm is composition in the same sense, but differs from these above approaches because it uses data-flow to further reduce the decomposition of the program.

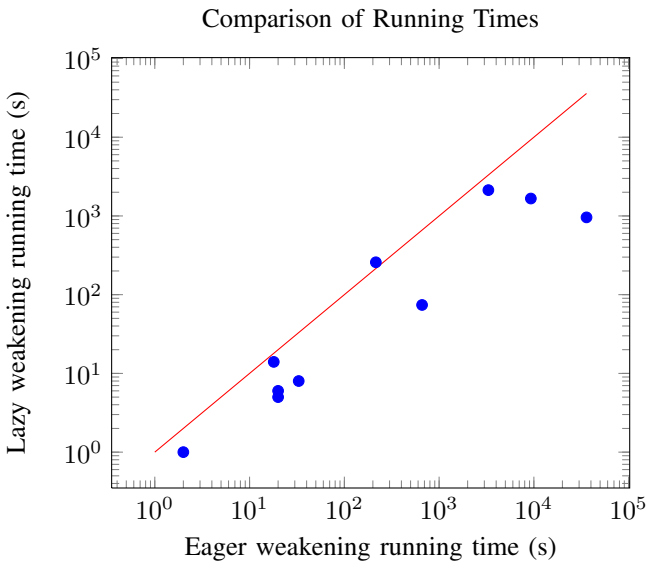


Fig. 4: A comparison of BLITZ running times with lazy against eager weakening of preconditions, with 24-hr timeout.

A key technique for relevance-driven program analysis is Craig interpolation, which has numerous applications in formal verification. Interpolation was applied to over-approximate reachable states in hardware model checking in [15], to infer preconditions in [16], [10], and to learn annotations from failed explorations to improve backtracking efficiency [17], [21]. We do not use standard techniques for computing interpolants from proofs because these lead to large formulae and consume more memory. The preconditions we generate do satisfy the interpolation condition and can be viewed as a restricted type of interpolants that have compact representations in addition to satisfying the usual vocabulary and implication conditions.

Our approach of generalising satisfying instances is in the same spirit as IC3. IC3 is a technique for incremental construction of inductive proofs for modular analysis [4]. Unlike IC3, BLITZ only seeks to discover assertion violations and does not attempt proofs of correctness.

There are two main strategies to apply scope-bounded analysis. One strategy partitions a program into multiple sub-programs at the outset (each with a subset of paths), and solve all of them in parallel [20]. Though amendable to a parallel architecture, it may generate a prohibitively large number of sub-programs. Another strategy is to overapproximate behaviour outside the scope of bounded analysis, and iteratively refine it. Called 'structural abstraction' in CALYSTO [2], it also used in CORRAL [11] and ALTER [21]. DC2 further uses a lightweight static analysis to infer pre- and post-conditions to model the behavior of procedures outside the scope of analysis, but its scope expansion is not automatic. CORRAL automates forward (callee) scope expansion with stratified inlining of procedures and selective variable abstraction. ALTER alternates between eager forward (callee) and lazy backward (caller) expansion, and uses interpolation to learn procedure preconditions that lead to failed scope expansions, to prevent re-exploration on backtracking. BLITZ recasts satisfying assignments to obtain unsatisfiable formulae, and then derives intermediate preconditions from unsatisfiability proofs. Unlike existing approaches where the size of a BMC instance grows as more procedures are *inlined*, BLITZ can bound the size of instances.

Slicing is commonly used to reduce the size of BMC instances by removing variables that are *syntactically* irrelevant to a property (e.g., [19], [18]). It is also known as 'cone of influence' reduction, where the set of relevant variables expands with the size of the slice, diminishing its effectiveness (e.g., despite slicing, CBMC and ESBMC frequently run out of time/memory). Beyond syntactic relevance, BLITZ also uses proofs of unsatisfiability to reason about *semantic* relevance on small code fragments at a time, producing an overall cone with much smaller base.

## VII. CONCLUSIONS AND FUTURE WORK

We presented BLITZ, a new, compositional bounded model checking algorithm for software and showed that it scales BMC to real-world programs. BLITZ is capable of finding all known vulnerabilities in a known data set and has also discovered new vulnerabilities in widely deployed software. The new discoveries have led to bug fixes which will soon be deployed.

BLITZ works by constructing a series of BMC instances that when composed lead to a violation. A novel feature of our algorithm is the use of underapproximate preconditions in the context of BMC. The underapproximations are computed using information from resolution refutations generated by a SAT solver. The underapproximation guarantees that the tool does not generate false alarms about the existence of bugs and the proofs guarantee that the approximation is not too restricted by considering only facts relevant for the violation. Our procedure is parametric, allowing for the preconditions to be iteratively weakened to eventually derive Dijkstra's weakest precondition, if resources permit. The size of code fragments analysed is configurable and can range from a single statement to an entire procedure, depending on code-size and memory available.

There are several directions to explore. We did not use Craig interpolation over proofs because the formulae were large. An interesting question is whether proofs can be manipulated to yield interpolants that have compact representations and whether this improves performance. A second question is how one may guess the optimal configuration of the algorithm (unwinding, weakening, granularity) by a preliminary analysis of the code. Third, concrete and symbolic representations have been combined in symbolic execution with great success but such an approach has not been applied to BMC. We anticipate that combining concrete values with BMC may provide new opportunities for decomposition.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is partially supported by NSF under grant 0831501 CT-L; by AFOSR under MURI award FA9550-09-1-0539; by ONR under MURI grant N000140911081; and by DARPA under award HR0011-12-2-005.

## REFERENCES


[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Conference on Tools and algorithms for the construction and analysis of systems*, TACAS, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 211–220, New York, NY, USA, 2008. ACM.

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI'01: Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213. ACM Press, 2001.

[4] Aaron R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 368–371, New York, NY, USA, 2003. ACM.

[6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS'08: Proc. of the 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[7] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[8] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. *SIGPLAN Not.*, 45(1):43–56, January 2010.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[10] Daniel Kroening and Georg Weissenbacher. Interpolation-based software verification with wolverine. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 573–578, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *Proceedings of the 24th international conference on Computer Aided Verification*, pages 427–443, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[13] Inês Lynce and João P. Marques Silva. On computing minimum unsatisfiable cores. In *Conference on Theory and Applications of Satisfiability Testing*. Online Proceedings, 2004.

[14] K. L. McMillan. Relevance heuristics for program analysis. In *Symposium on Principles of programming languages*, pages 145–146. ACM, 2008.

[15] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Conference on Computer Aided Verification*, pages 1–13, Berlin, Heidelberg, 2003. Springer-Verlag.

[16] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Conference on Computer Aided Verification*, CAV'06, pages 123–136, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification*, pages 104–118, 2010.

[18] Bruno Cuervo Parrino, Juan Pablo Galeotti, Diego Garbervetsky, and Marcelo F. Frias. A dataflow analysis to improve sat-based bounded program verification. In *Proceedings of the 9th international conference on Software engineering and formal methods*, SEFM'11, pages 138–154, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Danhua Shao, Divya Gopinath, Sarfraz Khurshid, and Dewayne E. Perry. Optimizing incremental scope-bounded checking with data-flow analysis. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 408–417, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Danhua Shao, Sarfraz Khurshid, and Dewayne E. Perry. An incremental approach to scope-bounded checking using a lightweight formal method. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 757–772, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] Nishant Sinha, Nimit Singhania, Satish Chandra, and Manu Sridharan. Alternate and learn: finding witnesses without looking all over. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, pages 599–615, Berlin, Heidelberg, 2012. Springer-Verlag.