All-Solution Satisfiability Modulo Theories: applications, algorithms and benchmarks

Quoc-Sang Phan and Pasquale Malacaria
School of Electronic Engineering and Computer Science
Queen Mary University of London
Mile End Road, London E1 4NS, United Kingdom
Email: {q.phan, p.malacaria}@qmul.ac.uk

Abstract—**Satisfiability Modulo Theories (SMT) is a decision problem for logical formulas over one or more first-order theories. In this paper, we study the problem of finding all solutions of an SMT problem with respect to a set of Boolean variables, henceforth All-SMT. First, we show how an All-SMT solver can benefit various domains of application: Bounded Model Checking, Automated Test Generation, Reliability analysis, and Quantitative Information Flow. Secondly, we then propose algorithms to design an All-SMT solver on top of an existing SMT solver, and implement it into a prototype tool, called aZ3. Thirdly, we create a set of benchmarks for All-SMT in the theory of linear integer arithmetic QF_LIA and the theory of bit vectors with arrays and uninterpreted functions QF_AUFBV. We compare aZ3 against MathSAT, the only existing All-SMT solver, on our benchmarks. Experimental results show that aZ3 is more precise than MathSAT.**

Keywords*—*Satisfiability Modulo Theories; Symbolic Execution; Bounded Model Checking; Automated Test Generation; Reliability Analysis; Quantitative Information Flow;

I. Introduction

Satisfiability Modulo Theories [1] (SMT) is a decision problem for logical formulas over one or more first-order theories. Given a first-order formula φ, an SMT solver will try to construct a model for φ, and return `SAT` if such a model is found.

The SMT solver MathSAT, from version 4 [2], provides a functionality, called All-SMT, such that given a formula φ and a set V_I of *important* Boolean variables, MathSAT in All-SMT mode computes all models of φ with respect to the set V_I. The All-SMT functionality has been used to compute predicate abstraction in [3] and [4]. It was also proposed for installation optimization in [5].

In this paper, we extend the All-SMT problem with a set V_R of *relevant*, possibly non-Boolean, variables. The extended All-SMT(φ, V_I, V_R) problem is to compute all models of φ with respect to the set V_I and the models include value assignments for variables in V_R. We show how this All-SMT problem can be used to analyse the correctness, reliability and security of programs:

- *Bounded Model Checking* [6]: SMT-based Bounded Model Checking can only return a single error trace, the user has to fix the error, then runs the model checker again for other error traces. This is because the SMT solver can return only one model. Combining Bounded Model Checking with an All-SMT solver, we can compute multiple counterexamples in one run of the model checker.
- *Automated Test Generation*: an All-SMT solver can be combined with a Bounded Model Checker or Symbolic Execution tool for automated test input generation. Although traditional Symbolic Execution with an SMT solver is already capable of generating test inputs, it needs to make hundreds or thousands of calls to the SMT solver. In our approach, the Symbolic Execution tool needs to make only one call to the All-SMT solver for any program.
- *Reliability analysis*: we can combine an All-SMT solver with a Symbolic Execution tool to enumerate all path conditions of the program, then use the Barvinok model counting technique [7] to compute the number of inputs that go into each symbolic path. In this way, we can compute the reliability of the program, which means the probability that the program successfully accomplishes its task without errors.
- *Quantitative Security* [8]: the problem of computing channel capacity, i.e. maximum leakage in a program, can be casted into #SMT, the problem of computing the number of models of φ with respect to a set V_I. An All-SMT solver thus can be used to compute channel capacity.

Although the All-SMT functionality of MathSAT has been used in [3] and [4], apart from a brief description in [5], its algorithm is not documented. When using MathSAT for our analysis, we found that MathSAT was imprecise in several benchmarks. Hence, we propose a lightweight technique for the implementation of an All-SMT front-end on top of an off-the-shelf SMT solver. Our technique is based on depth-first search on important variables, and it is more efficient than the straightforward and well-known blocking clauses method. We implement our technique into a prototype tool, called aZ3, built on top of the SMT solver z3 [9].

In order to evaluate aZ3 and MathSAT, we create two sets of benchmarks. The first one are formulas in the theory of linear integer arithmetic, QF_LIA [10], which are generated using the symbolic execution platform Symbolic PathFinder [11]. The second set of benchmarks are formulas in bit vector with array, QF_AUFBV [10], generated using the model checker CBMC [12] and case studies in the literature of Quantification of Information Leaks.

A. Contribution

In summary, our contribution is threefold:

- We propose four new applications of an All-SMT solver: (i) computation of multiple counterexamples for Bounded Model Checking, (ii) generation of test inputs using either Symbolic Execution or Bounded Model Checking, (iii) reliability analysis of programs, and (iv) measurement of insecurity of programs.
- We introduce a general and lightweight approach for solving All-SMT.
- We provide standard benchmarks for the evaluation of All-SMT solvers.

The rest of the paper is organized as follows. Section II provides mathematical preliminaries on first-order theories and a brief introduction of SMT solver and the DPLL algorithm. In Section III, we demonstrate the use of an All-SMT solver in different application domains. Then in Section IV, we propose a technique to build an All-SMT solver, and implement into the prototype tool aZ3. We evaluate aZ3 and MathSAT in Section V. Section VI discusses the related work, and Section VII concludes our work.

II. Preliminaries

A. Propositional logic

Definition 1 (Propositional atom): A propositional atom or Boolean atom is a statement or an assertion that must be true or false.

Examples of Boolean atoms are: "all humans are mortal" and "program `P` leaks k bits". Boolean atoms are the most basic building blocks of *propositional formulas*, each Boolean atom A_i is also a formula.

Propositional formulas are constructed from Boolean atoms using *logical connectives*: `not` ($\neg$), `and` ($\wedge$), `or` ($\vee$), and `imply` ($\rightarrow$). That means if φ_1 and φ_2 are formulas, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, and $\varphi_1 \rightarrow \varphi_2$ are also formulas. For example, $(\neg A_1 \wedge A_2) \rightarrow A_3$ is a propositional formula.

A Boolean atom A_i or its negation $\neg A_i$ is called a *literal.* We denote by *Atom*(φ) the set $\{A_1, A_2, \ldots A_n\}$ of Boolean atoms that occur in φ. The truth of a propositional formula φ is a function of the truth values of the Boolean atoms it contains.

We denote by $\top$ and $\bot$ the truth values of true and false, respectively. The set $\mathbb{B} = \{\top, \bot\}$ is called *Boolean domain.*

Definition 2: Given a propositional formula φ, a truth assignment μ of φ is defined as a function which assigns each Boolean atom of φ a truth value:

$$\mu : Atom(\varphi) \rightarrow \mathbb{B}$$

A partial truth assignment of a formula φ is a function $\mu : \mathcal{A} \rightarrow \{\top, \bot\}$ where $\mathcal{A}$ is any subset of *Atom*(φ). A (partial) truth assignment μ satisfies a propositional formula φ, denoted by $\mu \models \varphi$, if φ is evaluated to $\top$ under μ. For example $\mu : A_3 \mapsto \bot$ satisfies the formula $(\neg A_1 \wedge A_2) \rightarrow A_3$.

A formula φ is *satisfiable* if there exists a (partial) truth assignment such that $\mu \models \varphi$. If $\mu \models \varphi$ for every truth assignment μ, then φ is *valid*. Either a formula is valid or its negation is satisfiable.

Definition 3: A propositional formula φ is in Conjunctive Normal Form (CNF) if and only if it is a conjunction of disjunctions of literals:

$$\varphi = \bigwedge_{i=1}^{N} \bigvee_{j=1}^{M_i} l_{ij}$$

Any propositional formula can be converted to CNF by an algorithm with worst-case linear time [13], [14].

B. First-order logic

We assume countable sets of variable $\mathcal{V}$, function symbols $\mathcal{F}$ and predicate symbols $\mathcal{P}$. A first-order logic *signature* is defined as a partial function $\Sigma : \mathcal{F} \cup \mathcal{P} \rightarrow A$ ($A \subset \mathbb{N}$). Each $a \in A$ corresponds to an *arity* of an symbol. Obviously, a 0-ary predicate is a Boolean atom, and a 0-ary function symbol is called a *constant.*

A Σ-term τ is either a variable $x \in \mathcal{V}$ or it is built by applying function symbols in $\mathcal{F}$ to Σ-terms, e.g. $f(\tau_1, \ldots, \tau_n)$ where $f \in \mathcal{F}$ and $\Sigma(f) = n$. For example, $f(x, g(x))$ is a Σ-term if $\Sigma(f) = 2$ and $\Sigma(g) = 1$.

Definition 4: If $\tau_1, \ldots, \tau_n$ are Σ-terms, and $p \in \mathcal{P}$ is a predicate symbol such that $\Sigma(p) = n$, then $p(\tau_1, \ldots, \tau_n)$ is a Σ-atom.

A Σ-atom or its negation is called Σ-literal. We use the infix equality sign "=" as a shorthand for the equality predicate. If τ_1 and τ_2 are Σ-terms, then the Σ-atom $\tau_1 = \tau_2$ is called Σ-equality. $\neg(\tau_1 = \tau_2)$ or $\tau_1 \neq \tau_2$ is called Σ-disequality.

Σ-atoms are the most basic building blocks of Σ-formulas, each Σ-atom $p(\tau_1, \ldots, \tau_n)$ is also a Σ-formula. Similar to the construction of propositional formulas, Σ-formulas are constructed from Σ-terms which are connected by `universal quantifiers` ($\forall$), `existential quantifiers` ($\exists$), and logical connectives. That means if φ_1 and φ_2 are Σ-formulas, then $\forall x : \varphi_1$, $\exists x : \varphi_1$, $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, and $\varphi_1 \rightarrow \varphi_2$ are also Σ-formulas.

A *quantifier-free* Σ-formula does not contain quantifiers; a *sentence* is a Σ-formula without free variables. A first-order theory is defined as follows:

Definition 5 (First-order theory): A first-order theory $\mathcal{T}$ is a set of first-order sentences with signature Σ.

We call a Σ-atom in the theory $\mathcal{T}$ as $\mathcal{T}$-atom in short. A Σ-structure M is a triple $(|M|, \Sigma, \mathcal{I})$ consisting of a non-empty domain $|M|$, a signature Σ, and an interpretation $\mathcal{I}$. The interpretation $\mathcal{I}$ assigns meanings to symbols of Σ: for each function symbol $f \in \mathcal{F}$ such that $\Sigma(f) = n$, f is assigned a n-ary function $\mathcal{I}(f)$ on the domain $|M|$; for each predicate symbol $p \in \mathcal{P}$ such that $\Sigma(p) = n$, p is assigned a n-ary predicate $\mathcal{I}(p)$, represented by a subset of $|M|^n$. For each variable $x \in \mathcal{V}$, $\mathcal{I}(x) \in |M|$.

A Σ-structure M is a model of the Σ-theory $\mathcal{T}$ if it satisfies all sentences in $\mathcal{T}$. If a Σ-formula is satisfiable in a model of $\mathcal{T}$, then it is called $\mathcal{T}$-*satisfiable*.

Definition 6 (The SMT problem): Given a Σ-formula φ and a theory or a combination of theories $\mathcal{T}$, the Satisfiability Modulo Theories problem (SMT) is the problem of deciding $\mathcal{T}$-satisfiability of φ.

Most of the work on SMT focus on quantifier-free formulas, and *decidable* first-order theories. Following [15], we define a bijective function $\mathcal{BA}$ (*Boolean abstraction*) which maps Boolean atoms into themselves and Σ-atoms in theory $\mathcal{T}$ into fresh Boolean atoms. The *Boolean refinement* function $\mathcal{BR}$ is then defined as the inverse of $\mathcal{BA}$, which means $\mathcal{BR} = \mathcal{BA}^{-1}$.

Definition 7 (The All-SMT problem): Given a Σ-formula φ and a theory or a combination of theories $\mathcal{T}$, All-Solution Satisfiability Modulo Theories (All-SMT) is the problem of enumerating all models M of $\mathcal{T}$ with respect to a set of Boolean variables V_I such that φ is $\mathcal{T}$-satisfiable in M.

Here we also require that each of the resulting models includes the value assignments for all elements of the set V_R of *relevant* (possibly non-Boolean) variables.

C. The DPLL algorithm

At a high level, an SMT solver is the integration of two components: a SAT solver and $\mathcal{T}$-solvers. SMT solving can be viewed as the iteration of the two following steps. First, the SAT solver searches on the Boolean abstraction of the formula, $\varphi^P = \mathcal{BA}(\varphi)$, and returns a (partial) truth assignment μ^P. The $\mathcal{T}$-solvers then check the Boolean refinement of the candidate, $\mathcal{BR}(\mu^P)$, whether it is consistent with the theories $\mathcal{T}$.

```
function DPLL($\varphi$){
    $\mu \leftarrow$ TRUE;
    status = BCP($\varphi, \mu$);
    if (status = SAT) return SAT;
    else if (status = UNSAT) return UNSAT;
    while (TRUE) {
        $l$ = choose_literal($\varphi$);
        $\mu = \mu \wedge l$;
        status = BCP($\varphi, \mu$);
        if (status == SAT) return SAT;
        else if (status == UNSAT)
            if (all_states_are_explored())
                return UNSAT;
            else backtrack($\varphi, \mu$);
} }
```

Fig. 1. DPLL algorithm on the propositional formula φ

The dominant approach for SAT solvers is the DPLL family of algorithms [16]. The simplest form of DPLL is depicted in Figure 1, its input is a propositional formula in Conjunctive Normal Form (CNF), which means φ takes the form:

$$\varphi = \bigwedge_{i=1}^{N} \bigvee_{j=1}^{M_i} l_{ij}$$

A (finite) disjunction of literals $(l_1 \vee l_2 \cdots \vee l_k)$ is called a clause, and a literal l_i is an atom or its negation. A clause that contains only one literal is called a *unit clause*. At a high level, DPLL is a stack-based depth-first search procedure which iteratively performs the two following steps: first, choose a literal l_i from the remaining clause, then add it to the current truth assignment; secondly, apply Boolean Constraint Propagation (BCP), and backtrack if there is a conflict. These two steps are repeated until a model is found or all states are explored without finding a model.

The procedure BCP for a literal l_i removes all the clauses containing l_i, and removes $\neg l_i$ from the remaining clauses. If the removals result an empty clause, the search encounters a conflict.

III. Applications

In this section, we discuss some of the applications of All-SMT that have not been studied prior to this work.

A. Multiple Counterexamples for BMC

The aim of Bounded Model Checking [6] (BMC) is to find bugs or to prove their absence up to some bounded *k* number of transitions. In BMC, a program *P* is modelled as a transition system:

$$P = (S, I, F, T)$$

where S is the set of program states; $I \subseteq S$ is the set of initial states; $F \subseteq S$ is the set of final states; and $T \subseteq S \times S$ is the transition relation. Under this setting, a trace of execution of the program P is represented by a sequence of states: $\sigma = s_0 s_1 .. s_k$ such that $s_0 \in I, s_k \in F$ and $\langle s_i, s_{i+1} \rangle \in T$ for all $i \in \{0, .., k-1\}$.

A state s_e is called an error state if the program reaches an error, e.g. buffer overflow, or it violates a specification at s_e. A trace σ_e that contains one or more error states is called an error trace.

A trace can be also seen in logical form: the set *I* and the relation *T* can be written as their characteristic functions: $s_0 \in I$ iff $I(s_0)$ holds; $\langle s_i, s_{i+1} \rangle \in T$ iff $T(s_i, s_{i+1})$ holds. In this way, a trace σ is represented by the formula:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \tag{1}$$

In order to finds bugs (or prove their absence) up to some bounded *k* number of transitions, BMC encodes all error traces σ_e of the program *P* into a formula, and checks the satisfiability of this formula with a SAT or SMT solver. For each error trace $\sigma_e = s_0 s_1 .. s_k$, s_k needs not to be in *F*, and there is an error state $s_e \in \sigma_e$.

Since a SAT or SMT solver can only return a single model, state-of-the-art SAT-based or SMT-based Bounded Model Checker can only return a single error trace per run. The user has to fix the error, then runs the model checker again to find other error traces. On the other hand, All-SMT solver can return all models w.r.t. a set of Boolean variables, it can be exploited to find multiple counterexamples for BMC.

To illustrate our approach, we reconsider an example in Figure 2 from [17], which was used to illustrate CBMC [12], a popular BMC tool for ANSI-C. We made a small modification by adding the assertion `(y > 1)`, so that the program P contains more than one error. At the first step, the program is transformed into Static Single Assignment (SSA) form [18], variables are renamed when they are reassigned.

```
x=x+y;
if(x!=1){
  x=2;
  if(z) x++;
  assert(y > 1);
}
assert(x ≤ 3);
```

$\rightarrow$

```
x1=x0+y0;
if(x1!=1){
  x2=2;
  if(z0) x3=x2+1;
  assert(y0 > 1);
}
assert(x3 ≤ 3);
```

$$\rightarrow \begin{array}{l} \mathcal{C} := x_1 = x_0 + y_0 \wedge \\ \qquad x_2 = ((x_1 \neq 1)?2 : x_1) \wedge \\ \qquad x_3 = ((x_1 \neq 1 \wedge z_0)?x_2 + 1 : x_2) \\ \mathcal{P} := (x_1 \neq 1 \rightarrow y_0 > 1) \wedge (x_3 \leq 3) \end{array}$$

Fig. 2. Example, modified from [17]: the program is transformed into Static Single Assignment form, and then encoded into a logical formula

CBMC transforms a program P and a guard g into logical formula using two functions: $\mathcal{C}(P, g)$ transforms the program constraints, and $\mathcal{P}(P, g)$ transforms the program specification, namely assertions. Both functions are defined by induction on the syntax of program as in Figure 3 (interested readers are pointed to [17] for full details):

$$\begin{array}{l} \mathcal{C}(\text{“}\texttt{if(c)}\ I_1\ \texttt{else}\ I_2\text{”}, g) := \mathcal{C}(I_1, g \wedge \rho(c)) \wedge \mathcal{C}(I_2, g \wedge \neg\rho(c)) \\ \mathcal{P}(\text{“}\texttt{if(c)}\ I_1\ \texttt{else}\ I_2\text{”}, g) := \mathcal{P}(I_1, g \wedge \rho(c)) \wedge \mathcal{P}(I_2, g \wedge \neg\rho(c)) \\ \mathcal{C}(\text{“}I_1; I_2\text{”}, g) := \mathcal{C}(I_1, g) \wedge \mathcal{C}(I_2, g) \\ \mathcal{P}(\text{“}I_1; I_2\text{”}, g) := \mathcal{P}(I_1, g) \wedge \mathcal{P}(I_2, g) \\ \mathcal{P}(\text{“}\texttt{assert(a)}\text{”}, g) := g \rightarrow \rho(a) \\ \mathcal{C}(\text{“}\texttt{v = e}\text{”}, g) := (v_\alpha = (g?\rho(e) : v_{\alpha-1})) \end{array}$$

Fig. 3. The two functions $\mathcal{C}(\texttt{P}, g)$ and $\mathcal{P}(\texttt{P}, g)$ for program transformation. $\rho(c)$ is expression c after being renamed as per SSA form; $v_{\alpha-1}$ and v_α are value of v before and after the assignment respectively.

As shown in Figure 2, applying $\mathcal{C}$ and $\mathcal{P}$ on the SSA program results in the set of guards $(x_1 \neq 1)$ and $(z_0 \neq 0)$. We denote Boolean variable g_1 and g_2 such that $g_1 := \mathcal{BA}(x_1 \neq 1)$ and $g_2 := \mathcal{BA}(z_0 \neq 0)$.

A model of the formula $\mathcal{C} \wedge \neg\mathcal{P}$ will correspond to a trace of the program that violates the specification $\mathcal{P}$. By asking an All-SMT solver to return all models of $\varphi = \mathcal{C} \wedge \neg\mathcal{P}$ with respect to the set of Boolean abstractions of variables in the guards, i.e. $V_I = \{g_1, g_2\}$, we can get a set of all models, each one corresponds to an error trace.

Since a single trace represents a set of concrete executions, to get the inputs for just one representative concrete execution that triggers the error, we set them as the relevant variables to be included in the models, which means $V_R = \{x_0, y_0, z_0\}$. In this case hence V_I and V_R represents the guards and the inputs of the program respectively.

B. Automated Test Generation

This section shows how an All-SMT solver can be used in two different approaches for Automated Test Generation (ATG), namely Bounded Model Checking and Symbolic Execution.

1) ATG using Bounded Model Checking: We use the same trick as in the previous section. The goal is to compute all models of a formula, each one corresponds to a program trace in the program. Take an example as in Figure 4. Different from the one in Figure 2, the program contains no error, thus it will go through CBMC without any solver being called. CBMC will not generate a formula either.

```
void foo(int x, y){
  if(x > 5){
    x++;
    if (x < 3)
      x--;
    else
      y = x + 1;
  }
  return;
}
```

$$\begin{array}{ll} (g_1 = x_1 > 5) & \wedge \\ (x_2 = 1 + x_1) & \wedge \\ (g_2 = x_2 < 3) & \wedge \\ (x_3 = -1 + x_2) & \wedge \\ (x_4 = x_2) & \wedge \\ (y_2 = 1 + x_4) & \wedge \\ (x_5 = g_2?x_3 : x_4) & \wedge \\ (y_3 = g_2?y_1 : y_2) & \wedge \\ (x_6 = \neg g_1?x_1 : x_5) & \wedge \\ (y_4 = \neg g_1?y_1 : y_3) & \end{array}$$

Fig. 4. A simple program encoded into a formula

In order to generate test inputs that cover all program paths (to a given bound), we instrument the program with a fake error "`assert (0);`" at the end of the program. Since this error is reachable by all program paths, CBMC will include all the paths into the formula.

The box in the right in Figure 4 shows the formula encoded by CBMC. We run the All-SMT solver on the formula with $V_I = \{g_1, g_2\}$, $V_R = \{x_1, y_1\}$. The result is a set of solutions, each one contains value assignments for x_1 and y_1, which can be used as test inputs for the function `foo`.

2) ATG using Symbolic Execution: Symbolic Execution [19] (SE) is a programming analysis technique which executes programs on symbolic inputs instead of concrete data. For each executed program path, a path condition pc is built which represents the condition on the inputs for the execution to follow that path, according to the branching conditions in the code. The satisfiability of the path condition is checked at every branching point, using off-the-shelf solvers. In this way only feasible program paths are explored. Test generation is performed by solving the path conditions.

Here we propose another approach for Symbolic Execution using an All-SMT solver. We use SE with the *constraint solver turned off*, to compute the set of all possible program paths: $pc_1, pc_2, \ldots pc_M$. Since there is no constraint solving, a pc_i can be infeasible. Hence, the program under test can be viewed as corresponding to the following formula $pc_1 \vee pc_2 \cdots \vee pc_M$.

To illustrate, let us consider again the program in Figure 4, which can be viewed as corresponding to the following formula:

$$\begin{aligned} \varphi := \ & ((x > 5) \wedge (x + 1 < 3)) \quad \vee \\ & ((x > 5) \wedge \neg(x + 1 < 3)) \quad \vee \\ & \neg(x > 5) \end{aligned} \tag{2}$$

Notice that the path $(x > 5) \wedge (x + 1 < 3)$ is infeasible. However, it is still included in the formula, since we do not check the condition at each branching point. Applying Boolean abstraction on φ leads to:

$$\mathcal{BA} := ((C_1 = (x > 5)) \wedge (C_2 = (x + 1 < 3)))$$

Similar to the case of using BMC for ATG, we run the All-SMT solver to find all the models of the formula $\varphi \wedge \mathcal{BA}$ with $V_I = \{C_1, C_2\}$ and $V_R = \{x, y\}$. The result is a set of solutions, each one contains value assignments for x and y, which can be used as test inputs for the function `foo`.

In summary, in either the case of using Symbolic Execution or using Bounded Model Checking for ATG, V_I represents the guards of the program and V_R holds the test vector to be generated.

C. Reliability analysis

Reliability analysis [20] aims to compute the probability that a program successfully accomplishes its task without errors. Most previous work perform reliability analysis at early stages of design, on an architectural abstraction of the program, and thus they are not applicable to source code.

In [21], Filieri et al. introduced the first approach that can compute the reliability of program from Java bytecode. Their approach is to use SE to enumerate each of the symbolic paths (and its path condition pc_i). The symbolic path is then labelled as: (i)**T** if the program accomplishes the task; (ii) **F** if the program reaches an error state; (iii) **G** if we cannot decide because the path is not fully explored (G stands for grey).

From the path condition pc_i, Filieri et al. use the tool Latte [7] to compute the number of inputs $\#(pc_i)$ that satisfies the path condition pc. The reliability of the program, i.e. the probability that the program accomplishes its task, is then computed as:

$$\mathcal{R} = \frac{\Sigma\#(pc^T)}{\Sigma\#(pc^T) + \Sigma\#(pc^F) + \Sigma\#(pc^G)}$$

This section introduces an alternative implementation for the approach in [21] by using our All-SMT-based SE instead of classical SE. The improvement is that we only need to make one call to the All-SMT solver to explore all feasible paths.

```
void foo(int x, y){
  if(x > 5){
    x++;
    if (x < 3)
      x--;
    else {
      y = x + 1;
      assert false;
    }
  }
  return;
}
```

Fig. 5. A simple program

Let us consider again the previous example with only one difference: we add an error for the path $x \geq 3$. Similar to the previous section, we use SE with the constraint solver turned off to encode the program into a logical formula φ as in (2). Moreover, the two paths $((x > 5) \wedge (x+1 < 3))$ and $\neg(x > 5)$ are labelled with **T** as the program finishes normally in these two paths. On the other hand, the path $((x > 5) \wedge \neg(x+1 < 3))$ is labelled with **F** since an error is reachable in this path.

Similar to the previous section, using an All-SMT solver on $\varphi \wedge \mathcal{BA}$ with $V_I = \{C_1, C_2\}$ and $V_R = \{x, y\}$ will eliminate the infeasible path $((x > 5) \wedge (x + 1 < 3))$. We then can use the Latte tool to count the models for each paths, and compute the reliability of the program.

D. Measurement of Information Leaks

There has been active research in recent years in the area of Quantitative Information Flow (QIF) [8], [22], which aims to measure the amount of information that a system leaked to an external observer.

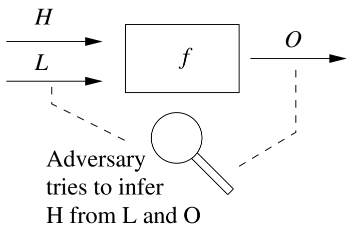

Fig. 6. Attacker model [23].

```
int check(int L, H){
  int O;
  if (H == L)
    O = ACCEPT;
  else O = REJECT;
  return O;
}
```

Fig. 7. Password Check

In the attacker model for QIF, pioneered by Denning [24], a system is represented as a probabilistic function f that maps a *high* security input H and a *low* security input L to an *observable* output O. An adversary tries to guess the high input by providing the low input and observing the output. A well-known example is the password check program in Figure 7, every time the adversary tries to guess the password with his input, the program leaks a small amount of information, and eventually leaks all information if the adversary is allowed to make enough number of attempts. The QIF problem is to "measure" the amount of information leaked by a system using information-theoretic metrics, e.g. Shannon entropy.

Let X_H , X_L and X_O be random variables representing the distribution of H, L, and O in the sample spaces Ω_H, Ω_L and Ω_O, respectively. If the adversary already knows $H = h$ then $\Omega_H = \{h\}$. If the adversary only knows that H is a 32-bit variable, then $\Omega_H = \{0, 1, .., 2^{31}\}$. Given a function $E(X)$ of uncertainty of X, leakage of information is computed as the reduction of uncertainty after observation.

$$\Delta_E(X_H) = E(X_H) - E(X_H|X_L = l, X_O)$$

where l is the low input chosen by the adversary, $E(X_H)$ is his initial uncertainty about H, and $E(X_H|X_L = l, X_O)$ is the remaining uncertainty.

A fundamental result in QIF is the theorem of channel capacity [25] [26], which states that the leakage of a program over all possible distributions is always less than or equal to the log of the number of observables of the program. In other words:

$$\Delta_E(X_H) \leq \log_2(N)$$

where N is the number of possible values of the output O. This theorem was proved for both of the cases E is Shannon entropy [25] and E is Rényi's min-entropy [26]. This result frees us from the tedious and expensive task of calculating conditional probabilities on software data. Hence, counting the number of observables N is the basis of state-of-the-art QIF methods, e.g. [27], [28], [29], [30], [31], [32], [23].

QIF as #SMT: The problem of counting the number of observables N can be casted into a model counting problem over first-order formula, defined as follows:

Definition 8 (The #SMT problem [33]): Given a theory or a combination of theories $\mathcal{T}$ and a Σ-formula φ, *Model Counting Modulo Theories* (#SMT) is the problem of counting all models M of $\mathcal{T}$ with respect to a set of Boolean variables V_I such that φ is $\mathcal{T}$-satisfiable in M.

Obviously All-SMT solver can be used for #SMT. Assuming the program P is in SSA form, and is transformed into a first-order formula φ_P in the theory of bit vector `QF_AUFBV`. This can be done automatedly using the model checker CBMC as we have described in the previous section. In SSA form, each variable is renamed when it is re-assigned. We denote by O_F the variable in φ_P that holds the final value of the output O after the execution of the program. Our setting is for integer variable of 32 bits, thus O_F is a 32-bit vector.

We instrument φ_P by adding a set of Boolean variables $V_I = \{p_0, p_1, \dots p_{31}\}$, each one tests the value of a bit of O_F. For example:

```
(assert (= (= #b1 ((_ extract 0 0) O_F)) p_0))
```

This statement in SMT-LIB v2 asserts that p_0 is equal to the value of the first bit of the bit vector O_F. Similar settings are applied for $p_1 \dots p_{31}$. By running an All-SMT solver on the instrumented φ_P with respect to the set of important variable V_I, we can count all possible values of O_F, and infer the maximum leakage of the program.

In the case of QIF, V_I represents the output bits of the program and V_R is empty.

IV. Algorithms

We build our algorithms from a number of API functions provided by the SMT solver, which we list below.

API function	Description
Assert(φ)	Assert formula φ into the solver.
Check()	Check consistency of all assertions.
Model()	Get model of the last **Check**.
Eval(`t`)	Evaluate expression `t` in current model.
Push()	Create a backtracking point.
Pop(n= 1)	Backtracks `n` backtracking points.

A key feature of SMT solvers for our algorithms is that of being *incremental* and *backtrackable*. The following example shows a sequence of API calls and their effects to the solver.

Assert(φ_1); **Check**();	φ_1	$\Rightarrow$ SAT
Push();	φ_1	
Assert(φ_2); **Check**();	$\varphi_1 \wedge \varphi_2$	$\Rightarrow$ SAT
Push();	$\varphi_1 \wedge \varphi_2$	
Assert(φ_3); **Check**();	$\varphi_1 \wedge \varphi_2 \wedge \varphi_3$	$\Rightarrow$ UNSAT
Pop(2);	φ_1	
Assert(φ_4); **Check**();	$\varphi_1 \wedge \varphi_4$	$\Rightarrow$ SAT

It is possible for an incremental SMT solver to add additional assertions to the original formula. Moreover, when **Check** is being called several times, the solver can remember its computation from one call to the other. Thus, when being called to check $\varphi_1 \wedge \varphi_2$ after checking φ_1, it avoids restarting the computation from scratch by restarting the computation from the previous state. Backtrackable means that the solver is able to undo steps, using **Push** and **Pop**, and returns to a previous state on the stack in an efficient manner.

Both z3 [9] and MathSAT [2] provide similar API functions to interact with the solver in incremental mode. Beside these functions, we develop a function `filter`(m, V_I, V_R) such that: given a model of the formula φ, a set of important Boolean variable V_I, and the set of relevant variables V_R, the function will return a subset $\mathtt{m}_{ir}$ of `m` that only contains literals from V_I and V_R. This function will be used in both algorithms.

A. Blocking clauses method

A straightforward approach for All-SMT is to add clauses that prevent the solver from finding the same solution again.

```
function ALL-BC($\varphi, V_I, V_R$) {
    N ← 0;
    $\Psi \leftarrow \epsilon$;
    Assert($\varphi$);
    while (Check() = SAT) {
        N ← N + 1;
        m ← Model($\varphi$);
        $m_{ir}$ ← filter(m, $V_I, V_R$);
        $\Psi \leftarrow \Psi \cup \{m_{ir}\}$;
        block ← FALSE;
        for all $p_i \in V_I$ {
            block ← block ∨ ($p_i \neq$ Eval($p_i$));
        }
        Assert (block);
    }
    return N, $\Psi$;
}
```

Fig. 8. Blocking clauses All-SMT

The pseudo-code for the blocking clauses method is shown in Figure 8. Every time the solver discovers a solution `m` of φ such that $\mathtt{m} = l_0 \wedge l_1 \wedge \dots \wedge l_n \wedge \dots$, in which only $l_0, l_1 \dots l_n$ are literals of $p_0, p_1 \dots p_n$ in V_I. The negation of $l_1 \wedge l_2 \wedge \dots \wedge l_n$ would be, by De Morgan's law, as follows:

$$\mathtt{block} = \neg l_0 \vee \neg l_1 \vee \dots \vee \neg l_n$$

A literal l_i in the model can be viewed as a mapping p_i to {TRUE, FALSE}, thus the negation $\neg l_i$ is $p_i \neq$ **Eval**(p_i). By adding this clause to the formula, by **Assert**(`block`), a solution with $l_0 \wedge l_1 \wedge \dots \wedge l_n$ will not be discovered again. This procedure repeats until no other solution is found. At that point, we have enumerated all the solutions of φ with respect to V_I. All solutions are stored in Ψ, and $\mathtt{N} = |\Psi|$ is the result for the corresponding #SMT problem.

The blocking clauses method is straightforward and it is simple to implement. However, adding a large number of blocking clauses will require a large amount of memory. Moreover, increasing the number of clauses also means that the *Boolean Constraint Propagation* procedure is slowed down. Despite these inefficiencies, the blocking clauses method can be used to verify the results of other techniques.

B. Depth-first search

To address the inefficiencies caused by adding a large number of clauses, we introduce an alternative method which avoids re-discovering solutions using depth-first search (DFS).

We divide the set of variables of φ into two sets: V_I is the set of important Boolean variables, and V_U is the set of unimportant, possibly non-Boolean, variables ($V_R \subseteq V_U$). Hence, with some

```
function ALL-DFS($\varphi, V_I, V_R$) {
    N ← 0;
    $\Psi \leftarrow \epsilon$;
    Assert($\varphi$);
    if (Check() ≠ SAT)  return N, $\Psi$;
    depth ← 0;
    finished ← FALSE;
    while (finished = FALSE) {
        $l$ ← choose_literal($V_I$);
        Push();
        Assert($l$);
        depth ← depth + 1;
        if (Check() = SAT)  {
            if (depth = $|V_I|$)  {
                N ← N + 1;
                m ← Model($\varphi$);
                $m_{ir}$ ← filter(m, $V_I, V_R$);
                $\Psi \leftarrow \Psi \cup \{m_{ir}\}$;
                backtrack();
            } }
        else backtrack();
    }
    return N, $\Psi$;
}
```

Fig. 9. Depth-first search All-SMT

abuse of notation the formula φ can be viewed as the function $f_\varphi(V_I, V_U)$ with codomain $\mathbb{B}$. Our All-SMT procedure is the integration of two components: the first component is a simple SAT solver to enumerate all possible partial truth assignments μ_I of V_I; the second component is the SMT solver to check the consistency of $\varphi \wedge \mu_I$.

The pseudo-code for DFS-based All-SMT is depicted in Figure 9. The method `choose_literal` chooses the next state to explore from V_I in a DFS manner, and the variable `depth` keeps the number of important variables that have been chosen. That means, `choose_literal` will select a literal $V_I[\texttt{depth}]$ or $\neg V_I[\texttt{depth}]$. This literal is then "*pushed*" to the formula as a unit clause. Recall that the plain DPLL algorithm [16] is a depth-first search combining with the BCP procedure. Here we do not perform BCP, however by adding all literals of μ_I as unit clauses to φ, we force the SMT solver to perform BCP on those literals.

When all important variables have been assigned a truth value, i.e. $\texttt{depth} = |V_I|$, and the formula in the solver is consistent, then the search has found a model. It then *backtracks* to find another one. The method `backtrack` implements a simple chronological backtracking, it "*pops*" the unit clauses and sets the variable `finished` to `TRUE` when all states are explored. It is also called when the formula in the solver is inconsistent.

Compared to the blocking clauses method, the DFS-based method is much more efficient in term of memory usage. The reason is that the blocking clauses method needs to add N blocking clauses to find all models while the DFS adds maximum $|V_I|$ of unit clauses. This memory efficiency leads to timing efficiency when there are a large number of models.

C. Implementation

We have implemented both of the algorithms discussed above in a prototype tool, called **aZ3**. The tool is built in Java, using the API functions provided by the SMT solver z3 [9]. aZ3 supports standard SMT-LIB v2 with two additional commands: the first one is `check-allsat`, also supported by MathSAT, to specify the list of important variables; and the second one is `allsat-relevant` to specify the list of relevant variables.

V. Benchmarks

The benchmarks, the aZ3 solver, and the wrapper of MathSAT for #SMT can be found at: https://github.com/qsphan/aZ3.

The first group of benchmarks are formulas in QF_LIA (integer linear arithmetic). These benchmarks are used to evaluate All-SMT solvers in the context of test input generation. They are generated using the symbolic execution tool Symbolic PathFinder [11] (SPF). SPF has a parameter, `symbolic.dp`, to set the constraint solver for it. This parameter is allowed to be set as `no_solver`, which makes the tool run without constraint solving. The architecture of SPF enables us to attach a "listener" to it. When SPF executes a program, the listener collects the path conditions, and outputs them to a QF_LIA formula. The models of the integer variables can be used as test inputs for the original programs.

The second group of benchmarks that we considered are QF_AUFBV (bit vector with array) formulas. The source of these benchmarks are case studies in the QIF literature, mostly collected in [28]. We use CBMC with the option `--smt2` to transform the programs into QF_AUFBV formulas, and then instrument the resulting formulas to make them #SMT problems. There are no relevant variables in these benchmarks.

Evaluation. Figure 10 summaries our experiments with aZ3 and MathSAT 5 on the benchmarks. In order to compare with MathSAT, we commented out the relevant variables in the QF_LIA benchmarks.

As shown in the Figure 10, MathSAT is faster than aZ3, but it is imprecise in several benchmarks (benchmarks with N in bold). This is not surprised, since we built the tool from the front-end, while the All-SMT functionality of MathSAT is built from the back-end, making use of the internal data structure. However, this back-end technique is only applicable for the DPLL($\mathcal{T}$) framework [36], and MathSAT returns incorrect models for bit vector formulas, whose solver is not DPLL($\mathcal{T}$). Especially, in the benchmark "Mix and duplicate" MathSAT is significantly faster than aZ3, but it is also extremely imprecise at the same time. Note that benchmarks in QF_AUFBV are derived from the QIF literature, and their numbers of models were already reported in other papers. For example "Mix and duplicate" was reported in [37] and [28] to have 2^{16} models.

The blocking clauses method is comparable, or even faster than the DFS-based method when the number of models is small. However, for example, for the benchmarks with 2^{16} models, adding 2^{16} blocking clauses is obviously not efficient in both time and memory. As a result, the method failed to provide the answer for such benchmarks. On the other hand, the DFS-based method was still able to provide the answer in a reasonable time.

	Benchmark	Expected N	MathSAT 5		aZ3	
			N	Time	**BC** time	**DFS** time
QF_LIA	Example in Figure 2	2	2	0.007	0.021	0.013
	Function `foo` in Figure 4	3	3	0.005	0.008	0.007
	Flap controller [21]	5	5	0.031	0.020	0.012
	Red-black tree [34]	31	31	0.016	0.054	0.073
	Bubble sort [35]	541	541	0.136	1.850	2.069
	Array false [35]	1370	1370	0.037	3.008	2.650
	Sum array false [35]	1024	1024	0.026	0.899	0.792
	Linear search false [35]	1024	1024	0.028	0.899	0.604
QF_AUFBV	Sanity check, base:`0x00001000` [28]	16	16	0.008	0.036	0.085
	Sanity check, base: `0x7ffffffa` [28]	16	16	0.009	0.040	0.087
	Implicit flow [28]	7	7	0.012	0.029	0.049
	Population count [28]	33	**71**	0.012	0.074	0.398
	Mix and duplicate [28]	65536	**162087**	4.648	-	136.947
	Masked copy [28]	65536	65536	1.319	-	18.630
	Sum query [28]	28	**64**	0.010	0.055	0.133
	Ten random outputs [28]	10	10	0.014	0.038	0.093
	CRC (8) [32]	8	**12**	0.018	0.041	0.099
	CRC (32) [32]	32	**36**	0.019	0.075	0.325

Fig. 10. N is the number of models, numbers in bold indicate incorrect results returned by MathSAT. **BC** time and **DFS** time are the time of aZ3 using the blocking clauses method and depth-first search-based method respectively. Times are in seconds. "-" means "timed out in 1 hour". Notice that for both aZ3 implementations the number of models is Expected N.

Benchmark	Leaks	**sqifc** [32]	Our new technique		
			CBMC time	**aZ3** time	Total time
Sanity check, base = `0x00001000` [28]	4	6.672	0.163	0.085	0.248
Sanity check, base = `0x7ffffffa` [28]	4	114.760	0.170	0.087	0.257
Implicit flow [28]	2.81	5.033	0.169	0.049	0.218
Population count [28]	5.04	17.278	0.162	0.398	0.560
Mix and duplicate [28]	16	-	0.154	136.947	137.101
Masked copy [28]	16	-	0.175	18.630	18.805
Sum query [28]	4.81	64.557	0.162	0.133	0.295
Ten random outputs [28]	3.32	64.202	0.160	0.093	0.253
CRC (8) [32]	3	2.551	0.184	0.099	0.283
CRC (32) [32]	5	7.755	0.193	0.325	0.518

Fig. 11. Comparing our technique with the sqifc tool in [32]. Leaks are in bits. aZ3 runs with the DFS-based algorithm. Times are in seconds, "-" means timeout in one hour.

VI. Related Work

The whole content of this paper has previously appeared as a chapter in the PhD thesis of the first author [33].

A. Multiple Counterexamples

The most relevant work to ours is that of Bhargavan et al. [38] embodied in the Verisim testing tool for network protocols. When an error trace is found to violate the specification, which is an extended LTL formula ϕ, Verisim uses a technique, called *tuning*, to replace ϕ with φ that ignores the violation. Tuning is not fully automatic.

Another technique introduced by Ball et al. [39] is embodied in the SLAM tool-kit. The algorithm uses a model checker as a sub-routine. When the model checker finds an error trace, SLAM localizes the error cause, modifying the source code with a `halt` statement at the error cause. The model checker is then invoked again, and the `halt` statements instruct the model checker to stop exploring paths at the previously found error causes. This procedure is very expensive, it requires comparing the error trace with all correct traces to localize the error, and requires to run the model checker several times. Our work is much simpler, and faster but there is a possibility that multiple error traces come from one cause.

B. Automated Test Generation

The closest to our work is FShell [40], which also uses CBMC for automated test generation. FShell transforms the program under test into a CNF formula, and solves it using an incremental SAT solver. Every time the SAT solver finds a solution representing a symbolic path, FShell adds a blocking clause to prevent that path from being explored again. As our experiments have shown, the blocking clauses method suffers from space explosion.

Traditional Symbolic Execution also uses SMT solvers to check the satisfiability of path conditions. In this approach, the SMT solver is called whenever a conditional statement is executed, hence it may be called hundreds or thousands of times. In our approach, the symbolic executor makes only one call to the All-SMT solver. This idea has been presented in a student workshop [41].

C. Reliability analysis

Our approach to reliability analysis is based on the paper of Filieri et al. [21] that uses classical Symbolic Execution and Barvinok model counting tool. We extend the approach using our new All-SMT-based Symbolic Execution instead of classical Symbolic Execution. The main difference is the same as in the case of test generation, our approach only makes one call to the All-SMT solver in the whole analysis.

D. Quantitative Information Flow

The closest to the approach in this paper is our own recent work [32], which introduced an SMT-based approach for QIF, casting the problem into #SMT. However, the definition of #SMT in [32] requires that the set of important variable V_I are Boolean abstractions of $\mathcal{T}$-atoms. Here we relax that requirement, making the definition more general.

In [32], we did not build a #SMT solver either. Suppose, for example, we want to check if $\varphi \wedge p_0$ is satisfiable, where φ is the program constraints, and p_0 corresponds to the first bit of output O. We make a driver to extract p_0 using bitwise operators of C, and make an assertion that $\neg p_0$. If CBMC returns a counterexample for this assertion, then $\varphi \wedge p_0$ is satisfiable.

As such, in [32] the source code of the driver was modified several times, and CBMC was called after each modification. Figure 11 compares the performance of our All-SMT based technique with the tool sqifc in [32]. The results show that our implementation with the All-SMT solver can significantly outperform sqifc for analysis of leakage.

E. All-Solution SAT Modulo Theories

As we have discussed throughout the paper, MathSAT is the only SMT solver that supports All-SMT. Its algorithm is briefly described in [5], and it was used to compute predicate abstraction in [3] and [4]. However, this approach is only applicable for the DPLL($\mathcal{T}$) framework [36], while bit vector formulas are solved by flattening into propositional formulas. For this reason, as shown by our experiments, MathSAT is imprecise in bit vector benchmarks.

Also in the context of predicate abstraction, Lahiri et al. [42] proposed several techniques using the SMT solver Barcelogic to generate the set of all satisfying assignments over a set of predicates. However, we are not able to include Barcelogic in our experiments, since the solver provided to us by the author does not support All-SMT. These techniques are also only applicable for the DPLL($\mathcal{T}$) framework.

A principal difference between the work mentioned above and ours is that we implemented from the front-end of an SMT solver. As a result, our approach does not depend on theory-specific implementation, and thus we can handle bit vector formulas precisely. Our approach can also be implemented even for closed-source SMT solvers that do not support All-SMT but provide the API functions used in Section IV.

VII. Conclusion and Future Work

In this paper, we have demonstrated the use of an All-SMT solver in four different domains of application: Bounded Model Checking, Automated Test Cases Generation, Quantitative Information Flow and Reliability analysis. We also introduce a lightweight technique for All-SMT using the API functions provided by an SMT solver. Future work include integrating the All-SMT solver with CBMC to localize error causes using similar idea in [39]. Models of $\varphi_1 = \mathcal{C} \wedge \mathcal{P}$ correspond to correct traces that satisfy the specification, and models of $\varphi_2 = \mathcal{C} \wedge \neg\mathcal{P}$ correspond to error traces that violate the specification. Using an All-SMT solver, we can compute the sets of all models of φ_1 and φ_2 with respect to the set of guards. Comparing the two sets of models, we can localize the transitions that only appear in error traces.

Acknowledgment

The authors are most grateful to the anonymous reviewers for their helpful, constructive comments, which improved this manuscript. Pasquale Malacaria's research was supported by grant EP/K032011/1.

References

[1] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, pp. 69–77, Sept. 2011.

[2] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT Solver," in *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, (Berlin, Heidelberg), pp. 299–303, Springer-Verlag, 2008.

[3] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar, "Computing Predicate Abstractions by Integrating BDDs and SMT Solvers," in *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '07, (Washington, DC, USA), pp. 69–76, IEEE Computer Society, 2007.

[4] A. Cimatti, J. Dubrovin, T. A. Junttila, and M. Roveri, "Structure-aware computation of predicate abstraction," in *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design*, FMCAD '09, pp. 9–16, IEEE, 2009.

[5] C. Papadopoulos, A. Cavallo, A. Cimatti, and M. Bozzano, "Installation optimisation." http://www.google.com/patents/US20130076767, Mar. 28 2013. US Patent App. 13/623,977.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, (London, UK, UK), pp. 193–207, Springer-Verlag, 1999.

[7] "LattE." http://www.math.ucdavis.edu/~latte/.

[8] D. Clark, S. Hunt, and P. Malacaria, "A static analysis for quantifying information flow in a simple imperative language," *J. Comput. Secur.*, vol. 15, pp. 321–371, Aug. 2007.

[9] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[10] C. Barrett, A. Stump, and C. Tinelli, "The smt-lib standard: Version 2.0," in *SMT Workshop*, 2010.

[11] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, (New York, NY, USA), pp. 179–180, ACM, 2010.

[12] E. Clarke, D. Kroening, and F. Lerda, " A Tool for Checking ANSI-C Programs ," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176, Springer, 2004.

[13] D. A. Plaisted and S. Greenbaum, "A Structure-preserving Clause Form Translation," *J. Symb. Comput.*, vol. 2, pp. 293–304, Sept. 1986.

[14] T. Boy de la Tour, "An Optimality Result for Clause Form Translation," *J. Symb. Comput.*, vol. 14, pp. 283–301, Oct. 1992.

[15] R. Sebastiani, "Lazy Satisfiability Modulo Theories," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, pp. 141–224, 2007.

[16] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[17] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and verilog programs using bounded model checking," in *Proceedings of the 40th annual Design Automation Conference*, DAC '03, (New York, NY, USA), pp. 368–371, ACM, 2003.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, (New York, NY, USA), pp. 25–35, ACM, 1989.

[19] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[20] R. C. Cheung, "A User-Oriented Software Reliability Model," *IEEE Trans. Softw. Eng.*, vol. 6, pp. 118–125, Mar. 1980.

[21] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 622–631, IEEE Press, 2013.

[22] P. Malacaria, "Assessing security threats of looping constructs," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, (New York, NY, USA), pp. 225–235, ACM, 2007.

[23] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. d'Amorim, "Quantifying Information Leaks Using Reliability Analysis," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, (New York, NY, USA), pp. 105–108, ACM, 2014.

[24] D. E. Robling Denning, *Cryptography and Data Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1982.

[25] P. Malacaria and H. Chen, "Lagrange multipliers and maximum information leakage in different observational models," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, (New York, NY, USA), pp. 135–146, ACM, 2008.

[26] G. Smith, "On the Foundations of Quantitative Information Flow," in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS '09, (Berlin, Heidelberg), pp. 288–302, Springer-Verlag, 2009.

[27] J. Heusser and P. Malacaria, "Quantifying information leaks in software," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, (New York, NY, USA), pp. 261–269, ACM, 2010.

[28] Z. Meng and G. Smith, "Calculating bounds on information leakage using two-bit patterns," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, (New York, NY, USA), pp. 1:1–1:12, ACM, 2011.

[29] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, "Symbolic Quantitative Information Flow," *SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, Nov. 2012.

[30] V. Klebanov, N. Manthey, and C. Muise, "SAT-Based Analysis and Quantification of Information Flow in Programs," in *Quantitative Evaluation of Systems*, vol. 8054 of *Lecture Notes in Computer Science*, pp. 177–192, Springer Berlin Heidelberg, 2013.

[31] Z. Meng and G. Smith, "Faster Two-Bit Pattern Analysis of Leakage," in *Proceedings of the 2nd International Workshop on Quantitative Aspects in Security Assurance*, QASA '13, 2013.

[32] Q.-S. Phan and P. Malacaria, "Abstract Model Counting: A Novel Approach for Quantification of Information Leaks," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, (New York, NY, USA), pp. 283–292, ACM, 2014.

[33] Q.-S. Phan, *Model Counting Modulo Theories*. PhD thesis, Queen Mary University of London, 2015.

[34] "Symbolic PathFinder's repository." http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.

[35] "Benchmarks of loops in the Software Verification competition 2014." https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp14/loops/.

[36] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *J. ACM*, vol. 53, pp. 937–977, Nov. 2006.

[37] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, (New York, NY, USA), pp. 73–85, ACM, 2009.

[38] K. Bhargavan, C. A. Gunter, I. Lee, O. Sokolsky, M. Kim, D. Obradovic, and M. Viswanathan, "Verisim: Formal Analysis of Network Simulations," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 129–145, Feb. 2002.

[39] T. Ball, M. Naik, and S. K. Rajamani, "From Symptom to Cause: Localizing Errors in Counterexample Traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, (New York, NY, USA), pp. 97–105, ACM, 2003.

[40] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement," in *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, (Berlin, Heidelberg), pp. 209–213, Springer-Verlag, 2008.

[41] Q.-S. Phan, "Symbolic Execution as DPLL Modulo Theories," in *2014 Imperial College Computing Student Workshop*, vol. 43 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 58–65, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[42] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, "SMT Techniques for Fast Predicate Abstraction," in *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, (Berlin, Heidelberg), pp. 424–437, Springer-Verlag, 2006.