Satisfiability Modulo Counting: A New Approach for Analyzing Privacy Properties

Matthew Fredrikson Somesh Jha
University of Wisconsin, Madison
{mfredrik, jha}@cs.wisc.edu

Abstract

Applications increasingly derive functionality from sensitive personal information, forcing developers who wish to preserve some notion of privacy or confidentiality to reason about partial information leakage. New definitions of privacy and confidentiality, such as differential privacy, address this by offering precise statements of acceptable disclosure that are useful in common settings. However, several recent published accounts of flawed implementations have surfaced, highlighting the need for verification techniques.

In this paper, we pose the problem of *model-counting satisfiability*, and show that a diverse set of privacy and confidentiality verification problems can be reduced to instances of it. In this problem, constraints are placed on the outcome of model-counting operations, which occur over formulas containing *parameters*. The object is to find an assignment to the parameters that satisfies the model-counting constraints, or to demonstrate unsatisfiability.

We present a logic for expressing these problems, and an abstract decision procedure for model-counting satisfiability problems fashioned after CDCL-based SMT procedures, encapsulating functionality specific to the underlying logic in which counting occurs in a small set of black-box routines similar to those required of theory solvers in SMT. We describe an implementation of this procedure for linear-integer arithmetic, as well as an effective strategy for generating lemmas. We conclude by applying our decision procedure to the verification of privacy properties over programs taken from a well-known privacy-preserving compiler, demonstrating its ability to find flaws or prove correctness sometimes in a matter of seconds.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Logics of programs; D.4.5 [*Security and Protection*]: Verification

General Terms Security, Verification, Theory

1. Introduction

The trend towards *personal data-driven* applications has changed the way in which both consumers and developers of applications think about privacy and confidentiality. Whether an application requests one's location to provide relevant suggestions for restaurants and entertainment, or permission to use one's medical records as part of a large-scale data mining operation, we are compelled to think of these properties in terms of a trade-off: *are the benefits provided by this application worth the risk of disclosing some amount of personal information?* Recent reports suggest that in many cases the risks surpass consumers' initial expectations [1, 19, 43], but this has not slowed the acceptance and widespread use of these applications, as they provide utility that cannot be readily weighed against such risks.

This has led to a set of privacy and confidentiality definitions [11, 18, 20, 35, 40, 42, 46] that give precise notions of *acceptable disclosure* or *partial information leakage*. Whereas non-interference [17] specifies the strict absence of an information channel, these definitions answer questions such as, *how likely is it that this computation will reveal my contribution to the dataset?* Developers can use these definitions to provide guarantees of privacy or confidentiality in their applications, while still deriving functionality from sensitive personal information. However, implementing programs that satisfy these definitions is notoriously difficult, as information leaks may arise from unexpected sources [26, 36]: exceptional control flow, the granularity of floating-point numbers, and even the decision not to publish a result can give an adversary enough information to infer more than intended by the privacy definition. Given the subtlety of these problems and growing need for such guarantees, the importance of verifying programs against them is immediate.

Recent work [5, 27–30] has established the connection between partial information leakage and *model counting*. This approach views leakage in terms of a relation that characterizes which inputs are *indistinguishable* on observing a program's output: two inputs are related if it is not possible to distinguish which was used to produce a given output. These relations are often expressed as logical formulas over pairs of a program's input variables, so the number of models in each equivalence class of the relation provides a measure of the degree to which privacy and confidentiality are preserved by a program. If the input space is partitioned into singleton classes, then the program leaks complete information, whereas if the relation corresponds to a single class then no information is leaked. However, because each class must be individually counted, these techniques remain limited to settings where either the relevant input class is fixed, or the number of input classes is small and can be exhaustively enumerated. Some properties, such as *differential privacy* [20], are thus not readily modeled in this way (see Section 2).

In this paper, we describe an extension to counting-based approaches that is based on the satisfiability of *parameters* in formulas with model-counting operations. Parameters appear as variables in model-counting operations that take a *fixed* interpretation when the primitive is evaluated. Thus, each such operation is effectively a function from its parameters to the natural numbers. Parameters

allow us to reason about families of related model-counting operations symbolically, opening the application of model-counting techniques to the difficult cases described above. We define a theory of model counting satisfiability that is agnostic to the logic over which counting operations are defined, and can be applied to a diverse set of privacy and confidentiality properties. To summarize, we make the following contributions:

- We introduce and formalize the problem of *model-counting satisfiability*, defining a new logic $\mathcal{L}_{\#}(\mathsf{L})$ for expressing instances of the problem. We present an abstract CDCL-based decision procedure for solving it.
- We present countersat, an instance of our model-counting satisfiability decision procedure over linear-integer arithmetic. Our procedure uses Barvinok's theory of polyhedral lattice points [7] to evaluate potential solutions and generate lemmas.
- We apply countersat to the verification of a set of programs from the Fairplay secure compiler project [8]. We are able to verify six out of our eleven benchmarks in under ten seconds.

The rest of this paper is organized as follows: section 2 gives an overview of our approach; section 3 defines our logic and its semantics; section 4 describes the abstract decision procedure, and section 5 its linear-integer instantiation; section 6 describes our evaluation; section 7 discusses related work.

2. An Illustrative Example

We begin by illustrating a reduction from the verification of differential privacy to model-counting satisfiablity. Our goal is to demonstrate how counting primitives, with parameters and additional constraints on them, can be used to encode the essential components of differential privacy; similar encodings exist for many other notions of partial information leakage, and are briefly discussed in Section 6.

Consider a simple deterministic routine addnoise, which takes an integer input x and a uniform-random integer input r, and adds pseudo-random noise derived from r to x via the function noisegen. This is a common construction in programs that attempt differential privacy, due to the popularity of the Laplace and Geometric mechanisms [20]. Differential privacy states that for any two inputs satisfying a *neighbor relation*, the ratio of the probabilities of producing a given output starting from each neighboring input does not exceed the pre-defined bound $\mathsf{exp}(\epsilon)$, where ϵ is taken to be a constant *privacy parameter*.

For the purposes of this discussion, let us assume that the neighbor relation for our program noisegen holds for all pairs of inputs $\mathtt{x}_1$, $\mathtt{x}_2$ whose absolute difference is less than some constant S. We are now in a position to state differential privacy as it applies to noisegen: *for all* $\mathtt{x}_1, \mathtt{x}_2, \mathtt{s}$*, and any* $\mathtt{r}_1, \mathtt{r}_2$ *drawn uniform-randomly*,

$$-S < \mathtt{x}_1 - \mathtt{x}_2 < S \rightarrow \frac{\Pr[\mathsf{addnoise}(\mathtt{x}_1, \mathtt{r}_1) = \mathtt{s}]}{\Pr[\mathsf{addnoise}(\mathtt{x}_2, \mathtt{r}_2) = \mathtt{s}]} \leq \mathsf{exp}(\epsilon)$$

Recall that we assume addnoise is deterministic, so the probabilities are defined over the random choice of inputs $\mathtt{r}_1$ and $\mathtt{r}_2$.

Because we draw $\mathtt{r}_1$ and $\mathtt{r}_2$ uniform-randomly, if we assume that each variable has a finite domain, then we can equate each of these probabilities with the ratio of assignments satisfying the given condition:

$$\Pr[\mathsf{addnoise}(\mathtt{x}_1, \mathtt{r}_1) = \mathtt{s}] = \frac{\mathsf{count}(\mathtt{r}_1, \mathsf{addnoise}(\mathtt{x}_1, \mathtt{r}_1) = \mathtt{s})}{\textit{Size of } \mathtt{r}_1\textit{'s domain}}$$

Here we assume that $\mathtt{x}_1$ and s are given a fixed interpretation, so the only free variable in the condition is $\mathtt{r}_1$, which is the variable whose satisfying assignments are counted. In words, we can say that the probability of the event $\mathsf{addnoise}(\mathtt{x}_1, \mathtt{r}_1) = \mathtt{s}$, for a given choice of $\mathtt{x}_1$ and s, is equivalent to the fraction of values of $\mathtt{r}_1$ that cause addnoise to return s when given $\mathtt{x}_1$ and $\mathtt{r}_1$. Again, this follows from our assumption that $\mathtt{r}_1$ is chosen uniform-randomly from its domain. This gives us a way of expressing differential privacy in terms of a formula characterizing the number of models of the input/output relation of our computation.

In the interest of making the example more concrete, suppose we instantiate noisegen with a trivial function that takes the uniform-random input r, and converts it to a uniform-random integer in the range $(-B, B)$, for some constant B. While this function can indeed provide differential privacy for large enough B, it does not provide much utility; we use it here for illustrative purposes only, as a function that encodes a useful differentially-private noise generator would be so large as to distract from the main point of this discussion. This gives us a simple input/output semantics for addnoise in terms of the relation $\Phi(\mathtt{x}, \mathtt{r}, \mathtt{s})$: $\Phi(\mathtt{x}, \mathtt{r}, \mathtt{s}) \equiv (\mathtt{s} = \mathtt{x} + \mathtt{r} \wedge -B < \mathtt{r} < B)$. We can now write a verification condition for noisegen in terms of model-counting operations:

$$-S < \mathtt{x}_1 - \mathtt{x}_2 < S \wedge \frac{\mathsf{count}(\mathtt{r}_1, \Phi(\mathtt{x}_1, \mathtt{r}_1, \mathtt{s}))}{\mathsf{count}(\mathtt{r}_2, \Phi(\mathtt{x}_2, \mathtt{r}_2, \mathtt{s}))} > \mathsf{exp}(\epsilon)$$

Note that we have negated the original formula to encode a satisfiability problem whose models correspond to counterexamples, so that we can either prove or disprove the differential privacy of addnoise by reasoning about the assignments to $\mathtt{x}_1, \mathtt{x}_2$, and s. For example, if we choose $S = 10, B = 5, \epsilon = 1$, then addnoise does not satisfy the property; under these constants, we see that $\mathtt{x}_1 = 1$, $\mathtt{x}_2 = 6$, $\mathtt{s} = 1$ implies that there is a single solution to $\mathtt{r}_1$ and no solutions to $\mathtt{r}_2$.

A few characteristics of this example illustrate the need for a notion of model-counting satisfiability. First, notice that we are not pre-supposing any constant bound on the result of the counting operations, so a propositional encoding of counting that asserts the existence of a set number of distinct values for $\mathtt{r}_1$ and $\mathtt{r}_2$ that satisfy the remaining constraints, will not suffice here; we would need to enumerate all possible ratios of counting solutions in a monolithic disjunction, which would become immediately intractible. Second, recalling that $\mathtt{x}_1, \mathtt{x}_2$, and s are assumed to take a fixed interpretation in each counting operation, we have actually encoded a *collection* of counting problems in this formula: one for each valuation of these variables. It is clear that the approach of expanding this formula into a large disjunction of distinct counting operations, one for each valuation of the above variables, and applying model counting as it has been used in previous work [5, 27, 29], is also intractible. Finally, while existing parametric model counting techniques [44] help address many of these problems by returning analytic expressions that characterize model counts in terms of parameters, these expressions alone do not allow us to answer the question needed to verify the example: do there exist values for $\mathtt{x}_1, \mathtt{x}_2$, and s that violate differential privacy? *A technique for answering this type of question is the focus of this paper.*

In this paper, we describe the problem of model-counting satisfiability, and a logic $\mathcal{L}_{\#}(\mathsf{L})$ for expressing its instances. $\mathcal{L}_{\#}(\mathsf{L})$ allows us to treat $\mathtt{x}_1, \mathtt{x}_2$, and s as *counting parameters* whose values are constrained by both the neighbor and input/output relations in the reduction shown in our example. Each parameter is free in the formula outside of any counting operation, but assumes a fixed value when interpreting the counting operations that mention it. We then describe a decision procedure for this problem, which uses Barvinok's theory of polyhedral lattice points [7] to prove or disprove the existence of parameter values that satisfy the verification condition, without resorting to the intractible case-splitting procedures described above.

3. A Logic for Model-Counting Satisfiability

In this section, we define a logic $\mathcal{L}_{\#}(\mathsf{L})$ to embody our theory of model counting satisfiability. $\mathcal{L}_{\#}(\mathsf{L})$ is parameterized by a *base logic* L that contains the formulas whose models are counted, i.e., the targets of counting operations. To define $\mathcal{L}_{\#}(\mathsf{L})$ for a particular L, we assume two fundamental primitives: a *satisfiablity oracle* $\mathcal{O}_{\mathsf{sat}}$, and a *model counting oracle* $\mathcal{O}_{\mathsf{cnt}}$, both for the logic L. The satisfiability oracle $\mathcal{O}_{\mathsf{sat}}(\phi)$ is a function from a formula $\phi \in \mathsf{L}$ to $\{\mathsf{true}, \mathsf{false}\}$ that returns true iff ϕ is satisfiable in L. The model counting oracle $\mathcal{O}_{\mathsf{cnt}}(V, \phi)$ is a function from a formula $\phi \in \mathsf{L}$ and a set of variables V appearing in ϕ, and returns the number of distinct models of V that satisfy ϕ, i.e., such that $\mathcal{O}_{\mathsf{sat}}(\phi[v_1 \mapsto \mathtt{v}_1, \ldots, v_n \mapsto \mathtt{v}_n])$ returns true.

A key aspect of $\mathcal{L}_{\#}(\mathsf{L})$ is support for *counting parameters*, or variables that appear in a counted base logic formula and have two additional properties: *(1)* they are free in the formula outside of any counting terms, and *(2)* when interpreting a counting term, they are assumed to take a specific value. We often call counting parameters *shared variables* to reflect the fact that they are shared between counting operations over L and constraints in $\mathcal{L}_{\#}(\mathsf{L})$. For example, if we assume that the variable $y^{\mathtt{s}}$ is a parameter (or *shared*, denoted by the superscript $\mathtt{s}$) in the linear-integer formula $0 \leq y^{\mathtt{s}} \leq 5 \wedge \mathsf{count}(x, 1 \leq x \leq y^{\mathtt{s}}) \leq 2$, then the formula describes all values for the parameter $y^{\mathtt{s}}$ that lie in the interval $[0, 5]$ and result in at most two solutions for the variable x when $1 \leq x \leq y^{\mathtt{s}}$. The meaning of a $\mathcal{L}_{\#}(\mathsf{L})$ formula is defined by the set of models taken by counting parameters. *The presence of parameters in a $\mathcal{L}_{\#}(\mathsf{L})$ formula prevents counting terms from being reduced to constants by $\mathcal{O}_{\mathsf{cnt}}$, and is the key challenge in reasoning about $\mathcal{L}_{\#}(\mathsf{L})$.*

In what follows, we use V to refer to a set of variables, and $V(\phi)$ to the set of variables in a given formula ϕ. We use the symbol α to define an interpretation of the variables in a formula, i.e., a mapping from $V(\phi)$ to constants. Abusing notation, we will often apply a mapping α to a formula, with the understanding that each variable in the domain of α is replaced with its image in the resulting formula. Given a formula ϕ, we indicate that it has free variables exclusively in the set V by writing $\phi(V)$.

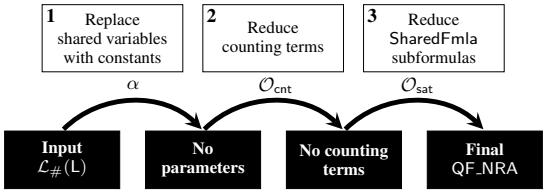
Syntax. Figure 1 shows the syntax for our logic. To avoid confusion between the parts of $\mathcal{L}_{\#}(\mathsf{L})$ that refer to counting and those that are targets of counting statements (i.e., the formulas whose models are counted), we partition the syntax into three parts: the *base component*, *counting component*, and *outer component*. The base component contains only formulas that are the target of counting operations, and resides exclusively in L. The counting component contains functions and predicates for reasoning about the number of satisfying solutions to formulas in L: a count term for counting operations, arithmetic over counting operations and reals, as well as the usual relational operators $(\leq, <, =)$ between numeric terms. The outer component contains formulas from L that specify constraints on counting parameters, as well as the usual propositional connectives. Similarly, we separate the variables appearing in a formula from $\mathcal{L}_{\#}(\mathsf{L})$ into two groups: *base* (V_{base}) and *shared* (V_{share}). Base variables appear only in base logic subformulas, whereas shared variables appear in both counting operations and formulas from L in the outer component (i.e., instances of $\mathsf{SharedFmla}$). Unless it is clear from the context, shared variables are given the superscript $\mathtt{s}$, e.g. $x^{\mathtt{s}}$. Note that base logic formulas used in counting operations can contain variables that are neither shared nor counted; our definition of $\mathcal{O}_{\mathsf{cnt}}$ treats these as "don't care" variables that are existentially quantified out of the base logic formula when such a counting operation is interpreted. For example, $\mathcal{O}_{\mathsf{cnt}}$ gives $\mathsf{count}(x, 0 \leq x, y \leq 3 \wedge x + y = 5)$ the same interpretation as $\mathsf{count}(x, \exists y : 0 \leq x, y \leq 3 \wedge x + y = 5)$.

Base	BaseFmla	::= $\phi(V_{\mathsf{share}}, V_{\mathsf{base}}) \in \mathsf{L}$
Counting	Const	::= c from $\mathbb{R}$
	Term	::= Const | Term $\{+, \times\}$ Term
		| $\mathsf{count}(v_0, \ldots, v_n, \mathsf{BaseFmla})$
		where $v_0, \ldots, v_n \subseteq V_{\mathsf{base}}$ in BaseFmla
	Atom	::= Term $\{\leq, <, =\}$ Term
Outer	SharedFmla	::= $\phi(V_{\mathsf{share}}) \in \mathsf{L}$
	Fmla	::= Atom | (SharedFmla) | (Fmla) | $\neg$Fmla
		| Fmla $\wedge$ Fmla

Figure 1. $\mathcal{L}_{\#}(\mathsf{L})$ grammar.

$$
\begin{aligned}
[\![\neg e]\!](\alpha) &\rightarrow \neg [\![e]\!](\alpha) \\
[\![e_1 \odot e_2]\!](\alpha) &\rightarrow [\![e_1]\!](\alpha) \odot [\![e_2]\!](\alpha) \\
[\![\phi^{\mathsf{SharedFmla}}]\!](\alpha) &\rightarrow \mathcal{O}_{\mathsf{sat}}(\alpha(\phi^{\mathsf{SharedFmla}})) \\
[\![\mathsf{count}(v_1, \ldots, v_n, \phi)]\!](\alpha) &\rightarrow \mathcal{O}_{\mathsf{cnt}}(\{v_1, \ldots, v_n\}, \alpha(\phi))
\end{aligned}
$$

(a) Semantic denotation operator $[\![\cdot]\!](\alpha)$.



(b) Denotational transformation to QF_NRA.

Figure 2. Semantics of $\mathcal{L}_{\#}(\mathsf{L})$.

Semantics. We define the semantics by giving a denotational translation $[\![\cdot]\!](\alpha)$ from $\mathcal{L}_{\#}(\mathsf{L})$ formulas and variable mappings to sentences in the quantifier-free fragment of non-linear arithmetic over the reals (QF_NRA). Because the semantics of QF_NRA is well-understood, we can define the satisfiability of a formula ϕ in $\mathcal{L}_{\#}(\mathsf{L})$ in terms of the existence of a variable mapping α that results in a satisfiable QF_NRA formula $[\![\phi]\!](\alpha)$.

In order to ensure compatibility between our denotation operator and QF_NRA, which does not encode ∞, we define semantics only for formulas whose count terms refer to base logic formulas with a finite set of models, called *finite-base* formulas (Definition 1). A formula is finite-base iff each count term is given a finite interpretation by $\mathcal{O}_{\mathsf{cnt}}$ for any assignment to its parameters.

Definition 1. (*Finite-base formula*). A formula $\phi_{\mathsf{L}} \in \mathcal{L}_{\#}(\mathsf{L})$ is *finite-base* iff for each instance of

$$\mathsf{count}(\{x_1, \ldots, x_n\}, \phi_{\mathsf{L}}(x_1, \ldots, x_n, y_1^{\mathtt{s}}, \ldots, y_m^{\mathtt{s}}))$$

appearing in ϕ, for all assignments α to the parameters $y_1^{\mathtt{s}}, \ldots, y_m^{\mathtt{s}}$, $\mathcal{O}_{\mathsf{cnt}}(\{x_1, \ldots, x_n\}, \alpha(\phi_{\mathsf{L}}))$ is finite.

The denotational translation $[\![\cdot]\!](\alpha)$ on which our semantics is based works in three phases as shown in Figure 2(b). When given a total mapping α from the shared variables in ϕ to constants from L, all variables in $V_{\mathsf{share}}(\phi)$ are first replaced by their image under α. The $\mathsf{count}(\{x_1, \ldots, x_n\}, \phi)$ terms are then replaced with constants from QF_NRA, using the oracle $\mathcal{O}_{\mathsf{cnt}}(\{x_1, \ldots, x_n\}, \phi)$. Lastly, each $\mathsf{SharedFmla}$ subformula is reduced to a Boolean constant using $\mathcal{O}_{\mathsf{sat}}$. The final step is always possible because

SharedFmla subformulas contain free variables only from V_{share}, which were replaced with constants in the first step. The result is a QF_NRA sentence ϕ' that we define to be equivalent to $\alpha(\phi)$.

A full definition of the denotational tranformation operator $[\![\cdot]\!]$ is given by the set of reduction rules shown in Figure 2(a). We use $[\![\cdot]\!](\alpha)$ to define the satisfiability of a $\mathcal{L}_{\#}(\mathsf{L})$-formula, given in Definition 2.

Definition 2. *(Satisfiability of formulas in $\mathcal{L}_{\#}(\mathsf{L})$).* A finite-base $\mathcal{L}_{\#}(\mathsf{L})$ formula ϕ is satisfiable in $\mathcal{L}_{\#}(\mathsf{L})$ iff there exists a mapping α such that $[\![\phi]\!](\alpha)$ is valid in QF_NRA.

In general, when we place no restrictions on L, $\mathcal{L}_{\#}(\mathsf{L})$ is not decidable (Theorem 1). However, $\mathcal{L}_{\#}(\mathsf{L})$ is decidable in several important cases, as discussed in Sections 4 and 5.

Theorem 1. $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ is undecidable.

4. An Abstract Decision Procedure for $\mathcal{L}_{\#}(\mathsf{L})$

In this section, we give an abstract decision procedure called $\mathsf{sat}^{\#}$ for satisfiability of formulas in $\mathcal{L}_{\#}(\mathsf{L})$. We call the procedure abstract because we do not make specific assumptions about L, instead giving a small set of general requirements in the form of three *black-box* interface functions needed to make the decision procedure work: genparam (Definition 3), explain (Definition 4), and reduce (Definition 5).

$\mathsf{sat}^{\#}$ takes after CDCL satisfiability and SMT solvers [38], and borrows concepts from the model-constructing satisfiability calculus of de Moura and Jovanovíc [37]. Problems are posed as sets of CNF clauses, and facts about a potential solution are iteratively decided. Unit propagation or counting-theory specific reasoning are used throughout for inference, continuing until either all clauses are satisfied, or a conflict arises. On encountering a conflict, the procedure backtracks to a previous decision point and learns a lemma to avoid similar conflicts in the future, either using resolution or counting-theory lemmas. If the procedure determines that the clause set is not satisfied in the current context, and there are no decisions on which to backtrack, it terminates with unsat.

Why is $\mathsf{sat}^{\#}$ different from other CDCL solvers? The procedure differs from traditional CDCL SMT procedures in a few key aspects:

1) It does not assume the ability to decide the satisfiability of an arbitrary conjunction of $\mathcal{L}_{\#}(\mathsf{L})$ literals. The main primitive, $\mathcal{O}_{\mathsf{cnt}}$, counts formulas without parameters, leaving no clear way of deciding conjunctions with count terms. As such, satisfiability and conflict detection on partial solutions almost always requires a partial model assignment. To cope with this, $\mathsf{sat}^{\#}$ incrementally constructs models, and aggressively learns counting theory lemmas for hypothetical assignments.

2) It allows lemmas containing base logic literals not appearing in the original problem statement. This is a *necessary* condition for $\mathsf{sat}^{\#}$ in light of the previous condition.

3) It can generate base logic literals not appearing in the original problem statement in the form of non-interactive *advice* to the model generator. This is not needed for correctness, but for efficiency it helps to incorporate specialized counting-theory reasoning that optimistically narrows the model space, but does not necessarily entail a valid counting theory deduction.

In this section, we describe a set of transition rules that addresses each of these issues.

4.1 Abstract Procedure

We describe $\mathsf{sat}^{\#}$ as a transition system in the style of de Moura and Jovanović [37] and Nieuwenhuis *et al.* [38]. Following [37], the states in the transition system are of the form $\langle M, C\rangle$, where M is a sequence of *trail elements* and C is a set of clauses. M is referred to as the *trail*, and reflects a series of decisions and inferences. Trail elements consist of *decided literals* (appearing simply as l in the trail, for a given literal l), *model assignments* (appearing as $x \mapsto \mathtt{v}$), and *propagated literals* (appearing as $c \to l$, for a clause c and literal l). Decided literals and model assignments are facts assumed to be true at a given point in a trail, whereas propagated literals are the result of *inference* carried out in a given context. We write M_{L} to denote the set of clauses consisting of the base-logic literals decided or implied on M.

The set of model assignments on a trail M induce an interpretation α_M of shared variables, i.e., if $x \mapsto \mathtt{v}$ appears on M, then $\alpha_M(x) = \mathtt{v}$. Similarly, a trail M induces an interpretation of literals appearing in the clause set C. Following again de Moura and Jovanović [37], we define two functions $\mathsf{value}_b(l, M)$ and $\mathsf{value}_c(l, M)$. $\mathsf{value}_b(l, M)$ corresponds to the value of l given the decided literals (i.e., the *Boolean interpretation*) on the trail M, whereas $\mathsf{value}_c(l, M)$ corresponds to the value of l given the model assignments on M and the resulting interpretation of l under the semantics of L. The semantics of these functions is straightforward: $\mathsf{value}_b(l, M) = \mathsf{true}$ (resp. false) if it is decided or implied true (resp false) on the trail M, and $\mathsf{value}_c(l, M) = \mathsf{true}$ (resp. false) if the semantics of the literal l (Figure 2(a)) reduce to true (resp. false) under the interpretation α_M. Both are undefined (undef) otherwise. The full definition of these functions is given in Figure 3(c).

M is *consistent* if $\mathsf{value}_c(l, M) \neq \mathsf{false}$ whenever $\mathsf{value}_b(l, M) = \mathsf{true}$. For consistent trails, we define a unified version of value, given in Figure 3(c). We extend this function to clauses in the natural way, so $\mathsf{value}(c, M) = \mathsf{true}$ if at least one literal in c evaluates to true, false if all literals in c evaluate to false, and undef otherwise. If $\mathsf{value}(c, M) = \mathsf{true}$, then c is *satisfied* by M, denoted $\mathsf{satisfied}(c, M)$. We extend the definition of satisfied to a set of clauses C: $\mathsf{satisfied}(C, M) = \mathsf{true}$ if $\mathsf{value}_c(l, M) = \mathsf{value}_b(l, M)$ for all literals on M, and $\mathsf{satisfied}(c, M) = \mathsf{true}$ for each clause $c \in C$. If there exists a clause $c \in C$ such that $\mathsf{value}(c, M) = \mathsf{false}$, or a set of literals $l_1, \ldots, l_n$ asserted on M such that $\neg\mathcal{O}_{\mathsf{sat}}(\alpha_M(l_1) \wedge \cdots \wedge \alpha_M(l_n))$, then we say that C is *in conflict* under M, denoted $\mathsf{conflict}(C, M)$.

We are now in a position to define the three black-box functions required by $\mathsf{sat}^{\#}$: genparam (Definition 3), explain (Definition 4), and reduce (Definition 5). Each of these functions assumes computable versions of the oracles $\mathcal{O}_{\mathsf{cnt}}$ and $\mathcal{O}_{\mathsf{sat}}$ introduced in Section 3. genparam builds partial assignments for $\mathcal{L}_{\#}(\mathsf{L})$ models that satisfy a given set of clauses in the base logic. Its role is to guide $\mathsf{sat}^{\#}$ towards satisfying solutions, or solutions that quickly lead to conflicts, using the base logic facts and counting lemmas available in the context.

Definition 3. $\mathsf{genparam}(C, x)$. Given a set C of clauses from L, $C = \{c_1, \ldots, c_n\}$, and a variable $x \in V_{\mathsf{share}}$, $\mathsf{genparam}(C, x)$ returns a value v from the constants in L such that $\mathcal{O}_{\mathsf{sat}}(c_1 \wedge \cdots \wedge c_n \wedge x = \mathtt{v})$ is true.

explain produces counting theory lemmas from a trail and a literal implied by the trail. Clauses returned from explain must always be valid counting theory deductions, following from $\mathcal{O}_{\mathsf{cnt}}$ and the base-logic semantics.

Definition 4. $\mathsf{explain}(l, M)$. Given a consistent trail M and a literal l implied by M, $\mathsf{explain}(l, M)$ returns a $\mathcal{L}_{\#}(\mathsf{L})$ clause $c = l_1 \vee \cdots \vee l_n \vee l'$ such that $\mathsf{value}(l_i, M) = \mathsf{false}$ for all i and $l' \to l$.

Finally, $\mathsf{reduce}(M, l)$ is a bridge between the oracles and $\mathsf{sat}^{\#}$. Intuitively, it applies $\mathcal{O}_{\mathsf{cnt}}$ to each count term in its input literal, or $\mathcal{O}_{\mathsf{sat}}$ whenever l is a base-logic literal. When the model assignments on the trail are total over the shared variables in l, this amounts to

DECIDE		
$\langle M, C\rangle \mapsto \langle [M, l'], C\rangle$	**if**	l appears in C $l' = l$ or $l' = \neg l$ $\mathsf{value}(l, M) = \mathsf{undef}$
PROPAGATE		
$\langle M, C\rangle \mapsto \langle [M, c' \to l], C\rangle$	**if**	$c' = (l_1 \vee \ldots \vee l_n \vee l) \in C$ $\mathsf{value}(l_i, M) = \mathsf{false}$ $\mathsf{value}(l, M) = \mathsf{undef}$
SAT		
$\langle M, C\rangle \mapsto \mathsf{sat}$	**if**	$\mathsf{satisfied}(C, M)$
UNSAT		
$\langle M, C\rangle \mapsto \mathsf{unsat}$	**if**	$\mathsf{conflict}(C, M)$ $\mathsf{explain}(\mathsf{false}, M) = \mathsf{false}$ M has no decided literals
LEARN		
$\langle M, C\rangle \mapsto \langle M, C \cup \{c\}\rangle$	**if**	$\mathsf{conflict}(C, M)$ $C \models c, c = l_1 \vee \cdots \vee l_n$ $l_1, \ldots, l_n$ appear in C $\mathsf{value}(c, M) = \mathsf{false}$
FORGET		
$\langle M, C\rangle \mapsto \langle M, C \setminus \{c\}\rangle$	**if**	$c \in C$ and c is learned $\neg\mathsf{conflict}(C, M)$
BACKJUMP		
$\langle [M, N], C\rangle \mapsto \langle [M, c \to l], C\rangle$	**if**	$\mathsf{conflict}(C, [M, N])$ $C \models c, c = l_1 \vee \cdots \vee l_n \vee l$ $l_1, \ldots, l_n$ appear in C $\mathsf{value}(l_i, M) = \mathsf{false}$ $\mathsf{value}(l, M) = \mathsf{undef}$ N starts with a decision

(a) Traditional clausal rules.

C-DECIDE		
$\langle M, C\rangle \mapsto \langle [M, x \mapsto \mathtt{v}], C\rangle$	**if**	$x \in V_{\mathsf{share}}(C)$ not decided in M $C' = \{c_1, \ldots, c_n\}$ $c_i, \ldots, c_n \in \mathsf{L}(V_{\mathsf{share}})$ $C' \cap M_{\mathsf{L}} \cap C$ is maximal $\mathtt{v} = \mathsf{genparam}(C', x)$ $\mathsf{consistent}([M, x \mapsto \mathtt{v}])$
C-PROPAGATE		
$\langle M, C\rangle \mapsto \langle [M, c \to l], C\rangle$	**if**	$l \in C, \mathsf{value}_b(l, M) = \mathsf{undef}$ $\mathsf{conflict}(C, [M, \neg l])$ $c = \mathsf{explain}(l, M)$
C-MODEL-DECIDE		
$\langle M, C\rangle \mapsto \langle [M, l], C\rangle$	**if**	x is not decided in M $\neg\mathsf{consistent}([M, x \mapsto \mathtt{v}])$ $c = \mathsf{explain}(\mathsf{false}, [M, x = \mathtt{v}])$ $c = l_1 \vee \cdots \vee l_n \vee l$ $\mathsf{value}(l, M) = \mathsf{undef}$
C-LEARN		
$\langle M, C\rangle \mapsto \langle M, C \cup \{c\}\rangle$	**if**	$\mathsf{conflict}(C, M)$ $c = \mathsf{explain}(\mathsf{false}, M)$
C-BACKJUMP		
$\langle [M, N], C\rangle \mapsto \langle [M, c \to l], C\rangle$	**if**	$\mathsf{conflict}(C, [M, N])$ $\mathsf{explain}(\mathsf{false}, [M, N]) = c$ $c = l_1 \vee \cdots \vee l_n \vee l$ $\mathsf{value}(l_i, M) = \mathsf{false}$ $\mathsf{value}(l, M) = \mathsf{undef}$
C-BACKJUMP-DECIDE		
$\langle [M, x \mapsto \mathtt{v}, N], C\rangle \mapsto \langle [M, l], C\rangle$	**if**	$\mathsf{conflict}(C, [M, x \mapsto \mathtt{v}, N])$ $\mathsf{explain}(\mathsf{false}, [M, x \mapsto \mathtt{v}, N]) = c$ $c = l_1 \vee \cdots \vee l_n \vee l$ $\exists l_i.\mathsf{value}(l_i, M) = \mathsf{undef}$ $\mathsf{value}(l, M) = \mathsf{undef}$

(b) Counting theory rules.

$$\mathsf{value}_b(l, M) = \begin{cases} \mathsf{true}, & l \text{ or } c \to l \in M \\ \mathsf{false}, & \neg l \text{ or } c \to \neg l \in M \\ \mathsf{undef}, & \text{otherwise} \end{cases} \quad \mathsf{value}_c(l, M) = \begin{cases} \mathsf{true}, & \mathsf{reduce}(M, l) = \mathsf{true} \\ \mathsf{false}, & \mathsf{reduce}(M, l) = \mathsf{false} \\ \mathsf{undef}, & \text{otherwise} \end{cases}$$

$$\mathsf{value}(l, M) = \begin{cases} \mathsf{value}_b(l, M), & \mathsf{value}_b(l, M) \neq \mathsf{undef} \\ \mathsf{value}_c(l, M), & \text{otherwise} \end{cases}$$

(c) Auxiliary functions

Figure 3. Transition rules for $\mathsf{sat}^{\#}$.

simple evaluation of terms. When the assignments are not total, reduce uses $\mathcal{O}_{\mathsf{sat}}$ to determine the *feasability* of base-logic literal l in the current trail: l is infeasible if it cannot be satisfied in the current trail, so reduce assigns it the value false. If l is feasible, it is given the value undef, as future decisions might make it unsatisfiable.

Definition 5. $\mathsf{reduce}(M, l)$. Given a trail M and a literal l, $\mathsf{reduce}(M, l)$ returns a new literal l' derived from l by applying two changes:

1. If l contains a $\mathsf{count}(\{v_1, \ldots, v_n\}, \phi)$ term such that α_M is total over $V_{\mathsf{share}}(\phi)$, reduce replaces it with $\mathcal{O}_{\mathsf{cnt}}(\{v_1, \ldots, v_n\}, \alpha_M(\phi))$ and simplifies any resulting QF_NRA sentences to Boolean constants.
2. If $l \in \mathsf{L}$ and α_M is total over $V_{\mathsf{share}}(l)$, reduce replaces l with $\mathcal{O}_{\mathsf{sat}}(\alpha_M(l))$. If α_M is not total over $V_{\mathsf{share}}(l)$, reduce replaces it with false when $\mathcal{O}_{\mathsf{sat}}(\alpha_M(l) \wedge M_{\mathsf{L}})$ is false, and undef otherwise.

Transition Rules. Given a set of clauses C_0, $\mathsf{sat}^{\#}$ begins in the state $\langle [], C_0\rangle$. Rules from Figures 3(a) and 3(b) are applied with the goal of entering either the sat or unsat state. Figure 3(a) shows the clausal search and conflict rules, which are similar to those used by traditional CDCL solvers, and are taken with minor modifications from [37].

The DECIDE rule assigns a Boolean value to a literal l from the clause set C, as long as l does not already have a value assigned to it in the current trail. PROPAGATE performs unit propagation. SAT replaces the current state with the terminal state sat whenever the trail M implies that each clause in C is satisfied. UNSAT enters the terminal state unsat whenever the trail implies that a clause in C is false, and there are no further decisions in the trail on which to backtrack. LEARN supports clausal lemma learning when a conflict exists in the trail: if the clause c is a valid deduction containing only literals appearing in the current clause set, and it is false in the current context, it can be added to the clause set. FORGET allows the procedure to remove a previously-learned clause from the set, as long as the trail is not in conflict. Lastly, BACKJUMP supports backtracking whenever a conflict arises. After backtracking, the new trail is a prefix of the current one, with a new inference $c \to l$ appended.

Figure 3(b) shows the rules specific to $\mathcal{L}_{\#}(\mathsf{L})$. C-DECIDE uses genparam to assign a value to a shared variable that does not already contain an assignment in the current context. The assignment must be consistent with the trail, in order to ensure that value remains well-defined. The clause set C' passed to genparam can con-

tain literals from outside of C; they can be arbitrary clauses from L, although C' must always contain all base logic literals on the trail, as well as any pure L clauses from the original set. This allows $\mathsf{sat}^\#$ to pass information to genparam that might yield correct values with higher probability, while simultaneously forcing it to communicate all relevant constraints in the current context. For example, given the trail $[\mathsf{count}(\{x, y\}, 0 \leq x, y \leq y^{\mathsf{s}}) = 121]$, $\mathsf{sat}^\#$ might pass the clause set $C' = \{6 \leq p \leq 12\}$ to genparam. There are no base logic literals asserted on the trail and no additional constraints on y^{s}, so genparam might generate many incorrect values leading to a large number of lemmas, before finding the correct value. To avoid this, $\mathsf{sat}^\#$ passed the advice $6 \leq p \leq 12$, which drastically narrows the solution space ($y^{\mathsf{s}} = 10$ is a satisfying value). $\mathsf{sat}^\#$ can obtain this sort of information through theory-specific mechanisms or approximation methods; an example is given in Section 5.

C-PROPAGATE supports propagation specific to the semantics of $\mathcal{L}_\#(\mathsf{L})$. C-MODEL-DECIDE adds a derived assumption to the trail that explains the infeasability of a hypothetical model assignment. This is sometimes necessary to ensure progress from C-DECIDE. Because C-DECIDE prevents assignments that result in conflicting states, it may not be able to produce an assignment to a variable if the trail is infeasible and there are no opportunities for backtracking. For example, if we modify the previous example slightly and assume the trail $[\rightarrow \mathsf{count}(\{x, y\}, 0 \leq x, y \leq y^{\mathsf{s}}) = 120]$, where the literal has been propagated. The procedure cannot continue unless explain concludes that y^{s} is unsatisfiable. This may not be possible because $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$ is undecidable, but if it can use a hypothetical assignment $y^{\mathsf{s}} \mapsto 9$ to give a lemma that implies $y^{\mathsf{s}} > 9$, and subsequently produce a similar assignment-driven lemma implying $y^{\mathsf{s}} < 10$, then it can stop in unsat. This is how our explain for QF_LIA works, as we show in Section 5. C-MODEL-DECIDE incorporates these facts when they are needed.

The counting theory-specific learning rule, C-LEARN, behaves like clausal LEARN, but instead using explain to produce a lemma. There are two backtracking rules specific to the counting theory. The first, C-BACKJUMP, is similar to the clausal BACKJUMP but uses counting theory-specific reasoning to explain the conflict. Finally, C-BACKJUMP-DECIDE allows the procedure to recover from a conflict involving a model assignment $x \mapsto \mathsf{v}$, where the normal C-BACKJUMP rule does not apply because multiple literals in the explanation become undef when the model assignment is removed. Because an inference cannot be placed on the trace in this situation, a decision literal is extracted from the explanation instead. This rule is borrowed from the model-constructing satisfiability calculus [37].

Properties. Given the undecidability of $\mathcal{L}_\#(\mathsf{L})$ in the general case, it should come as no surprise that $\mathsf{sat}^\#$ is not generally sound and complete. However, it is always sound, as shown in Theorem 2.

Theorem 2. *(Soundness)* Given an initial clause set C_0, $\mathsf{sat}^\#(\langle[], C_0\rangle)$ enters the terminal sat (respectively, unsat) state only if C_0 is satisfiable (respectively, unsatisfiable), whenever explain and reduce are sound with respect to L.

There are certain cases where $\mathsf{sat}^\#$ is complete as well. As long as the explain function can only return a finite number of clauses, then $\mathsf{sat}^\#$ will always terminate. Borrowing from a similar argument of de Moura and Jovanovíc [37], we can show this by imposing an order on the states entered by $\mathsf{sat}^\#$, and reasoning that the set of such states in any run progresses monotonicially. When the space of possible lemmas is finite, this order has a maximal element, and completeness follows.

Theorem 3. *(Completeness)* Given an initial clause set C_0, if explain returns clauses from a finite set, then $\mathsf{sat}^\#([], C_0)$ terminates in a finite number of steps.

5. A Linear-Integer Instantiation of $\mathsf{sat}^\#$

We describe an implementation of $\mathsf{sat}^\#$ for $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$ called countersat. It is based on Barvinok's algorithm [7] for counting the lattice points in convex polyhedra, and uses mesh-based black-box optimization to provide advice to the model construction process. countersat is implemented in 22,297 lines of C and C++ on top of Z3 [16], `libbarvinok` [44], the Nomad optimization library [2], and Mathematica.

5.1 Barvinok's Theory of Polyhedral Lattice Points

Barvinok presented a constructive proof that the number of lattice points in a convex rational polyhedron can be counted in polynomial time, when the dimension of the polyhedron is fixed in advance [7]. His proof is based on generating functions of polyhedral lattice points and the *algebra of polyhedra*, which takes as its basis the space of polyhedral lattice point indicator functions. Definition 7 gives the lattice-point generating function for a rational polyhedron (Definition 6).

Definition 6. *(Rational polyhedron).* A rational polyhedron $P \subset \mathbb{R}^d$ is the set of solutions of a finite system of linear inequalities with integer coefficients:

$$P = \{x \in \mathbb{R}^d \ : \ A\mathbf{x} + B \geq \gamma\}$$

where A, B, and γ are constants.

Definition 7. *(Lattice-point generating function [7]).* Let $P \subset \mathbb{R}^d$ be a rational polyhedron. With the set of lattice points $P \cap \mathbb{Z}^d$ in P, we associate the generating function in d complex variables $\mathbf{x} = (x_1, \ldots, x_d)$,

$$f(P; \mathbf{x}) = \sum_{m \in P \cap \mathbb{Z}^d} \mathbf{x}^m, \text{ where } \mathbf{x}^m = x_1^{m_1} \cdots x_d^{m_d}$$

Note that $f(P, [1, \ldots, 1]) = |P \cap \mathbb{Z}^d|$, so the generating function provides a way to count the number of lattice points in P. The details of Barvinok's proof are beyond the scope of this work, but we present his main result in Theorem 4.

Theorem 4. *(A. Barvinok [7])* There is a polynomial-time algorithm which, for each rational polyhedron $P \subset \mathbb{R}^d$, produces a rational function $f(P; \mathbf{x})$ in d complex variables $\mathbf{x} = (x_1, \ldots, x_n)$ with the following properties:

1. f is a *valuation*: if $P_1, \ldots, P_n$ are rational polyhedra whose indicator functions F_{P_i} satisfy a linear relation, then the rational functions satisfy the same relation.
2. If $m + P$ is a translation of P by an integer vector m, then $f(m + P; \mathbf{x}) = \mathbf{x}^m f(P; \mathbf{x})$
3. $f(P; \mathbf{x})$ matches the generating function for P on any $\mathbf{x}$ such that the series converges absolutely: $f(P; \mathbf{x}) = \sum_{m \in P \cap \mathbb{Z}^d} \mathbf{x}^m$
4. If P contains a straight line then $f(P; \mathbf{x}) = 0$.

Property *(3)* of Theorem 4 is perhaps the most important takeaway: in polynomial time for fixed dimension, we can compute the generating function of a rational polyhedron.

To support certain types of deduction over $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$, we make use of an extension to Barvinok's original theory over *rational parametric polyhedra* [44].

Definition 8. *(Rational Parametric Polyhedron).* A rational parametric polyhedron $P_{\mathbf{p}}$ is the set of solutions to a set of linear inequalities with integer coefficients of the form

$$P_{\mathbf{p}} = \{\mathbf{x} \in \mathbb{R}^d : A\mathbf{x} + B\mathbf{p} \geq \gamma\}$$

where $\mathbf{p} \in \mathbb{Z}^{d'}$ is the *parameter set* and γ is a constant vector.

Verdoolaege *et al.* give a polynomial-time algorithm (again, for fixed dimension) for computing valuations $f_{\mathbf{p}}(P_{\mathbf{p}}; \mathbf{x})$ over rational parametric polytopes, and evaluating them at $\mathbf{x} = \mathbf{1}$ [44]. The result is a *parametric enumerator*, which is a piecewise *quasi-polynomial* over the variables in $\mathbf{p}$ to integers, representing the number of lattice points in $P_{\mathbf{p}}$ for a given configuation of $\mathbf{p}$. A quasi-polynomial differs from a traditional polynomial in that its coefficients correspond to periodic functions with integral period. For the purposes of this work, the fact that the parametric enumerator returns a piecewise quasi-polynomial is not important, as we can use tools that operate over traditional polynomials where necessary by making appropriate modifications to the expression [39].

5.2 countersat: $\mathsf{sat}^{\#}$ for $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$

Using Barvinok's algorithm, we implemented $\mathsf{sat}^{\#}$ for counting over quantifier-free linear-integer arithmetic. In this section, we discuss a few of the most important components of countersat: a realization of reduce (Definition 5), explain (Definition 4), and genparam (Definition 3) using Barvinok's algorithm and Z3. We describe these components in the context of an example formula:

$$s_1^{\mathsf{s}} = s_2^{\mathsf{s}} \wedge \mathsf{count}(\{x, y\}, 0 \leq x \leq s_1^{\mathsf{s}} \wedge 0 \leq y \leq s_2^{\mathsf{s}}) = 120$$

This formula specifies a square with 120 lattice points; such a square does not exist, so the formula is unsatisfiable. In the following, we refer to the two atoms in this formula by the following symbols:

$$\begin{array}{lcl} l_1 & \equiv & s_1^{\mathsf{s}} = s_2^{\mathsf{s}} \\ l_2 & \equiv & \mathsf{count}(\{x, y\}, 0 \leq x \leq s_1^{\mathsf{s}} \wedge 0 \leq y \leq s_2^{\mathsf{s}}) = 120 \end{array}$$

Implementing reduce. Our implementation of reduce relies heavily on `libbarvinok`, which supports parametric counting of Presburger formulas containing existential quantifiers. Notice that these features allow non-polyhedral and even non-convex sets, so `libbarvinok` converts formulas into disjoint disjunctive-normal form and projects out existentially-quantified variables before applying Barvinok's theory. These operations may be expensive, so the use of these features in $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$ formulas should be minimized. Our example uses neither of these features, so applying `libbarvinok` to counting the base formula $0 \leq x \leq s_1^{\mathsf{s}} \wedge 0 \leq y \leq s_2^{\mathsf{s}}$ yields a parametric enumerator with two chambers, the first corresponding to the positive quadrant of the plane:

$$f(s_1^{\mathsf{s}}, s_2^{\mathsf{s}}) = \begin{cases} 1 + s_1^{\mathsf{s}} + s_2^{\mathsf{s}} + s_1^{\mathsf{s}} s_2^{\mathsf{s}} & \text{if } s_1^{\mathsf{s}} \geq 0 \wedge s_2^{\mathsf{s}} \geq 0 \\ 0 & \text{if } s_1^{\mathsf{s}} < 0 \vee s_2^{\mathsf{s}} < 0 \end{cases}$$

When called on a formula containing l_2, reduce caches this answer. If the trail passed to reduce contains values for s_1^{s} and s_2^{s}, it simply evaluates f on the given values and sends the result to Z3 for evaluation. For example, invoking $\mathsf{reduce}(l_2, [s_1^{\mathsf{s}} \mapsto 5, s_2^{\mathsf{s}} \mapsto 5])$ activates the first chamber, and yields the invalid formula $1 + (5) + (5) + (5)(5) = 120$. As there are no shared variables remaining in this formula, we ask Z3 for validity, and return the answer false.

Implementing explain. Our implementation of explain produces two types of lemmas: those arising from QF_LIA not involving counting theory, and those arising due to counting theory conflicts. To illustrate the first type, suppose that that explain is given a trail resulting from the unit-propagation of l_1 and l_2, and the following constraints on shared variables:

$$\mathsf{explain}(s_2^{\mathsf{s}} \neq 5, [\rightarrow l_1, \rightarrow l_2, s_1^{\mathsf{s}} \mapsto 3])$$

Recall that the first argument given to explain is implied by the trail, so its negation is conflicting. In this case, the conflict arises from the assignment of 3 to s_1^{s}, the assertion $s_2^{\mathsf{s}} = 5$, and the QF_LIA literal l_1. explain detects this by using Z3 to check the satisfiability of each assignment, all QF_LIA literals asserted on the trail, and the negation of the conflicting literal $s_2^{\mathsf{s}} \neq 5$:

$$s_1^{\mathsf{s}} = 3 \wedge s_1^{\mathsf{s}} = s_2^{\mathsf{s}} \wedge s_2^{\mathsf{s}} = 5$$

Z3 tells us that this formula is unsatsfiable, and returns the entire clause set as an unsatisfiable core, which is passed on by explain as its final result: $s_1^{\mathsf{s}} \neq 3 \vee \neg(s_1^{\mathsf{s}} = s_2^{\mathsf{s}}) \vee s_2^{\mathsf{s}} \neq 5$.

If the conflict arises due to some fact implied by counting theory, then explain takes a different approach. Often, the trail implies an upper- or lower-bound on shared variables. For example, suppose that we have the following call to explain: $\mathsf{explain}(s_2^{\mathsf{s}} \neq 10, [\rightarrow l_1, \rightarrow l_2, s_1^{\mathsf{s}} \mapsto 10])$. Recalling again that the negation of $s_2^{\mathsf{s}} \neq 10$ is in conflict with the trail, we see that the conflict does not come from l_1. Rather, it arises due to the count term, as

$$\mathsf{reduce}(l_2, [s_1^{\mathsf{s}} \mapsto 10, s_2^{\mathsf{s}} = 10]) \textit{ yields } (120 = 121)$$

explain can always return the naïve-but-valid lemma $\neg l_2 \vee s_1^{\mathsf{s}} \neq 10 \vee s_2^{\mathsf{s}} \neq 10$, however this often leads to non-terminating behavior, as $\mathsf{sat}^{\#}$ may continue to request lemmas of this type with an unbounded number of values for s_1^{s} and s_2^{s}. Instead, explain exploits that fact that the right-hand-side of l_2 is a constant, while the left-hand-side varies positively in s_1^{s} and s_2^{s}, and derives bounding conditions on these variables. Chamber polynomials always have bounded degree, so they can be decomposed into a finite set of *monotonic regions* with respect to each shared variable. Depending on which region the current assignment to a shared variable resides, and whether the current value for the count term is greater- or less-than its constant constraint, valid solutions to a given shared variable must be bounded from above or below its current value.

To make this more concrete, let us derive bounding conditions for the current example. We see that the current assignment applies to the first chamber polynomial and constraint,

$$p(s_1^{\mathsf{s}}, s_2^{\mathsf{s}}) = 1 + s_1^{\mathsf{s}} + s_2^{\mathsf{s}} + s_1^{\mathsf{s}} s_2^{\mathsf{s}}, \quad \phi \equiv s_1^{\mathsf{s}} \geq 0 \wedge s_2^{\mathsf{s}} \geq 0$$

We start with s_1^{s}, attempting to find its monotone regions in p. To do so, we apply *cylindrical algebraic decomposition* [15] (CAD) to the partial derivative $D_{s_1^{\mathsf{s}}} p(s_1^{\mathsf{s}}, s_2^{\mathsf{s}})$ of p over s_1^{s}, which produces a set of regions over which $D_{s_1^{\mathsf{s}}} p(s_1^{\mathsf{s}}, s_2^{\mathsf{s}})$ is sign-invariant. Care must be taken before applying CAD, as any floor terms in the quasi-polynomial must first be removed by adding fresh existentially-quantified variables [39]. We restrict the decomposition to the region specified by the chamber constraint ϕ, further simplifying the solution and propagating bounds when necessary. Our implementation uses Mathematica's CAD routine, which has a simple API for accomplishing this restriction. This operation tells us that p is monotonic in s_1^{s} whenever $s_1^{\mathsf{s}} \geq 0$. Similarly, p is monotonic in s_2^{s} whenever $s_2^{\mathsf{s}} \geq 0$. From this, we conclude that any satisfying solution to these variables will be bounded from above by the current assignment, for at least one variable. explain returns the lemma:

$$\neg l_1 \vee \neg l_2 \vee \neg(s_1^{\mathsf{s}} \geq 0) \vee \neg(s_2^{\mathsf{s}} \geq 0) \vee s_1^{\mathsf{s}} < 10 \vee s_2^{\mathsf{s}} < 10$$

Notice that the nearest solution that satisfies all QF_LIA constraints, as well as this lemma, is $s_1^{\mathsf{s}} \mapsto 9, s_2^{\mathsf{s}} \mapsto 9$, which by the same reasoning causes explain to return the lemma:

$$\neg l_1 \vee \neg l_2 \vee \neg(s_1^{\mathsf{s}} \geq 0) \vee \neg(s_2^{\mathsf{s}} \geq 0) \vee s_1^{\mathsf{s}} > 9 \vee s_2^{\mathsf{s}} > 9$$

This quickly leads to termination with unsat.

Note that in some cases, CAD returns a region specified by non-linear constraints, or non-integer constants. In the latter case, we simply round to the nearest appropriate integer. In the former, we must *specialize* the chamber polynomial, replacing variables with constants from the current assignment until CAD returns a linear region; in the worst case, this occurs when the polynomial is one-dimensional. Specialization yields weaker lemmas. explain is complete on a practical fragment of $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$, termed the

monotone fragment (Definition 9). All but one of our benchmarks (**diffpriv**) are monotone.

Definition 9. *Monotone Fragment.* A $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$ formula is in the *monotone fragment* (or simply *monotone*) if it is equivalent to a formula that satisfies the following conditions: *1)* Any atom containing a binary relation has parametric count terms on only one side of the relation, and each shared variable appears in at most one such term in the atom. *2)* The coefficients of all parametric count terms in a given atom have matching signs. *3)* The chamber polynomials of each count term are all monotonic in every shared variable, within their corresponding chamber constraints.

Theorem 5. Given a monotone clause set from $\mathcal{L}_{\mathsf{count}}(\mathsf{QF\_LIA})$, countersat terminates in finite steps.

Implementing genparam. Implementing $\mathsf{genparam}(C, x)$ is possible using Z3's built-in support for linear-integer arithmetic: simply pass the clauses C to Z3, and collect a model for x. countersat generates advice for $\mathsf{genparam}(C, x)$ using mesh-based black-box optimization [2]. Black-box optimization assumes nothing about the structure of its objective function, instead relying on a module to compute its value on candidate points, making it ideal in this setting. We construct an objective function g over the shared variables that operates as a penalty function on models, mirroring the penalty methods [32] used in constrained optimization. We use black-box optimization over this objective, applying Barvinok's algorithm as needed to evaluate count terms, in an attempt to find a (nearly) satisfying assignment for x. When the procedure completes, typically after a timeout, the best solution is used to construct an advice clause for genparam that specifies a region surrounding this value of x. The region is subsequently expanded until either a valid assignment is found, or a threshold is reached.

Continuing with the current example, countersat will derive the following objective function:

$$g(s_1^{\mathsf{s}}, s_2^{\mathsf{s}}) = (s_1^{\mathsf{s}} - s_2^{\mathsf{s}})^2 + (b(s_1^{\mathsf{s}}, s_2^{\mathsf{s}}) - 120)^2$$

$b(s_1^{\mathsf{s}}, s_2^{\mathsf{s}})$ corresponds to the parametric enumerator for the count term in l_2, which is evaluated on-demand. The procedure will pass advice to genparam that suggests solutions in this vicinity: $7 \leq s_1^{\mathsf{s}}, s_2^{\mathsf{s}} \leq 13$. The size of the vicinity is heuristic; our implementation uses a region of size 7 in each dimension.

6. Evaluation

We evaluated the performance of countersat on a set of benchmarks derived from privacy and confidentiality verification tasks. We sought to answer the following questions:

- *Can* countersat *be brought to bear on verifying privacy-preserving computations?*
- *Are the CAD-based lemmas generated by* explain *necessary, or do simpler schemes suffice in most cases?*
- *Is the use of CAD in* explain *practical, or is the double-exponential complexity an issue in common instances?*
- *Is advice for* genparam *useful, or does it tend to get in the way?*

Summarizing, we found that countersat was able to verify the privacy properties of several programs written for *secure multi-party computation* [8], oftentimes in a few seconds. We found that in many cases CAD-based lemmas are needed for termination, and often increase performance by orders of magnitude. Additionally, the average time spent in CAD over all benchmarks was small (1.3 seconds). We found that our optimization-based advice routine for genparam was essential to performance in a few cases, offering as much as a 118× performance increase, while in others causing a modest slow-down.

6.1 Benchmarks

We evaluated countersat on eleven benchmarks; the details of most are given in Figure 4(a). Two correspond to geometry problems (**hypercube** and **a3**), and nine to privacy verification problems. Of the nine verification benchmarks, two correspond to programs written by us (**editdist** and **diffpriv**), and the remaining were distributed as part of the Fairplay secure multi-party compiler [8]. Fairplay compiles programs written in Secure Function Definition Language (SFDL), which resembles a subset of C with bounded loops, into a Boolean circuit. Each of the verification benchmarks suffixed with **-circ** was generated directly from the Boolean circuit compiled by Fairplay. Observing that Boolean satisfiability is reducible to linear-integer programming, we achieve this translation by converting CNF clauses into 0-1 linear-integer constraints. The remaining verification benchmarks were translated from Fairplay by hand, without being first reduced to Boolean operations.

The privacy properties we verified relate to how much of one party's input another party is able to learn, given their input and the output of the computation. Writing the verification conditions for these properties follows in a similar vein to the running example given in Section 2. For example, to verify the voting program, we want to demonstrate that any observer who knows fewer than $\lfloor \frac{n}{2} \rfloor$ parties' votes, and the output of the program, cannot learn any of the other parties' votes. Supposing there are six parties and m candidates, we select two parties' inputs v_1, v_2 and the output o as *known observations*, and establish that if only these values are known to an observer, then the remaining inputs can still take their full range of values: we make v_1, v_2, and o shared variables, and assert the negation of our goal:

$$\mathsf{count}(v_3, \ldots) < m \vee \cdots \vee \mathsf{count}(v_6, \ldots) < m$$

A satisfying assignment on v_1, v_2 and o to this formula corresponds to a set of votes for the two known parties, and an outcome for the election, that allows one to learn more than intended about the other parties' votes.

6.2 Results

All experiments were performed on a MacBook Pro with 8 GB of memory and a 4-core 2.2 GHz Intel Core i7 running OS X 10.8. Each benchmark was given a 30-minute time limit. The results are displayed in Figure 4(b). For each benchmark, we give performance characteristics for three configurations: using optimization-based advice for genparam and CAD-based lemmas ("With advice"), using no advice with CAD-based lemmas ("No adv."), and using no advice and no CAD-based lemmas ("No CAD"). We also evaluate the ability of Z3's non-linear integer solver to directly solve the problems, given the chamber polynomials produced by libbarvinok ("NLSAT"). For each configuration, we give the total runtime in seconds. For the first configuration (using advice with CAD lemmas), we also give the amount of time spent model counting in `libbarvinok` (which is invariant across configurations) and the amount of time spent generating advice.

countersat was able to complete each of the benchmarks, taking anywhere from a few seconds to several minutes to finish. As previously mentioned, the programs from which these benchmarks were derived were small. Even so, when **editdist**, **diffpriv**, and **median** were translated into Boolean circuits, countersat timed out in the model counting phase. We addressed this by translating the code into $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$ by hand, treating integers as single variables rather than a set of binary digits. In this form, countersat was able to complete verification. Notice the speedup from **auction-circ** to **auction**; this suggests that the Boolean encoding introduces a significant overhead in counting. In future work we will explore approximation methods, as well as formula decomposition strate-

Name	*Description*
editdist	Two-party edit distance. Result should not leak contents of a full character from party's string.
diffpriv	Differentially-private set cardinality.
median	Median of two parties' lists. Verify that one party cannot cause the other to leak his entire list.
auction	Vickrey auction. Verify that the result does not leak more of a player's bid than implied by the winning price.
auction-circ	Same as above, but translated directly from circuit.
mill-circ	Millionaire's problem.
manymill-circ	Multi-party variant of above.
keydb-dirc	Keyed database lookup. Result should leak no information about unselected rows.
voting-circ	Simple majority voting circuit. A coalition of fewer than half minus one should not learn another party's vote.

(a) Privacy verification benchmarks.

	With advice			No adv.	No CAD	NLSAT
	count	*advice*	***total***	***total***	***total***	***total***
hypercube	0.01	0.47	0.55	65.03	–	NA
a3	0.01	2.66	5.83	663.81	–	NA
editdist	0.42	0.57	20.91	20.19	–	NA
diffpriv	4.76	1.41	69.41	55.89	–	NA
median	0.20	1.57	17.60	34.81	–	NA
auction	0.03	0.01	0.57	0.56	–	NA
auction-circ	50.69	0.78	125.42	125.63	126.73	125.87
mill-circ	0.33	0.40	0.90	0.51	0.45	0.45
manymill-circ	0.93	0.01	2.45	2.46	0.24	0.23
keydb-circ	29.26	0.66	240.81	235.24	236.47	125.27
voting-circ	0.94	0.01	2.46	2.45	2.41	2.40

(b) Performance characteristics for `countersat`. All times are measured in seconds. – signifies timeout, NA signifies that the solver returned "unknown".

Figure 4. `countersat` evaluation benchmarks and results.

gies that utilize the inclusion-exclusion principle, to mitigate this bottleneck on larger benchmarks.

The results indicate that in a few cases, advice for `genparam` is helpful, at times yielding improvements two orders of magnitude faster than without. In most cases, it neither helped performance nor hurt it significantly, although in the worst case added an additional fourteen seconds to the total (**diffpriv**). The discrepancy can be explained by the structure of the parametric enumerator's chamber space: whenever the chambers encompass a large area of the parameter space and there is substantial variation in the chamber polynomial, advice is generally useful. This property tends to hold for geometric problems, as well as verification problems without many constraints on parameters (e.g., **median**). When this property does not hold, a chamber is usually dispatched as not feasible before advice is even generated; in the few cases it is, it tends to produce no useful information at a slight cost.

The results also indicate that CAD-based lemmas are necessary in all but the circuit-based benchmarks. This is due to the fact that the parameters in the circuit benchmarks each take Boolean values, so the bounds produced by CAD lemmas are of little value. In other cases, without CAD lemmas `countersat` tends to exhaust each chamber before ruling it out, which quickly becomes intractible. Finally, the approach of translating libbarvinok's chamber polynomials into nonlinear-integer constraints and solving directly using Z3's engine worked only for the circuit benchmarks – Z3 returned "unknown" on all other benchmarks. We suspect that this result follows from the same condition that allowed non-CAD solving to succeed: in circuit benchmarks, parameters correspond to binary values, whereas in the other benchmarks the values are arbitrary integers and the chamber polynomials are often complex non-linear polynomials.

7. Related Work

Model Counting. Logicians have studied the expressive power and other properties of various first-order and fixed-point logics extended with counting quantifiers [9, 24, 25, 31]; see [23] for an overview. The syntax and semantics of our logic are inspired by these early works, but given our specific focus on the application of $\mathcal{L}_{\#}(\mathsf{L})$ to verification, it differs in a few aspects. First, we do not allow nested counting terms. This makes the logic less expressive, but in our experience does not seem to affect its application to privacy properties. Second, we parameterize both the syntax and semantics on a base logic, effectively making $\mathcal{L}_{\#}(\mathsf{L})$ a *logic schema* that can be instantiated over many different base logics. Lastly, we explicitly consider the problem of satisfiability over parameters and develop an algorithm for its decision problem, while these works focused on expressiveness and complexity.

Modal logics have been extended with counting operators, including corresponding satisfiability procedures [3, 4]. In these settings, counting terms may be compared against constants. Many have explored the problem of counting quantifier-free propositional models (called #SAT); see Gomes *et al.* [22] for a survey. To the best of our knowledge, parametric counting has not been explored in this context. Feifei *et al.* extend the problem to SMT [34], supporting volume computation by counting solutions to linear-integer atoms and summing over all valid Boolean combinations, but they do not consider parameters or a corresponding notion of satisfiability. In the artificial intelligence community, some have proposed [13, 41] reducing Bayesian inference problems to weighted #SAT, but parameters are not used. Luu *et al.* studied the problem of counting models over string domains [33], and applied their algorithm to quantifying information flow in programs that operate over encrypted data. We will explore the possibility of incorporating their technique as a solver for base logics that use string values. Closely related to our procedure is the work on counting linear-integer models [7, 12, 14, 39, 44]. Our decision procedure makes heavy use of this work, defining a useful notion of satisfiability over the parametric counting procedures.

Privacy and Confidentiality Verification. There is a large body of work related to verifying programs against privacy and confidentiality properties. Partial information flows must be handled using declassifiers, which can leak unintended information. Sabelfeld and Myers proposed a model of *delimited release*, where acceptable disclosure is specified syntactically in the program text as an expression. It may be possible to use $\mathcal{L}_{\#}(\mathsf{L})$ to provide assurance that a delimited release prevents unintended leakage. Others have explored *quantitative information flow* [5, 27, 28, 35, 45] as a relaxation of non-interference, wherein various information theoretic concepts are used to impose bounds on the "amount" of information leaked by a program. Model counting is intimately related to this line of research, and has well-documented [5] applications to its verification problem. Klebanov recognized the potential of parametric model counting when applied to quantitative information flow [28], but did not further develop the notion of satisfiability necessary for automated reasoning, or consider applying the primitive to other privacy and confidentiality properties.

Gaobardi *et al.* [21] gave a linear-dependent type system capable of verifying differential privacy in functional programs. However, it is not able to verify programs that achieve differential privacy using "non-standard" techniques. This is not an issue with the strategy outlined here, which establishes the needed privacy

property from lower-level semantics. Others have studied relational variants of the Hoare logic [10] with applications to information flow analysis. Barthe *et al.* [6] extended this reasoning to approximate relations over probablistic programs, with the immediate application to verifying differential privacy. This framework was implemented alongside the Coq proof assistant, and used to produce machine-verified proofs of differential privacy for several interesting programs. We see a deep connection between their work and ours, and plan to explore complementary verification strategies as immediate next steps in future work.

8. Conclusion

In this paper, we introduced the problem of *model-counting satisfiability* and its application to verifying notions of privacy. We introduced a new logic for expressing instances of this problem, developed an abstract decision procedure for it, and instantiated it on linear-integer arithmetic. We described an effective routine for producing counting theory lemmas over this logic, that is complete on a useful fragment. We applied our procedure to the verification of several privacy-preserving programs, and showed that it often completes in a matter of seconds. In the future we will attempt to verify larger programs, which will undoubtedly involve incorporating our decision procedure into a more sophisticated program analysis and abstraction framework. We will also study approximation and decomposition strategies that help mitigate the worst-case complexity of linear-integer model counting in practice.

References

[1] Insurance data: Very personal finance. *The Economist*, June 2012.

[2] M. Abramson, C. Audet, G. Couture, J. Dennis, Jr., S. Le Digabel, and C. Tribes. The NOMAD project. `http://www.gerad.ca/nomad`.

[3] C. Areces, G. Hoffmann, and A. Denis. Modal logics with counting. In *Logic, language, information and computation*, 2010.

[4] F. Baader, M. Buchheit, and B. Hollunder. Cardinality restrictions on concepts. In *Advances in Artificial Intelligence*. 1994.

[5] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *Oakland*, 2009.

[6] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *POPL*, 2012.

[7] A. I. Barvinok. A polynomial-time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematical Operations Research*, Nov. 1994.

[8] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS*, 2008.

[9] M. Benedikt and H. J. Keisler. Expressive power of unary counters. In *Database Theory*, 1997.

[10] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[11] R. Bhaskar, A. Bhowmick, V. Goyal, S. Laxman, and A. G. Thakurta. Noiseless database privacy. In *Asiacrypt*, 2011.

[12] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *PLDI*, 2001.

[13] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, Apr. 2008.

[14] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *VLSI Signal Process. Syst.*, July 1998.

[15] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In *Automata Theory and Formal Languages*, 1975.

[16] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[17] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *CACM*, July 1977.

[18] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. *Journal of Computer Security*, Jan. 2004.

[19] C. Duhigg. How companies learn your secrets. *New York Times*. February 16, 2012.

[20] C. Dwork. Differential privacy: a survey of results. In *Theory and applications of models of computation*, 2008.

[21] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *POPL*, 2013.

[22] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of Satisfiability*. 2009.

[23] E. Grädel, P. G. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Y. Vardi, Y. Venema, and S. Weinstein. *Finite Model Theory and Its Applications*. Springer-Verlag New York, Inc., 2005.

[24] E. Grädel and M. Otto. Inductive definability with counting on finite structures. In *Workshop on Computer Science Logic*, 1993.

[25] S. Grumbach and C. Tollu. On the expressive power of counting. *Theoretical Computer Science*, 1995.

[26] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *USENIX Security*, 2011.

[27] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *ACSAC*, 2010.

[28] V. Klebanov. Precise quantitative information flow analysis using symbolic model counting. In *Workshop on Quantitative Aspects in Security Assurance*, 2012.

[29] V. Klebanov, N. Manthey, and C. Muise. SAT-based analysis and quantification of information flow in programs. In *Quantitative Evaluation of Systems*, 2013.

[30] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *CAV*, 2012.

[31] L. Libkin. Logics with counting, auxiliary relations, and lower bounds for invariant queries. In *LICS*, 1999.

[32] D. Luenberger and Y. Ye. *Linear and Nonlinear Programming*. 2008.

[33] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *PLDI*, 2014.

[34] F. Ma, S. Liu, and J. Zhang. Volume computation for Boolean combination of linear arithmetic constraints. In *CADE*. 2009.

[35] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, 2008.

[36] I. Mironov. On significance of the least significant bits for differential privacy. In *CCS*, 2012.

[37] L. Moura and D. Jovanović. A model-constructing satisfiability calculus. In *VMCAI*. 2013.

[38] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Joural of the ACM*, Nov. 2006.

[39] W. Pugh. Counting solutions to Presburger formulas: how and why. In *PLDI*, 1994.

[40] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Software Security – Theories and Systems*. 2004.

[41] T. Sang, P. Beame, and H. Kautz. Solving Bayesian networks by weighted model counting. In *AAAI*, 2005.

[42] L. Sweeney. k-anonymity: a model for protecting privacy. *Journal on Uncertainty and Fuzziness in Knowledge-Based Systems*, Oct. 2002.

[43] J. Valentino-Devries, J. Singer-Vine, and A. Soltani. Websites vary prices, deals based on users' information. *Wall Street Journal*. December 24, 2012.

[44] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, Mar. 2007.

[45] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *POPL*, 2000.

[46] S. Zdancewic and A. C. Myers. Robust declassification. In *CSF*, 2001.

A. Proof of Theorem 1

We begin with a brief sketch. We show that the problem of determining whether an integer solution to a Diophantine equation exists can be reduced to determining the satisfiability of a formula in $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$. A Diophantine equation is of the form:

$$c_1 x_1^{q_1} + \ldots + c_n x_n^{q_n} = 0$$

where each a_i and q_i is an integer. It is known that the problem of determining the existence of integer solutions to $x_1, \ldots, x_n$ for an arbitrary Diophantine equation is undecidable (*viz.* the negative solution to Hilbert's tenth problem). The key to the reduction is that a variable x in a Diophantine equation can be replaced with the expression

$$\mathsf{count}(x, 0 \leq x < x^{\mathsf{s}})$$

For any assignment to x^{s}, this expression will take the same value.

Let $p(x_1, \ldots, x_n)$ be a Diophantine equation. Assume wlog. that we restrict the solutions of p to non-negative integers. This simplifies the proof, as count expressions cannot return negative values. We show how to obtain a $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ formula $\phi(x_1^{\mathsf{s}}, \ldots, x_n^{\mathsf{s}})$ that is satisfiable if and only if p has a solution in the integers. The transformation from p to ϕ proceeds in two steps:

1. "Unroll" each exponent appearing in p, so that

   $$x^q \text{ becomes } \underbrace{x \times \cdots \times x}_{q \text{ times}}$$

   Note that while this transformation increases the size of the formula exponentially, this is not a concern, as we only consider decidability.

2. Replace each instance of each variable with a counting operation corresponding to its value in p. In other words,

   $$x \text{ becomes } \mathsf{count}(x, 0 \leq x < x^{\mathsf{s}})$$

First notice that ϕ is actually in $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$. Each term in p consists of a multiplication and an exponentiation. Step 1 transforms the exponentiation into a sequence of multiplications, so each individual term is in $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$. The terms are combined with addition, and the only other operation is an equality comparison, both of which are in $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$. Each count term added in step 2 operates over two conjoined inequality comparisons (i.e., $0 \leq x$ and $x < x^{\mathsf{s}}$), and $0 \leq x \wedge x < x^{\mathsf{s}}$ is in QF_LIA. Lastly, these terms are also finite-base, because any substitution of x^{s} with an integer yields a bounded value for the term. As such, the count terms have well-defined semantics in $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$.

Now observe that any satisfying assignment α to $\phi(x_1^{\mathsf{s}}, \ldots, x_n^{\mathsf{s}})$ corresponds via the identity mapping to a solution for $p(x_1, \ldots, x_n)$. First, $\alpha(x_i^{\mathsf{s}})$ is an integer for all i. Next, notice that the left-hand side of $\phi(\alpha(x_1^{\mathsf{s}}), \ldots, \alpha(x_n^{\mathsf{s}}))$ evaluates to the same value as that in $p(\alpha(x_1^{\mathsf{s}}), \ldots, \alpha(x_n^{\mathsf{s}}))$. We can see this on a term-by-term basis:

1. In ϕ, each $\mathsf{count}(x, 0 \leq x < x^{\mathsf{s}})$ evaluates to $\alpha(x^{\mathsf{s}})$. In p, each occurrence of x also evaluates to $\alpha(x^{\mathsf{s}})$.
2. Via the previous step, replacing $\mathsf{count}(x, 0 \leq x < x^{\mathsf{s}}) \mapsto \alpha(x^{\mathsf{s}})$ in ϕ and $x \mapsto \alpha(x^{\mathsf{s}})$ in p, we see that the terms in ϕ evaluate identically to those in p:

   $$c_i \underbrace{\alpha(x_i^{\mathsf{s}}) \times \cdots \times \alpha(x_i^{\mathsf{s}})}_{q_i \text{ times}} = c_i \alpha(x_1^{\mathsf{s}})^{q_i}$$

Thus, p has a solution in the integers whenever ϕ is satisfiable. The same result holds in the opposite direction by identical reasoning, so it follows that reduction to $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ satisfiability yields a procedure for solving Diophantine equations. □

B. Proof of Theorem 2

First, notice that the only terminal states are SAT and UNSAT. Showing this amounts to demonstrating that at least one rule applies whenever the following condition holds:

$$\begin{array}{lll} & (\neg\mathsf{satisfied}(C, M) & \wedge \quad \neg\mathsf{conflict}(C, M)) \\ \vee & (\neg\mathsf{satisfied}(C, M) & \wedge \quad \mathsf{explain}(\mathsf{false}, M) \neq \mathsf{false}) \\ \vee & (\neg\mathsf{satisfied}(C, M) & \wedge \quad M \text{ has a decided literal}) \end{array}$$

This condition corresponds to the negation of the conjoined preconditions for the SAT and UNSAT rules. In the following, to make our explanations as simple as possible we assume that each case is mutually-exclusive. For example, the first bullet corresponds to the case where C is *not* satisfied by M, there is *not* a conflict between C and M, $\mathsf{explain}(\mathsf{false}, M) = \mathsf{false}$, and M has *no* decided literals:

- $(\neg\mathsf{satisfied}(C, M) \wedge \neg\mathsf{conflict}(C, M))$ implies that at least one shared variable is not assigned in M or a literal from the C has not been decided or propagated. In this case, either DECIDE, C-DECIDE, PROPAGATE, C-PROPAGATE, or C-MODEL-DECIDE applies.
- $(\neg\mathsf{satisfied}(C, M) \wedge \mathsf{explain}(\mathsf{false}, M) \neq \mathsf{false})$ implies that the current trail is conflicting and a non-trivial explanation exists. Thus, because explain does not return false, one of the backjump and clause learning rules can be applied to add a new non-trivial clause to C.
- $(\neg\mathsf{satisfied}(C, M) \wedge M \text{ has a decided literal})$ implies that either a literal appearing in C has not yet been decided, a variable has not been assigned, or the current trail is conflicting. In the first two cases, DECIDE or C-DECIDE can be applied. In the third case, one of the backjump rules can be applied.

The above reasoning shows that whenever the preconditions for SAT and UNSAT do not hold, another rule can be applied. Thus, the procedure will not terminate in any state outside of SAT or UNSAT.

Now, suppose that $\mathsf{sat}^{\#}$ enters SAT. The precondition for this state is simply: $\mathsf{satisfied}(C, M)$. Then at least one literal from each clause in C evaluates to true, and $\mathsf{value}_c(l, M) = \mathsf{true}$ for all such literals. The first condition implies that C is satisfiable *clausally*, i.e., it does not contain any contradictions on a purely Boolean level. The second condition implies that the values of shared variables given by the model assignments in M do not invalidate the Boolean satisfiability, i.e., the semantics of $\mathcal{L}_{\#}(\mathsf{L})$applied to C under the assignments in M agree with the clausal satisfiability. To see why, consider the definition of $\mathsf{value}_c(l, M)$ (Figure 3c). For any l such that $\mathsf{value}_b(l, M) = \mathsf{true}$, the first condition implies that $\mathsf{reduce}(M, l) = \mathsf{true}$. This implies that α_M is total over $V_{\mathsf{share}}(l)$, and that $\mathcal{O}_{\mathsf{sat}}(\alpha_M(l)) = \mathsf{true}$. This, in turn, implies that C is satisfiable as a QF_NRA formula, so by Definition 2 C is satisfiable.

On the other hand, if $\mathsf{sat}^{\#}$ enters UNSAT, then three conditions must hold:

1. $\mathsf{conflict}(C, M) = \mathsf{true}$
2. $\mathsf{explain}(\mathsf{false}, M) = \mathsf{false}$
3. M has no decided literals, i.e., further backtracking is not possible.

We list the causes that can result in these conditions holding simultaneously, and explain how each implies that C is not satisfiable.

- *(Model space exhaustion).* The model α_M causes one of the clauses to evaluate to false, thus causing the conflict on C, M. Additionally, explain returns false, so there are no other valid model assignments to try. To see why, suppose that the trail consists of assignments $[x_1^{\mathsf{s}} \mapsto \mathtt{v}_1, \ldots, x_n^{\mathsf{s}} \mapsto \mathtt{v}_1]$ and a formula ϕ. Then because explain returns a valid theory lemma,

we see that $x_1^{\mathsf{s}} \neq \mathtt{v}_1 \vee \cdots \vee x_n^{\mathsf{s}} \neq \mathtt{v}_n \vee \neg\phi \vdash \mathsf{false}$. It follows that no satisfying assignment to the shared variables exists, and C is unsatisfiable.

- *(Conflicting lemmas and inferences).* PROPAGATE and C-PROPAGATE have inferred a set of literals that causes one of the clauses to evaluate to false, and there are no model assignments on the trail. M contains no decided literals, so each propagation is implied by the original clause set and the lemmas added by LEARN, C-MODEL-PROPAGATE, C-LEARN, and the backjump rules. In each case, the lemma must be a valid propositional or counting theory deduction, so the inferences on the trail are all valid consequences of the original clause set. C must be unsatisfiable.
- *(Mixed model-based and base logic reason).* PROPAGATE and C-PROPAGATE have inferred a set of literals that, along with the model assignments on the trail, causes one of the clauses to evaluate to false. explain returns false, so there are no other assignments to shared variables that agree with the inferred literals, for if there were then explain would have returned $\phi \to l$, where $l \to x^{\mathsf{s}} \neq \alpha_M(x^{\mathsf{s}})$, for some ϕ and x^{s} assigned on M. It follows that C is unsatisfiable.

We conclude that when $\mathsf{sat}^{\#}$ enters UNSAT, C is unsatisfiable. □

C. Proof of Theorem 3

We begin with a brief sketch. This proof proceeds along the lines of that of Theorem 1 in [37]. We define a lexicographic partial order on states, and show that the transition rules in Figure 3 produce a monotonically-increasing sequence of states. One might think that C-DECIDE causes termination problems due to its freedom in generating literals from L. However, these literals are only passed to genparam, which can only affect the trail through model assignments, which are given smaller weight in the ordering. Similarly, one might expect C-MODEL-DECIDE to cause issues from its use of explain outside of a conflicting state; because the explanation is added directly to the trail, rather than the set of clauses, $\mathsf{sat}^{\#}$ avoids a non-terminating sequence of FORGET and C-MODEL-PROPAGATE applications. We give details of the proof below.

Following the termination proof of de Moura and Jovanović [37], we define a lexicographic partial order on $\mathsf{sat}^{\#}$ states. It is based on a weight assignment w for trail elements:

$$\begin{array}{lcl} w(\text{model assignment}) & = & 0 \\ w(\text{decided literal}) & = & 1 \\ w(\text{propagated literal}) & = & 2 \end{array}$$

Propagated literals are given the most weight, followed by decided literals, and finally model assignments. This ensures that when decisions made by the procedure are replaced with propagations *implied by* the current state, the new state is ranked higher than its predecessor. Similarly, when model assignments are replaced with decided literals, the resulting state is given a higher rank.

Now we define the partial order $\prec$ by five rules, making use of a secondary ordering $\sqsubset$ over trail assignments:

1)	$[\,] \sqsubset M$	iff	$M \neq [\,]$
2)	$[a, M_1] \sqsubset [b, M_2]$	if	$w(a) < w(b)$
3)	$[a, M_1] \sqsubset [b, M_2]$	if	$w(a) = w(b) \wedge M_1 \sqsubset M_2$
4)	$\langle M_1, C_1\rangle \prec \langle M_2, C_2\rangle$	if	$M_1 \sqsubset M_2$
5)	$\langle M_1, C_1\rangle \prec \langle M_2, C_2\rangle$	if	$M_1 = M_2 \wedge \lvert C_1\rvert > \lvert C_2\rvert$

Notice that $\prec$ is covariant in the lexicographic order over trail weights, and contravariant in the cardinality of the clause set.

Our assumption that explain returns clauses from a finite set C_{univ} implies minimal and maximal elements for a given application of $\mathsf{sat}^{\#}([\,], C_0)$: $\langle [\,], C_{\mathsf{univ}}\rangle$ is minimal, and any state containing a trail with $\lvert C_{\mathsf{univ}}\rvert$ propagations and a total model assignment, with clause set C_0 is maximal among those that will ever appear in a given application of the procedure. We have now to show that any valid sequence of applications is monotonic in $\prec$ after a finite number of steps. We proceed by cases:

- We can ignore the rules SAT and UNSAT, as their resulting states cause $\mathsf{sat}^{\#}$ to terminate immediately.
- Any rule that adds an element to the trail is immediately monotonic, as $\langle M, C_1\rangle \prec \langle [M, a], C_2\rangle$ by rules 1 and 3. This covers DECIDE, PROPAGATE, C-DECIDE, C-PROPAGATE, and C-MODEL-DECIDE.
- Any rule that removes a clause from C is monotonic by rule 5, so the result holds for FORGET.
- If BACKJUMP or C-BACKJUMP is applied, then the state will transition from $M_1 = [M, l, N]$ to $M_2 = [M, c \to l']$. Recall that $w(l) < w(c \to l')$, so by rules 1 and 3 $M_1 \prec M_2$.
- If C-BACKJUMP-DECIDE is applied, then the state will transition from $M_1 = [M, x^{\mathsf{s}} \mapsto \mathtt{v}, N]$ to $M_2 = [M, c \to l']$. Similar reasoning holds as with BACKUMP and C-BACKJUMP: $w(x^{\mathsf{s}} \mapsto \mathtt{v}) < w(c \to l')$, so rules 1 and 3 imply that $M_1 \prec M_2$.
- At first blush, LEARN and C-LEARN seem to pose a problem, as they result in larger clause sets, and $\prec$ is contravariant in the cardinality of the clause set. However, they must eventually transition to either unsat, or a greater state with a non-conflicting trail in a finite number of steps. Observe that in the "worst" case, all possible clauses from C_{univ} are learned, and the the clause set will reach its minimal configuration. When this occurs, LEARN and C-LEARN are no longer applicable, so a different rule must apply. Notice that the state cannot change without raising the rank of the trail M on $\sqsubset$, as the only way to do so would be FORGET, which cannot be applied on conflicting trails, and the presence of a conflict is a necessary precondition for LEARN and C-LEARN). Thus, the only rules that can apply are UNSAT (which we can ignore, as it leads to immediate termination), one of the backjump rules, or one of the propagate rules. When this happens, the resulting state will be ranked higher than the current regardless of the size of the new clause set (rule 4). Finally, because none of the rules move from $\langle M_1, C_1\rangle$ to $\langle M_2, C_2\rangle$ with $M_2 \sqsubset M_1$, this move to higher ranks is permanent.

To summarize, all rules except LEARN, C-LEARN, and FORGET transition the trail to higher ranks over $\sqsubset$, and are thus monotonic on $\prec$. Because $\prec$ is contravariant on the cardinality of the clause set, FORGET is monotonic. The only rules are are not immediately monotonic over single transitions are LEARN and C-LEARN; however, because only a finite number of clauses can be learned, these rules cannot be repeated infinitely, and must eventually transition to a higher-ranked state via an application of a backjump, propagation, or terminal UNSAT rule. Thus, the eventual monotonicity of the transitions, and the existence of a maximally-ranked state implies that the procedure must terminate after a finite number of transitions. □

D. Proof sketch of Theorem 5

Utilizing Theorem 3, we show that explain returns lemmas from a finite set on monotone problems. In the case of one shared variable x^{s}, we can assume without loss of generality that each atom containing a count term is of the form $c_1 \times \mathsf{count}(x^{\mathsf{s}}, \phi) \leq c_2$ for some constants c_1 and c_2 (see condition 1 of Definition 9). There are only finitely many points between any conflicting value of x^{s} and a value that satisfies the constraint (condition 3). If no such value exists, then the chamber space for this count term is finite,

and cannot cause explain to generate an infinite sequence of lemmas. This reasoning extends to multiple variables without much effort. This is due to condition 1, which stipulates that count terms appearing in a given atom cannot depend on the same shared variable, and condition 2, which disallows subtraction of count terms. These assumptions let explain work monotonically towards a satisfying solution or a conflict. Any conflict that does not result in termination will initiate another finite sequence of lemmas. Conditions 1 and 2 together imply bounds on each shared variable, so this process will eventually terminate. □