Understanding Privacy in the Era of "Privacy is Dead":
Inference Attacks and New Defenses

by

Matthew Fredrikson

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 08/14/2015

The dissertation is approved by the following members of the Final Oral Committee:

Somesh Jha, Professor, Computer Sciences
Benjamin Livshits, Researcher (Microsoft Research)
C. David Page Jr., Professor, Biostatistics and Medical Informatics
Thomas Reps, Professor, Computer Sciences
Thomas Ristenpart, Associate Professor, Computer Science (Cornell Tech)

For Emily

ACKNOWLEDGMENTS

This dissertation is the culmination of many years of work, and would not have been possible without the support and dedication of many collaborators, family, and friends. My advisor, Somesh Jha, has guided me well through all aspects of life as a computer scientist—his patience, encouragement, and sense of humor are the reason this document exists, and I can only hope to emulate his unique advising style. For this, I will remain indebted to Somesh indefinitely.

I also thank the members of my committee—Thomas Reps, Ben Livshits, Thomas Ristenpart, and David Page. At various points in my career as a graduate student, they have essentially served as my co-advisors, and I am grateful to have had the opportunity to work with them on interesting problems. My research, and this document, would look very different without their hard work and guidance. In addition to those on my committee, the work in this dissertation would not have been possible without the help of Eric Lantz and Simon Lin. Their expertise in bioinformatics, statistics, and medicine provided the direction needed to realize the most important results in Chapter 3.

During my time as a graduate student, I have been fortunate to work with researchers at a number of external labs during internships: Mihai Christodorescu and Rainer Sailer at IBM Research, Vinod Yegneswaran, Phillip Porras, and Hassen Saïdi at SRI International, and Ben Livshits, Nikhil Swamy, and Ben Zorn at Microsoft Research. Each internship refined my sense of "good research", and exposed me to problems and techniques that eventually allowed me to formulate and develop the work in this dissertation.

Finally, none of this would have been possible without the support and enthusiasm of my family and friends. Without them, I could not have made it through so many years of graduate school. I thank my parents

and sister for their steadfast belief in me, and their willingness to help at a moment's notice. To my wife Emily, thank you for being there to give me comfort, support, and the right perspective.

The research in this dissertation was generously supported by a Microsoft Research Graduate Fellowship; by the National Science Foundation under grant CCF-0524051; by the National Library of Medicine under grant R01LM011028; and by DARPA and AFRL under contract FA8650-10-C-7088.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Some of the most surprising and invasive privacy threats to materialize in recent years are brought about by so-called *inference attacks*: successful attempts to learn sensitive information by leveraging public data such as social network updates, published research articles, and web APIs. Unlike more traditional types of software vulnerabilities, effective mitigation strategies for inference attacks are not well-understood. This dissertation begins to address the problem of adversarial inference by developing tools and techniques that allow application developers to ensure that their systems make safe and responsible use of sensitive data. We begin by studying the tension between *privacy* and *utility* inherent to many applications. We present a new type of inference attack, called *Model Inversion*, on applications that use machine learning models. We show how to use model inversion constructively as part of an end-to-end analysis that evaluates the suitability of a potential countermeasure for a specific application. We then illustrate this approach using an in-depth case study of *pharmacogenomic warfarin dosing*, which is a well-studied application in personalized medicine. Our analysis concludes that one popular type of defense, called *differential privacy*, is not suited to this application, and highlights the need for new defenses. In the second part of the dissertation, we present a formal approach for reasoning about privacy guarantees in software, and give a new type of logic that allows one to reason explicitly about *adversarial uncertainty*. We then show how to construct decision procedures for this logic that developers can use to ensure that their software correctly enforces privacy protections. Through a better understanding of how proposed defenses impact real applications, and by providing tools that help developers implement rigorous defenses, we can begin to proactively identify potential inference attacks and take steps to ensure that they will not surface in practice.

1 INTRODUCTION

The trend towards data-driven applications has changed the way that people think about security and privacy. End-user applications, increasingly developed for data-rich mobile and web platforms, routinely collect users' geographic location, communication patterns, browsing history, media consumption habits, shopping tendencies, and other seemingly mundane day-to-day information in an effort to gain insight about *who* the user is. This insight holds value, as it can be leveraged in many ways to generate revenue—more relevant advertisements, differentiated pricing, and personalized experiences are just a few of the ways in which detailed knowledge of an individual is transformed into a lucrative asset.

This trend has made its way past end-user applications, and now plays a role in the way that many organizations make decisions. The abundance of data reflecting the tendencies of large populations, combined with powerful algorithms and inexpensive computing resources, enables statistics-driven decision-making in domains where this was previously not feasible or effective. The financial industry has used information about peoples' behavior to evaluate "credit worthiness" for many years, and has been quick to enhance its capabilities by incorporating new data sources available in large part because of peoples' everyday use of technology (Armour). The aggregation of medical histories with genetic and phenotypic information about individual patients allows medical researchers to develop better treatments by analyzing statistical trends among patient populations, giving rise to the field of personalized medicine (Wang et al., 2011). The goal in either domain is to use data to make better decisions.

The common thread among all of these applications is *inference*. Rather than having the user explicitly state her preferences, or hiring experts to formulate decision-making guidelines at the risk of missing subtle patterns, these applications use statistical algorithms to *infer* the relevant facts

from data. This approach yields excellent results as data proliferates, but in some cases it poses a risk to the individuals mentioned by the data. Although the confidentiality of medical records is protected by federal law (United States Congress, 1996), nothing prevents an analyst from using someone's web history to infer a "Diabetes Interest" or "Cholestoral Focus" (Ramirez et al., 2014). Similarly, just as a social network can infer one's taste in music to display relevant advertisements, it might also be able to infer their religion, race, political affiliation, or sexual orientation (Kosinski et al., 2013). Such personal information can be used for discriminatory purposes (Acquisti and Fong, 2012), or worse (U.N. Office of the High Commiss. for Human Rights, 2012). Thus, in the same way that applications use inference to learn helpful facts, others can use it to learn damaging or confidential information about a person—even though they did not explicitly disclose it. An unauthorized attempt to learn such information is called an *inference attack*.

In light of these attacks, we are compelled to weigh the benefits of using an application against its potential to make sensitive information vulnerable to an inference attack: *are the benefits provided by this application worth the risk of disclosing the information needed to run it?* As with any other type of vulnerability, it is the developer's responsibility to build applications that mitigate this risk to the greatest extent possible, e.g., by managing users' data safely and transparently, so that they know precisely how their data is handled and to whom it is disclosed. However, developers currently lack the tools needed to tell with any certainty whether their use of data introduces an inference vulnerability. Moreover, even if a well-intentioned developer manages to devise a good *theoretical* strategy for protecting sensitive data, the important task of ensuring that she has implemented it correctly requires techniques beyond the state-of-the-art in software verification and testing. In short, approaches for defending against these attacks lag behind those available for other types of vulnerabilities.

This dissertation begins to address the problem of adversarial inference by developing tools and techniques that allow application developers and data analysts to ensure that their systems make safe and responsible use of sensitive data. First, we examine the tension between data *privacy* and *utility*—the useful functionality that an application extracts from the data—and describe techniques that illustrate the inherent tension between these concerns as they arise in a specific application. This gives developers and analysts the ability to reason about the likelihood of inference vulnerabilities in an application, and to understand how well a proposed defense preserves the application's intended properties. We then move on to formal approaches for both specifying and reasoning about privacy guarantees in software. This work provides a new type of logic that equates adversarial uncertainty with a set of *constrained counting problems*, and demonstrates how to construct decision procedures for instances of the logic. Developers can use these procedures together with powerful software verification techniques to rigorously check their implementation for compliance with a set of closely-tailored privacy guarantees.

Understanding the Tension Between Privacy and Utility

Applications that use large amounts of data to drive their functionality typically rely on supervised machine learning (ML) to extract utility from the data. ML refers to a class of statistical techniques that attempt to extract general, population-wide trends from specific examples, and in so doing they accomplish two key efficiencies: i) salient patterns in the data are made explicit in a *model* that applications can efficiently query for predictions on new data as it arrives, and ii) parts of the data not relevant to these patterns are discarded, allowing a compressed representation of the parts that matter for the application. For example, an application that needs to recognize objects in digital images can use ML techniques on a large set of labeled image data to train a neural-network model (Taigman

Figure 1.1: The left image was recovered given only the classifier label corresponding to the individual and access to a softmax facial recognition model. The right image was one of the ten pictures of the subject's face used to train the model.

et al., 2014); this representation of the data is easily stored, transmitted, and evaluted on future unlabeled images.

Because ML models tend to abstract the data on which they are trained, they may not seem to pose an immediate threat to its confidentiality. After all, the models are meant to capture general properties of the distribution that generated the training data, and *not* the individual samples in the data itself—to do so would risk overfitting. For this reason, analysts often publish models (i.e., as part of an application) that were trained on sensitive data under the assumption that doing so is safe. However, this assumption is not always correct, and in some cases it is possible to reconstruct a close approximation to parts of the original training data after seeing a model. Looking at Figure 1.1, the image on the left was generated by analyzing the structure of a simple facial-recognition model to infer an image of one of the training subjects, and the right image is a training sample depicting that subject. This inference was generated using a technique called *model inversion* (MI), which we introduce in Chapter 3. MI is a technique for performing inference attacks on machine-learning models, but developers can also use it to estimate the likelihood of inference vulnerabilities in

their applications.

Researchers are aware of the potential privacy issues related to publishing ML models, so many have proposed *privacy-preserving* methods for learning them (Friedman and Schuster, 2010; LeFevre et al., 2005; Lindell and Pinkas, 2000; Rubinstein et al., 2012). These techniques aim to produce useful models that nonetheless can be published without leaking sensitive information about the training data, and vary considerably on the specific notion of privacy that they provide. Currently, the most popular such class of privacy-preserving learning techniques provide a guarantee known as *differential privacy* (DP) (Dwork, 2006), which is a general-purpose notion that is designed to obscure an individual's contribution to the data. In the context of ML, it stipulates that *anything that can be inferred from the model should be nearly as inferrable if any single individual's data is removed from the training set*.

However, these techniques are not yet prominent in practice. One difficulty in using them lies in the disconnect between the formal notion of privacy guaranteed by the technique, and the concrete privacy requirements of the application. Using DP as an example, the formal definition (see Chapter 2 Section 2.1 for a more precise version of this definition) stipulates that:

> *For any "neighboring" data sets* X *and* X' *differing in exactly one row, a randomized algorithm* A *is* ϵ*-differentially private if and only if,*
>
> $$\Pr[A(X) = r] \leqslant e^{\epsilon} \Pr[A(X') = r]$$
>
> *for all outputs* r*. The probabilities are taken over the randomness of* A*.*

This amounts to a guarantee that is both *approximate* and *relative* to a set of hypothetical "worlds", represented by inputs X and X', where each individual mentioned in the data set either disclosed or withheld her data—any inference that is possible from a model trained on X will be approximately as likely as an inference from a model trained on X'. This

guarantee is difficult to interpret in the context of a specific application when one lacks the tools necessary to determine how likely a harmful inference is in *any* world. Furthermore, the extent to which the guarantee is approximate is parameterized on ϵ, called the *privacy budget*. Selecting a budget that satsfies an application's privacy requirements can be a difficult judgement call (Lee and Clifton, 2011). Thus, lacking further guidance, it is often difficult to tell whether these techniques satisfy a given set of concrete privacy needs.

Another difficulty follows from the way that privacy-preserving learning algorithms typically achieve their guarantees: by *perturbation*, or adding random noise to a result in order to obscure an adversary's view of the original data. Again taking DP as an example, one common way to make an algorithm differentially-private is to add Laplace-distributed noise to its output (Dwork et al., 2006). The variance of this noise depends on the privacy budget ϵ, as greater variance ("more noise") is needed for budgets that guarantee stronger privacy. While the effect of ϵ on the resulting level of privacy is explicit in the definition of DP (ignoring the issues discussed above), the choice of ϵ also affects utility. Unfortuantely, this relationship is not explicit in the definition, and the impact of noise on components that use a model's output is context- and application-specific so most DP techniques do not attempt to characterize it. This leaves developers with a one-sided guarantee, and an unclear picture of how well their application will function if it uses such a technique.

This uncertainty on both fronts—privacy and utility—poses a major challenge for those who must make well-informed decisions about how to develop applications that are less vulnerable to inference attacks. These opposing concerns require a careful balance, and the steps needed to realize it are often obscured by the gap between general-purpose theoretical guarantees and an application's practical requirements. Chapter 3 describes an *in situ* approach for evaluating countermeasures with respect to

how well they achieve this balance. The crux of this method is that it relies on application-specific measures of privacy and utility, defined so that the effect of a countermeasure and its parameters on each concern is easily discernable. This approach gives developers a way to determine whether a countermeasure suits their application, and if so, the parameters under which it succeeds.

Chapter 3 goes on to illustrate this approach using an in-depth case study of personalized Warfarin dosing (International Warfarin Pharmacogenetic Consortium, 2009). This application uses patient histories, including genetic information, to build differentially-private ML models that suggest good initial doses of a dangerous medication. Here the tension between privacy and utility is stark: while patients' genetic information must be kept private, it is also needed to predict good doses, so the models need to encode some information about it—the question becomes whether that information can be misused to violate genomic privacy. The study uses model inversion to measure how well each trained model protects genetic information in the training data from inference, and detailed clinical trial simulations to measure the effectiveness of predicted doses. For this application, *effectiveness* is synonymous with the *avoidance of adverse health events*, so our simulations estimate the resulting propensity for stroke, two different types of hemorrhage, and mortality. As ϵ varies, these measures depict a clear trend in the tradeoff between privacy and utility: when ϵ is configured in a way that achieves good genomic privacy, the risk of adverse events including mortality is significantly higher than it is when patients are treated using the private-by-default non-genomic dosing regimen. This demonstrates that the chosen countermeasure is not an appropriate choice for any privacy budget.

```
1  def private_average(numbers,d,epsilon,seed):
2    avg = 0
3    for li in numbers:
4      assert li == 0 or li == 1
5      # scale by 10^d for fixed-point representation
6      avg += int(pow(10,d) * li / len(numbers))
7    # draw a laplace-distributed variate (scale=1, mean=0)
8    lap_var = int(pow(10,d)*-sgn(seed)*log(1-2*abs(seed)))
9    # transform its scale to the parameter needed for DP
10   noise = int((1/epsilon) * lap_var)
11   return noise + avg
```

Figure 1.2: Flawed implementation of DP using fixed-point numbers in an attempt to sidestep floating-point granularity issues (Mironov, 2012). DP is implemented on lines 8–11: a Laplace-distributed variate with scale parameter $\lambda = 1$ is drawn on line 8 using an inverse transform on the uniformly-distributed `seed`, then converted to scale $\lambda = 1/\epsilon$ on line 10, and finally added to the raw average on line 11.

A Logical Approach to Privacy-Correctness

Once a developer has identified an appropriate countermeasure, either from the theoretical literature or within a software library, she must correctly incorporate it into a larger application. As with modern cryptography, many privacy countermeasures have mathematically-sophisticated foundations that require careful translation into executable code. Even when implemented by domain experts, such countermeasures may exhibit subtle-but-critical bugs (Haeberlen et al., 2011). Moreover, information leaks in large software systems tend to crop up in unexpected places: exceptional control flow, global state, status messages, roundoff error, and sometimes even the decision to *not* publish a result can give an adversary enough information to mount an inference attack (Mironov, 2012; Haeberlen et al., 2011).

To get a sense of the sort of error that might arise when implementing strong privacy, consider the program in Figure 1.2. This program

computes a differentially-private average of a list of integers taking values in $\{0, 1\}$ (note the `assert` statement on line 4). To achieve DP, it adds Laplace-distributed noise with scale parameter $\lambda = 1/\epsilon$; this is the most commonly-used mechanism for making a deterministic real-valued function differentially-private (Dwork et al., 2006). As discovered by Mironov, DP cannot be directly implemented using floating-point numbers due to rounding issues (Mironov, 2012). Following a suggestion made in Mironov's manuscript, this implementation attempts to mitigate the problem using fixed-point numbers instead[1]. However, notice that if $\epsilon = 0.1$, then the variable `noise` computed on line 11 will be an integer multiple of $1/\epsilon = 10$. When `noise` is added to the final average on line 11, the least-significant digit of the result will thus be predictable. For example, if the input array `numbers` consists entirely of 0's then the last digit of the output will always be a 0, whereas on a neighboring array containing a single non-zero element, the last digit of output will always be non-zero. Thus, `private_average` does not satisfy DP despite its straightforward use of a well-known mechanism.

The sort of error exhibited in Figure 1.2 is easy to unwittingly introduce when trying to implement privacy. Traditional software testing approaches are unlikely to catch such a bug, as it only manifests on a small number of inputs, and to make matters worse, the program's outputs are random. *Rather, this problem calls for formal verification techniques that are able to cope both with modern privacy frameworks and the probabilistic nature of the programs that implement them.* Unfortunately, current verification approaches work by building program *abstractions* in one or more first-order theories that cannot account for the notions of probabilistic uncertainty required in this setting.

This dissertation describes a first-order theory that can accomodate these needs by incorporating *parametric model counting terms*. This ap-

[1] A fixed-point representation of a rational number q uses an integer n and a scaling factor d to represent a fixed degree of decimal precision, i.e., $q = n10^{-d}$.

proach equates uncertainty with the cardinality of sets defined by constraints from a *background theory*, and uses model-counting techniques in concert with automated deduction to reason about these cardinalities. For example, the program in Figure 1.2 requires reasoning about an adversary's uncertainty of the input `numbers`. Assuming a background theory capable of expressing the operations in this program, we can write a formula $\phi(n, d, \epsilon, s, r)$ that is true whenever $\texttt{private\_average}(n, d, \epsilon, s) = r$. We would then represent the probability expressions in the definition of DP as counting operations, i.e.,

$$\mathsf{count}(s, \phi(n, d, \epsilon, s, r)) \propto \Pr[\texttt{private\_average}(n, d, \epsilon, s) = r]$$

when n, d, and ϵ are fixed and s is drawn uniformly for the probability on the right. Notice that this count is *parametric* on n, d, ϵ and r: they are treated symbolically as fixed unknowns that affect the value of the counting term. The ability to reason in the presence of symbolic unknowns is necessary for verification, and distinguishes this approach from previous work that applies counting to related problems in information flow (Backes et al., 2009; Köpf et al., 2012; Klebanov, 2012; Klebanov et al., 2013; Heusser and Malacaria, 2010).

Chapter 4 defines this counting theory in a way that is agnostic to the background theory over which counting operators function, and describes an *abstract decision procedure* that solves the satisfiability problem over its formulas. The procedure is based on conflict-driven clause learning (Bayardo and Schrag, 1997), and incorporates background theory-specific reasoning with a small number of modular, well-defined interfaces. We then describe an instantiation of the procedure for counting over quantifier-free linear-integer arithmetic, and show how to use this for *bounded software model checking* to verify a range of privacy properties in applications developed for a cryptographic compiler.

Structure of the Dissertation

The remaining chapters of this dissertation are organized as follows. Chapter 2 presents background material supporting our main technical contributions in four sections: Section 2.1 covers differential privacy, Section 2.2 covers first-order theories and decision procedures, Section 2.3 covers bounded software model checking, and Section 2.4 covers polyhedral lattice-point counting. Chapter 3 presents model inversion, and our approach for measuring the balance between privacy and utility given by countermeasures. Chapter 4 details our first-order theory of counting, its abstract decision procedure, and application to bounded model checking. Chapter 5 concludes the thesis, and briefly discusses future directions.

2 BACKGROUND

This chapter provides background on a number of topics that are used later in the dissertation. It is not intended to be read sequentially, but rather as-needed by readers who are less familiar with certain areas. Section 2.1 covers differential privacy, including basic definitions, some well-known mechanisms, and a discussion of its semantics, intended use, and motivation. This material is primarily relevant to Chapter 3, although it is also used lightly in Chapter 4. Section 2.2 covers satisfiability decision procedures, after introducing first-order logic. Section 2.3 covers bounded software model checking, and shows how the problem can be reduced to satisfiability. Section 2.4 covers the problem of counting parametric polyhedra, and introduces Barvinok's famous result that this problem is solvable in polynomial time when the dimension of the polyhedra are fixed. Sections 2.1 through 2.4 are relevant only to Chapter 4.

2.1 Differential Privacy

Differential privacy (DP), originally formulated by Cynthia Dwork (Dwork, 2006), is a popular framework in the research literature for achieving a particular kind of privacy when releasing aggregate information about a database. DP is a property of algorithms, rather than the data they consume or produce. It achieves privacy by requiring that algorithms randomize their outputs in such a way that small variations in their inputs can only have limited impact on their output. It is an information-theoretic guarantee, holding regardless of the computational resources of an adversary. Variants of differential privacy that are only secure against computationally-bounded adversaries, called *computational differential privacy*, have also been proposed (Dwork, 2008b), but they are not considered in this dissertation.

There are two common variants of differential privacy: *bounded* and *unbounded* (given in Definitions 2 and 3 below, respectively). Unbounded DP was the original definition formulated by Dwork in 2006, and the bounded variant was introduced by McSherry and Talwar in the following year. For convenience, we will sometimes omit the use of "bounded" and "unbounded", and unless stated otherwise, assume that the term "differential privacy" refers to unbounded DP.

We consider DP algorithms that represent queries over databases, and produce outputs from a set Sof query results. A database D is a sequence of rows $[(x_1, y_1), \ldots]$ from the set $\mathcal{D} = X_1 \times \cdots \times X_d \times Y$, where each row is a tuple comprised of *features* $(x_1, \ldots, x_d) \in X_1 \times \cdots \times X_m$ and a *response* $y \in Y$. Databases are drawn from a distribution $\mathbb{D}$. Let $\mathcal{D}_r$ be the space of all databases with r rows; then the space of all databases $\mathcal{D} = \bigcup_{r=0}^{\infty} \mathcal{D}_r$. Although the ordering of tuples does not usually matter, we assume a fixed arbitrary ordering for notational purposes. We write $D[i]$ to refer to the i*th* row of D, and given a non- repetitive sequence of n natural numbers $I = \{i_1, \ldots, i_n\}$, we write $D[I]$ to refer to the database comprised of $D[i_1], \ldots, D[i_n]$.

We first give a generalized definition of DP called *relational differential privacy* (Definition 1).

Definition 1. *(*Relational Differential Privacy*) A mechanism* A *achieves relational* ϵ*-differential privacy with respect to a symmetric relation* $N \subseteq \mathcal{D} \times \mathcal{D}$ *if for all pairs of databases* $(D_1, D_2) \in N$ *and all* $O \in S$*,*

$$\Pr[A(D_1) = O] \leqslant \exp(\epsilon) \times \Pr[A(D_2) = O]$$

The probabilities are measured over the randomness of A*.*

Definition 1 stipulates that the outputs produced by A are not likely to change much (i.e., by more than a factor of $\exp(\epsilon)$) between pairs of inputs that are related by N, which we call the *neighbor relation*. The parameter ϵ

is called the *privacy budget*, and setting its value is a policy choice made by the database owner. We can now define the two main variants of DP by specifying the appropriate relations.

Definition 2. *(*Bounded Differential Privacy *(McSherry and Talwar, 2007)) A mechanism* K *achieves bounded* ϵ*-differential privacy if for all pairs of databases* D_1, D_2 *where* D_1 *can be obtained from* D_2 *by changing the value of exactly one row, and for all* $S \in \text{Range}(K)$*,*

$$\Pr[K(D_1) = S] \leqslant \exp(\epsilon) \times \Pr[K(D_2) = S]$$

The probabilities are measured over the randomness of A*.*

Definition 2 corresponds to a neighbor relation N_b containing all pairs of same-length databases where all but one of their rows can be permuted to match each other. In other words, if $(D_1, D_2) \in N_b$, then D_1 can be obtained from D_2 by changing the value of exactly one row.

Definition 3. *(*Unbounded Differential Privacy *(Dwork, 2006)) A mechanism* K *achieves unbounded* ϵ*-differential privacy if for all databases* D_1, D_2 *where* D_1 *can be obtained from* D_2 *by adding or removing exactly one tuple, and for all* $S \in \text{Range}(K)$*,*

$$\Pr[K(D_1) = S] \leqslant \exp(\epsilon) \times \Pr[K(D_2) = S]$$

The probabilities are measured over the randomness of A*.*

Definition 3 corresponds to a neighbor relation N_u containing all pairs of databases D_1, D_2 whose length differs by one, and whose rows can be permuted to match after removing a single row from only one of the databases. In other words, D_1 can be obtained from D_2 by adding or removing one row. Notice that bounded DP is equivalent to unbounded DP after increasing ϵ by at most a factor of two, because a change in one row can be modeled by removing one row and replacing it with another.

Algorithms for Achieving Differential Privacy

The first published mechanism for achieving differential privacy, known as the *Laplace mechanism* (Dwork et al., 2006), was designed for real-valued queries. It works by adding Laplace-distributed noise to the query response, the variance of which is calibrated to the *global sensitivity* (Definition 4) of the query function f.

Definition 4. *(*Global Sensitivity *(Dwork et al., 2006)) For* $f : \mathcal{D} \mapsto \mathbb{R}^d$*, the global sensitivity of* f *is*

$$\Delta f = \max_{D_1, D_2} \|f(D_1) - f(D_2)\|_1$$

for all D_1, D_2 *where* D_1 *can be obtained from* D_2 *by adding or removing one row.*

For example, the global sensitivity of a query that counts the number of rows in D that match a predicate ϕ:

$$\lambda D. \sum_{i=1}^{|D|} \text{ite}(\phi(D[i]), 1, 0)$$

is the constant 1 because each row in D contributes a constant value (i.e., 0 or 1) to the aggregate result. The Laplace mechanism (Theorem 2.1) uses global sensitivity to calibrate the amount of noise needed to achieve DP.

Theorem 2.1. *(*Laplace Mechanism *(Dwork et al., 2006)) For* $A : \mathcal{D} \mapsto \mathbb{R}^d$ *and* $\mathbf{r} \in \mathbb{R}^d$ *a vector whose components are independently sampled from the zero-centered Laplace distribution with variance* $2\Delta f/\epsilon$*, e.g.,*

$$\text{pdf}(r) \propto \exp\left(-\frac{\epsilon|r|}{\Delta A}\right)$$

The function $\tilde{A}(\mathbf{x}) = A(\mathbf{x}) + \mathbf{r}$ *satisfies unbounded* ϵ*-differential privacy.*

Because sampling from the Laplace distribution is easy, the Laplace mechanism is a straightforward and efficient way to achieve differential privacy for real-valued functions whose sensitivity is easy to analyze, and gives accurate answers when ΔA is independent of $|D|$, the magnitude of $A(D)$ grows with $|D|$, and D is large.

In cases where A is not real-valued, the *exponential mechanism* (Theorem 2.2) is a more general approach for achieving DP. For a query function $A : \mathcal{D} \mapsto S$, the exponential mechanism requires a *utility function* U that assigns a score to any input-output pair in A and a *base measure* μ over S, and selects an output corresponding to $\mathbf{x}$ that approximately maximizes U. The idea is that U should be constructed to assign high scores to "quality" outputs of A for a given input. The base measure is commonly uniform.

Theorem 2.2. *(*Exponential Mechanism *(McSherry and Talwar, 2007)) For* $U : \mathcal{D} \times S \mapsto \mathbb{R}$*, and a base measure* μ *over* S*, the function*

$$\mathsf{ExpMech}(U, \mathbf{x}) \stackrel{\text{def}}{=} \begin{array}{c}\text{Choose } s \text{ with probability proportional to} \\ \exp\left(\frac{\epsilon U(\mathbf{x}, s)}{2\Delta U}\right) \times \mu(s)\end{array}$$

satisfies unbounded ϵ*-differential privacy.*

McSherry and Talwar point out that this mechanism can capture any differentially-private algorithm by taking $U(\mathbf{x}, s)$ to be the logarithm of the probability density of $A(\mathbf{x})$ at x. Indeed, Theorem 2.1 is captured (in one dimension) by taking $U(\mathbf{x}, s) = -2|A(\mathbf{x}) - s|$.

In Chapter 3 we will study two differentially-private algorithms in considerable depth—*private projected histograms* (Vinterbo, 2012) and the *functional mechanism* (Zhang et al., 2012)—that are more complex in their operation and analysis. Nonetheless, they both rely in various ways on the clever application of Theorems 2.1 and 2.2 to achieve their goals.

Semantics and Motivation

Definitions 2 and 3 both require that the output of the algorithm remain stable as changes are made to individual rows in the input database. As described in (Dwork, 2008a), this is meant to provide strong privacy for the individuals whose data constitutes each row of the database:

> *A mechanism $\mathcal{K}$ satisfying this definition addresses concerns that any participant might have about the leakage of her personal information: even if the participant removed her data from the data set, no outputs (and thus consequences of outputs) would become significantly more or less likely. For example, if the database were to be consulted by an insurance provider before deciding whether or not to insure a given individual, then the presence or absence of that individual's data in the database will not significantly affect her chance of receiving coverage. Differential privacy is therefore an* ad omnia *guarantee.*

However, DP does not say anything *absolute* about what an adversary can infer of the database or individuals mentioned in it—it is a completely *relative* definition. This is intentional; Dwork formulated DP in this way to sidestep an impossibility result stating that there is no general-purpose absolute notion of private data release (Dwork, 2006).

The impossible privacy notion that Dwork wished to avoid is concisely summarized by Dalenius: *access to a database should not enable one to learn anything about an individual that could not be learned without access* (Dalenius, 1977). To demonstrate that this type of absolute privacy is unattainable, Dwork formalized a "game" that encompasses private data release. The entities used in this game are given in Table 2.1, and the game itself is given in Figure 2.1.

Absolute privacy is equated with the adversary (or simulator) being unable to win the game: the adversary wins if, when given its inputs,

Entity	*Description*
$\mathbb{D}$	Distribution over databases.
D	Database drawn from $\mathbb{D}$.
A	*Release mechanism* taking $\mathbb{D}$ and D as input, and producing a fact c about the database as output. c is available to the adversary.
C	*Breach decider*, a Turing machine taking $\mathbb{D}$, D, and a *breach string* s as input. Outputs 1 if s constitutes a privacy breach on $\mathbb{D}$ and D, and 0 otherwise. C must always halt.
X	*Auxiliary information generator*, a Turing machine taking $\mathbb{D}$ and D as input, and producing an *auxiliary information string* x as output. x is made available to the adversary.
Adv	*Adversary*, a communicating Turing machine reading $\mathbb{D}$, $\mathsf{A}(\mathbb{D}, \mathsf{D})$, and $\mathsf{X}(\mathbb{D}, \mathsf{D})$, and producing an (attempted) breach string as output.
Sim	*Simulator*, a communicating Turing machine reading $\mathbb{D}$, and $\mathsf{X}(\mathbb{D}, \mathsf{D})$, and producing an (attempted) breach string as output.
w	*Utility vector* of binary values. Intuitively, w contains answers to questions about D most of which should be learnable from $\mathsf{A}(\mathsf{D})$.

Table 2.1: Entities used in Dwork's privacy game

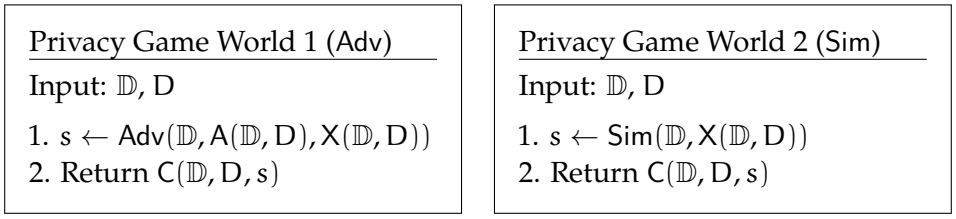


Figure 2.1: Dwork's privacy game. In the first world, the adversary is given the output of the release mechanism A and the auxiliary information generated by X. In the second world, the simulated adversary is only given the auxiliary information. In either world, the agent (Adv or Sim) *wins* when the game returns 1. This condition represents a successful privacy breach.

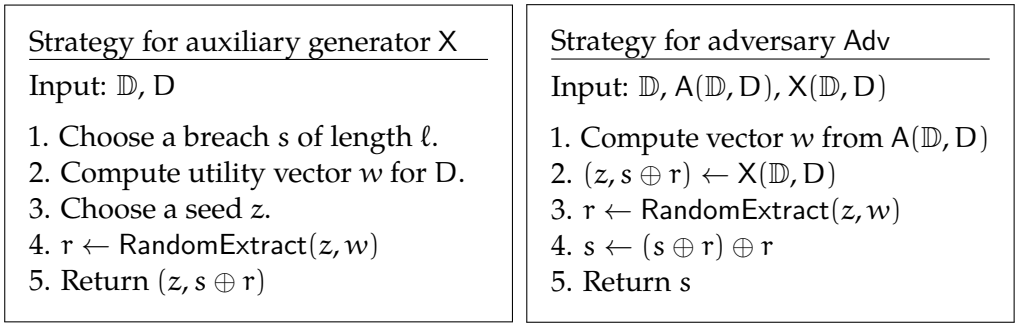


Figure 2.2: Strategy for auxiliary information generator X and adversary Adv in Dwork's privacy game. Both strategies rely on a *strong randomness extractor* RandomExtract to generate a one-time pad from the utility vector w, which is later used to encrypt the breach. These definitions assume that all of w is computable from the output of the release mechansim $\mathsf{A}(\mathbb{D}, d)$, but Dwork's full proof also contains a strategy that assumes the adversary can only compute a portion of w (Dwork, 2006).

it produces a successful breach string s^*, i.e., $\mathsf{C}(\mathbb{D}, \mathsf{D}, s^*) = 1$. Thus, a release mechanism A achieves absolute privacy if it produces outputs that no definition of Adv is able to leverage to its advantage along with some information given by an auxiliary information generator X. Dwork showed that for any release mechanism A and definition of privacy breach C, there exists an auxiliary information generator X and adversary Adv who wins this game with sufficiently greater odds than any other adversary *simulator* who is not given access to A. This is stated precisely in Theorem 2.3, and the full proof is given in (Dwork, 2006).

Theorem 2.3. *(*Impossibility of Absolute Privacy *(Dwork, 2006)) Fix any computable privacy mechanism* A *and breach decider* C *(see Table 2.1 for details on these entities). Then there is an auxiliary information generator* X *and an adversary* Adv *such that for all distributions* $\mathbb{D}$ *satisfying certain assumptions*[1]

[1]See (Dwork, 2006) for details; these assumptions most importantly require that w be difficult to guess for a given $(\mathbb{D}, \mathsf{D})$ without seeing the output of $\mathsf{A}(\mathbb{D}, \mathsf{D})$.

and for all simulators Sim,

$$\Pr[\mathsf{Adv}(\mathbb{D}, \mathsf{A}(\mathbb{D}, \mathsf{D}), \mathsf{X}(\mathbb{D}, \mathsf{D})) \text{ wins}] - \Pr[\mathsf{Sim}(\mathbb{D}, \mathsf{X}(\mathbb{D}, \mathsf{D})) \text{ wins}] \geqslant \Delta$$

where Δ *is a suitably chosen (large) constant. The probabilities are measured over the random choice of* D *according to* $\mathbb{D}$ *and the coin flips of* A, X, Adv, *and* Sim.

The strategies used by X and Adv forming the crux of the proof of Theorem 2.3 are given in Figure 2.2. These strategies rely on the ability to extract a one-time pad from the utility vector, which is used by the auxiliary information generator to encrypt a breach string. This is always possible using strong randomness extractors (Dodis et al., 2008) as long as the distribution of utility vectors has sufficiently high min-entropy. Then, on learning the output of A, the adversary can reconstruct the utility vector and proceed to extract the same one-time pad to decrypt the breach given as auxiliary information. Thus, one crucial ingredient in this impossibility result is the notion of utility: the output of A should be *useful* to observers by conveying information that is difficult to learn otherwise.

Another important ingredient is the freedom under which the auxiliary information generator and breach decider are defined—they can correspond to any computable function over $\mathbb{D}$, D, and s. Notably, an auxiliary information generator can be "wired" with information about an individual not even in the database, and subsequently produce breach strings that affect that individual. As described by Dwork (Dwork, 2008b), this is analogous to a situation where a person's height is considered sensitive (and thus constitutes a breach), and having the auxiliary information generator produce the string "Terry Gross is two inches shorter than the average Lithuanian woman." Then a release mechanism that outputs the average height from a database of Lithuanian women violates Terry Gross' privacy, despite her absence from the database.

Dwork proposed DP as one way to sidestep these issues by working to-

wards a relative, rather than an absolute, privacy goal: *a given breach will be almost as likely regardless of whether any individual is mentioned in the database*. She formalized this goal (Definition 3), provided useful algorithms for achieving it, and constructive research on the topic has subsequently flourished. Note that there may be other ways to achieve meaningful definitions of privacy despite Theorem 2.3, e.g., by placing restrictions on C or X, but an exploration of this topic remains future work.

Despite its popularity, the relative nature of differential privacy is often misunderstood by those who wish to apply it, as pointed out by Dwork and others (Dwork et al., 2011). Kifer and Machanavajjhala (Kifer and Machanavajjhala, 2011) addressed several common misconceptions, and showed that under certain conditions, it fails to achieve a privacy goal widely assumed to be synonymous with DP: *nearly all evidence of an individual's participation should be removed.* They argued that in order to achieve this goal, one must carefully consider how the private data was generated. Li *et al.* later gave more specific semantics to DP, under a framework called *membership privacy* (Li et al., 2013): *an adversary should not have a significant advantage in determining whether an individual is present in the dataset or not*.

One common misconception is that DP protects membership privacy regardless of an adversary's background knowledge and the properties of the data. To see why this is false, consider the following example which is slightly modified from one appearing in (Kifer and Machanavajjhala, 2011). Consider a $\mathbb{D}$ whose support only contains databases such that when Alice's row appears then there are 20000 cancer patients in the data, and when Alice's row does not appear, there are < 10000 cancer patients in the data. A differentially-private release of the number of cancer patients will contain noise with variance 200 (at $\epsilon = 0.1$), so an adversary who knows this property of $\mathbb{D}$ can divide and round the DP answer to learn, with high probability, whether Alice was present in the database. To

decrease the adversary's odds of success, the variance would need to be large enough to mask 10000 rows, which would preclude utility.

Others have studied the degree to which differential privacy leaks various types of information. Cormode showed that if one is allowed to pose certain queries relating sensitive attributes to quasi-identifiers, it is possible to build a differentially-private Naive Bayes classifier that accurately predicts the sensitive attribute (Cormode, 2011). Lee and Clifton (Lee and Clifton, 2011) recognize the problem of setting ϵ and its relationship to the *relative* nature of differential privacy, and later (Lee and Clifton, 2012) propose an alternative parametization of differential privacy in terms of the *probability that an individual contributes to the resulting model.* While this may make the privacy guarantee easier for non-specialists to understand, its close relationship to the standard definition suggests that it may not be effective at mitigating the types of disclosures documented later in this thesis.

2.2 Satisfiability for First-Order Theories

We now turn to discussing the satisfiability decision problem on formulas in first-order theories, and a procedure for solving them. Decision procedures are widely used in verification, program analysis, constraint solving, planning, artificial intelligence, and many other fields. The decision problem that we will focus on is *satisfiability of formulas in first-order theories*, sometimes called "Satisfiability Modulo Theories" (SMT). Informally, given a formula ϕ over a set of variables $x_1, \ldots, x_n$, the SMT problem asks, "Does there exist a valuation of $x_1, \ldots, x_n$ that satisfies ϕ?"

We will begin by discussing the syntax and semantics of first-order logic, define precisely what is meant by a *first-order theory*, and then move on to our discussion of SMT decision procedures. The most successful SMT algorithms are based on the classic DPLL procedure for solving the propositional satisfiability (SAT) problem, and are best understood as encoding-based reductions to DPLL. Thus, our discussion of decision procedures will first cover the original (propositional) DPLL algorithm before moving to the modern first-order extension, called DPLL(T).

First-Order Logic and Theories

First-order logic is the standard system for formalizing and reasoning about mathematical objects such as numbers and sets. It is distinguished from propositional logic by its use of quantifiers over objects from a chosen domain. The two main components of first-order logic are its syntax, which comprises the family of *first-order languages*, and its semantics, which are given by a *structure* that associates relations, functions, and constants to a given first-order language. A theory corresponds to the set of sentences from a given first-order language that *model*, or hold true in, the corresponding signature. In the following, we follow the conventions used in (Hinman, 2005) to make these statements precise.

Syntax of First-Order Languages

A first-order language L is specified by a tuple $(\mathsf{R}, \mathsf{F}, \mathsf{C}, \nu, \mathsf{V}, =, \neg, \vee, \exists)$ (see Table 2.2 for a description of each component). The members of R, F, and C are called *non-logical symbols*, and everything else are *logical symbols*. The rank function ν assigns a rank (or *arity*) to each relation and function symbol in L. Notice that this definition omits the connective $\wedge$ and the quantifier $\forall$; these can be defined in terms of $\neg, \vee$, and $\exists$.

The syntax of a first-order language is usually defined hierarchically, as shown in Figure 2.3. An *atomic term* (or simply *atom*) is an expression formed by a single constant symbol from C or variable from V. *Terms* correspond to the closure of atoms and applications of function symbols from F. *Atomic formulas* correspond to applications of the equality relation $=$ to two terms, as well as applications of relation symbols from R. *Formulas* consist of the closure of all atomic formulas and applications of the connectives $\neg, \vee$ as well as the existential quantifier. Finally, *sentences* are formulas containing no *free variables*, i.e., variables unbound by an existential quantifier.

Example 2.4. *If we wanted to study the theory of arithmetic, we could define the entities of a first-order language* L_Ω *as follows:*

$$\mathsf{R}_\Omega = \{\dot{<}\} \quad \mathsf{F}_\Omega = \{\dot{+}, \dot{\times}, \mathsf{succ}\} \quad \mathsf{C}_\Omega = \{\dot{0}\}$$

We use the "dot notation" on familiar non-logical symbols that have common meaning in other contexts so that we can easily distinguish between uses that appear within the language. Here succ *is the successor function. Then* $\nu(\dot{<}) = \nu(\dot{+}) = \nu(\times) = 2$ *and* $\nu(\mathsf{succ}) = 1$*. Then examples of atoms would be* $\dot{0}$*,* x*, and* z *(note that we did not define any restrictions on* V*, as it should always be clear from the context when we refer to a variable). Some terms defined by* L_Ω *are* $\dot{0}$*,* $\mathsf{succ}(\dot{0})$*, and* $\mathsf{succ}(\mathsf{succ}(\dot{0}))$*.* $\mathsf{succ}(\dot{0})\dot{<}\mathsf{succ}(\mathsf{succ}(\dot{0}))$ *is an atomic formula, and* $\exists x.\mathsf{succ}(\mathsf{succ}(\dot{0})) = \mathsf{succ}(x\dot{+}\dot{0})$ *is a sentence.*

Entity	*Description*
R	A set of *relation* symbols
F	A set of *function* symbols
C	A set of *constant* symbols
ν	A *rank function* defined over $R \cup F \mapsto \mathbb{N}$
$=, \neg, \vee, \exists$	Logical symbols with the usual meaning
V	A denumerable set of *variables*

Table 2.2: Components of a first-order language.

atom	$\stackrel{\text{def}}{=}$	$c \in C$ $\mid x \in V$
term	$\stackrel{\text{def}}{=}$	*atom* $\mid f(term_1, \ldots, term_{\nu(f)})$ for $f \in F$
atomic formula	$\stackrel{\text{def}}{=}$	$term_1 = term_2$ $\mid r(term_1, \ldots, term_{\nu(r)})$ for $r \in R$
formula	$\stackrel{\text{def}}{=}$	*atomic formula* $\mid \neg$ *formula* $\mid$ *formula*$_1 \vee$ *formula*$_2$ $\mid \exists$ *formula*

Figure 2.3: Syntax for the first-order language $L = (R, F, C, \nu, V, =, \neg, \vee, \exists)$

Semantics of First-Order Logic

First-order languages are given meaning by assigning relations, functions, and constants to the syntactic elements R, F, and C. The constants are drawn from a chosen domain, over which the relations and functions range. These assignments are collected in an L-*structure*, described in Definition 5.

Definition 5. *(*L-structure U*) For a first-order language* $L = (R, F, C, \nu, V, =, \neg, \vee, \exists)$*, an* L*-structure* U *consists of the following elements:*

- *A non-empy set* A *called the* domain *or* universe *of* U*.*
- *For each* $p \in R$*, a relation* $p_U \subseteq A^{\nu(p)}$*.*

- *For each* $f \in F$, *a function* $f_U : A^{\nu(f)} \mapsto A$.
- *For each* $c \in C$, *an element* $c_U \in U$. *These are sometimes called the* distinguished elements *of* U.

Given an L-structure U, we can give denotational semantics to formulas in a first-order language L by attaching values in $\{\mathsf{True}, \mathsf{False}\}$ to formulas. This requires interpreting the variables that occur in the formula with elements from the domain of U, by way of an *assignment* α (Definition 6). We define the denotation of a formula ϕ under an assignment α using the notation $\phi[\![\alpha]\!]$. This semantics is given in Figure 2.4.

Definition 6. *(*U-assignment α and extension.*) For an* L*-structure* U*, a* U-assignment *is a function* $\alpha : V \mapsto A$. *An* extension *to* α *on* $x \in V$*, denoted by* $\alpha[x \mapsto a]$*, is an assignment that agrees on* α *for all elements of* V *except* x*, where it takes value* a.

If $\phi[\![\alpha]\!] = \mathsf{True}$ (False), then we say that ϕ *evaluates to* True *(*False*) under* α. *Satisfiability* is the notion that, given a formula ϕ, there exists a structure U and assignment that causes ϕ to evaluate to True. *Validity* is the dual of satisfiability, so a formula ϕ is valid iff $\neg\phi$ is unsatisfiable. *Logical consequence* in this setting corresponds to the idea that one formula ϕ follows from others in a set Γ under all L-structures. *Consistency* of a set of formulas Γ says that there should exist at least one L-structure and assignment that makes each formula in the set evaluate to True. These notions are made precise in Definition 7.

Definition 7. (Satisfiability, logical consequence, consistency) *For any* L*-formula* ϕ *and set of* L*-formulas* Γ*,*

- ϕ *is* satisfiable *in an* L*-stucture* U *iff for some* U*-assignment* α*,* $\phi[\![\alpha]\!] =$ True.

For a variable $x \in V$, $x[\![\alpha]\!] \stackrel{\text{def}}{=} \alpha(x)$.

For a constant $c \in C$, $c[\![\alpha]\!] \stackrel{\text{def}}{=} c_U$.

For a function $f \in F$ *and terms* $t_1, \ldots, t_{\nu(f)}$,

$$f(t_1, \ldots, t_{\nu(f)})[\![\alpha]\!] \stackrel{\text{def}}{=} f_U(t_1[\![\alpha]\!], \ldots, t_{\nu(f)}[\![\alpha]\!])$$

For a relation $p \in R$ *and terms* $t_1, \ldots, t_{\nu(p)}$,

$$p(t_1, \ldots, t_{\nu(p)})[\![\alpha]\!] \stackrel{\text{def}}{=} (t_1[\![\alpha]\!], \ldots, t_{\nu(p)}[\![\alpha]\!]) \in p_U$$

For an L*-formula* ϕ,

$$(\neg\phi)[\![\alpha]\!] \stackrel{\text{def}}{=} \neg(\phi[\![\alpha]\!])$$

For L*-formulas* ϕ *and* ψ,

$$(\phi \vee \psi)[\![\alpha]\!] \stackrel{\text{def}}{=} \phi[\![\alpha]\!] \vee \psi[\![\alpha]\!]$$

For an L*-formula* ϕ *and variable* x,

$$(\exists x.\phi)[\![\alpha]\!] \stackrel{\text{def}}{=} \text{There exists } a \in A \text{ such that } \phi[\![\alpha[x \mapsto a]]\!]$$

Figure 2.4: Semantics of the language L under U and U-assignment α

- ϕ *is a* logical consequence *of (or* entailed *by)* Γ *(written* $\Gamma \models \phi$*) iff for all* L*-structures* U*,* U*-assignments* α*, and all* $\psi \in \Gamma$, $\psi[\![\alpha]\!] = \mathsf{True}$ *implies that* $\phi[\![\alpha]\!] = \mathsf{True}$.
- Γ *is* consistent *iff for some* L*-structure* U *and* U*-assignment* α*, then for all* $\psi \in \Gamma$, $\psi[\![\alpha]\!] = \mathsf{True}$.

Note that a sentence is satsfiable for a given structure U if and only if it evaluates to True under all assignments, which is identical to validity. We will often say that a sentence ϕ is *true in* U to mean that it is satisfiable. We will sometimes write $\models \phi$ as shorthand for $\varnothing \models \phi$.

First-Order Theories

A first-order theory is characterized by a set of *theorems* Γ, which is a set of sentences in some language L that is closed under logical consequence. A set of *axioms* is a subset of the theorems that *generates* the entire set, and is thus a characerization of the theory. Alternatively, a theory can be characterized by its set of *models*, or structures under which the theorems are true. These notions are made precise in Definition 8.

Definition 8. (Theory, axioms, models) *For any set* Γ *of* L*-sentences,*

- Γ *is an* L-theory *iff for all* L*-sentences* ϕ, $\Gamma \models \phi \implies \phi \in \Gamma$.
- *A set of* L*-sentences* Γ_{axiom} *are* axioms *of the* L*-theory* Γ *iff for all* $\phi \in \Gamma$, $\Gamma_{\text{axiom}} \models \phi$.
- *An* L*-structure* U *is a* model *of an* L*-theory* Γ *iff for all* $\phi \in \Gamma$, ϕ *is true in* U.

We refer to the set of models of a theory Γ by $\mathsf{Mod}(\Gamma)$. We refine the notions of satisfiability and entailment for a specific theory Γ by defining a formula ϕ to be Γ-satisfiable if ϕ is satisfiable for all $\mathsf{U} \in \mathsf{Mod}(\Gamma)$. ϕ is Γ-entailed by a set of sentences Γ', written $\Gamma' \models_\Gamma \phi$, if for all assignments α and $\psi \in \Gamma'$, $\psi[\![\alpha]\!] \implies \phi[\![\alpha]\!]$. The problem of determining whether an L-formula is Γ-satisfiable is known as the Γ-*satisfiability decision problem*, or colloquially as the *satisfiability modulo theories* (SMT) problem.

Because most interesting theories are undecidable for formulas containing quantifiers, we will usually want to restrict ourselves to a *quantifier-free fragment* of the theory in question. Simply put, we say that $\Gamma' \subseteq \Gamma$ is the quantifier-free fragment of the theory Γ if no $\phi \in \Gamma'$ contains a quantifier symbol.

Deciding Quantifier-Free First-Order Theories

A *satisfiability decision procedure* is a program that takes a formula in a given theory and returns Sat when the formula is satisfiable, Unsat when it is unsatisfiable, or $\perp$ if the formula cannot be decided. The latter can happen when the theory is undecidable, in which case the procedure makes a best-effort attempt at solving a given instance but sometimes fails. A decision procedure is *sound* if it returns Sat (respectively, Unsat) when the input formula is satisfiable (respectively, unsatisfiable). A decision procedure is *complete* if it always terminates, and when it does, returns Sat for a satisfiable input formula.

In the following, we describe the most prevalent method for solving SMT instances. As this approach is based on methods for solving satisfiability for propositional logic (Definition 9), we will fisrt make a brief digression to discuss satisfiability decision procedures for propositional logic.

DPLL and Propositional Satisfiability

Most modern SAT procedures are based on the *Davis-Putnam-Loveland-Logemann* (DPLL) framework (Davis and Putnam, 1960; Davis et al., 1962), Intuitively, this method works by making a decision about the value of a variable, *propagating* the decision through the rest of the formula, and backtracking when a *conflict* is detected—whenever a decision makes the formula evaluate to False. This process repeats until either all variables have been assigned without conflict (thus returning Sat), or a conflict is detected and no further backtracking is possible (thus returning Unsat). Nearly all high-performing procedures now also incorporate a type of learning that helps them avoid searching redundant parts of the solution space. Whenever the procedure encounters a conflict, it constructs a *lemma* that explains why certain related assignments would also lead to a conflict. This is called *conflict-driven clause learning* (CDCL).

The main algorithm is presented in Figure 2.7, and the subroutines it uses are summarized in Figure 2.6. We discuss the algorithm and its subroutines in detail below.

Definition 9. *(*Propositional logic*) Propositional logic corresponds to a first-order theory over the language* $\mathrm{L}_{\mathrm{prop}}$ *given by,*

$$\mathsf{F} = \mathsf{C} = \varnothing \quad \mathsf{R} = \{\dot{\mathsf{True}}, x_1, x_2, \ldots\}$$

All relations in R *are nullary.* $\dot{\mathsf{True}}$ *must always be true in any model of propositional logic, and the remaining relations correspond to Boolean variables. Conventionally, a propositional atomic formula or its negation is called a* literal*. Quantifiers are not used in propositional logic as they are unnecessary, i.e.,* $\exists x.\phi \equiv (\phi \wedge x) \vee (\phi \wedge \neg x)$*.*

Preliminaries. DPLL requires input formulas to be in *conjunctive normal form* (Definition 10). This is not an issue in practice, as formulas can be converted into CNF in linear time. Because their Boolean structure is regular, propositional formulas in CNF can be represented in as a set of clauses. For example, the CNF $(\neg x \vee y) \wedge (x \vee y \vee \neg z)$ can be represented by the set $\{\{\neg x, y\}, \{x, y, \neg z\}\}$. We will often use the symbols $l_1, l_2, \ldots$ to refer to literals in a CNF. Recall that a literal l can refer to a variable or its negation, so we can write l in place of $\neg x$. Given a propositional CNF ϕ, we use $\mathsf{Clauses}(\phi)$ to denote its corresponding clause set.

Definition 10. (Conjunctive Normal Form) *A propositional formula is in* conjunctive normal form *(CNF) if it is a conjunction of disjunctions of literals, i.e., it is in the form:*

$$\bigwedge_i \left(\bigvee_j l_{ij} \right)$$

Each inner disjunct is called a clause*, and a variable can appear at most once in each clause; any variable appearing more than once in a clause will either be*

redundant or make the clause trivially satisfiable. Similarly, a formula in the quantifier-free fragment of a first-order theory is in CNF if it is a conjunction of disjunctions of atomic formulas and their negations.

We describe DPLL as a procedure that takes two arguments. The first is a set F of clauses corresponding to $\mathsf{Clauses}(\phi)$ for a propositional formula ϕ, and the second a sequence M of literals called the *trail*. M is initially empty, and is used to denote a partial assignment to the variables in ϕ. Sometimes we will abuse notation and treat M as though it were a conjunction. So for a formula ϕ containing a single clause $x \vee \neg y$ where $l_1 \mapsto x, l_2 \mapsto \neg y$, the trail $[\neg l_1, \neg l_2]$ corresponds to the assignment $\{x \mapsto \mathsf{False}, y \mapsto \mathsf{True}\}$ and the conjunction $\neg x \wedge y$. M contains two types of literals: *decided* and *implied*. Decided literals correspond to splitting decisions explicitly made by the procedure in its search, and are annotated by the superscript ^{d} when written in the trail. Implied assignments correspond to assignments found during propagation or learned via a lemma, and are given no annotation. So if we annotate the trail from before $[\neg l_1^{d}, \neg l_2]$, then the first element corresponds to a decision made by the procedure to assign $x \mapsto \mathsf{False}$, and the second to the results of propagating that decision through ϕ.

A clause set F is *in conflict* under a trail M, written $\mathsf{Conflict}(F, M)$ if at least one clause in F evaluates to False under the assignments in M. A clause set F is *satisfied* under a trail M, written $\mathsf{Satisfied}(F, M)$, if all clauses in F evaluate to True under the assignments in M. Alternatively, $\mathsf{Satisfied}(F, M) = \mathsf{True}$ iff at least one literal from each clause in F appears positive in M.

Deciding literals. The most basic step taken by the procedure is to *decide*, or make an assignment to, a literal in F. The order in which literals are decided can have a drastic effect on the runtime of the procedure, and high-performing implementations typically use several heuristics to select this ordering. However, as this aspect does not affect correctness, we will not

go into further detail on the subject and simply assume that the procedure picks an arbitrary literal that remains unassigned in M. We model this procedure with the function $\mathsf{Decide}(F, M)$, which returns a literal l such that $l \in F$ or $\neg l \in F$, and does not appear in M.

Propagation. A clause is *unit* under a trail M if it contains only a single unassigned literal, and evaluates to False under M without that literal. In the above example, if $M = [\neg l_1]$ then l_2 is unit. Notice that because all other variables in the clause have been assigned, in order for the clause to be satisfied l_2 must be assigned the value True. This is called *unit propagation*, and it is how DPLL propagates decisions through the formula. Unit propagation is applied repeatedly to a formula until a fixed point is reached, as oftentimes a single application of the rule will result in additional unit clauses. This process is called *Boolean Constraint Propagation* (BCP).

We model BCP using the function $\mathsf{Propagate}(F, M)$ which takes a clause set F and trail M. $\mathsf{Propagate}$ finds a unit clause with unassigned literal l, constructs a new trail M' by appending l to M as an implied literal, and repeats the process until no further clauses in F are unit under M'. The new trail M' is returned.

Conflict-driven clause learning. On finding a conflicting clause, the original DPLL algorithm backtracks by always erasing the most recent decision l from M, and appending $\neg l$ to continue the search. CDCL can often backtrack further by analyzing the conflicting clause and adding a new clause (a *lemma*) to F. Doing so achieves two optimizations: decisions that are irrelevant to the conflict are erased and their negations are not searched, and conflicts that derive from the same root cause are avoided in the future.

To see how this works, consider the example in Figure 2.5, which appeared in (Nieuwenhuis et al., 2006). The conflict on the final line is

M	F	*Reason*
$\varnothing$	$\{\{l_1, l_2\}, \{l_3, l_4\}, \{\neg l_2, l_3, \neg l_4\}, \{l_5, l_6\}\}$	Decide $\neg l_1$
$[\neg l_1^d]$	$\{\{F, l_2\}, \{l_3, l_4\}, \{\neg l_2, l_3, \neg l_4\}, \{l_5, l_6\}\}$	Propagate l_2
$[\neg l_1^d, l_2]$	$\{\{F, T\}, \{l_3, l_4\}, \{F, l_3, \neg l_4\}, \{l_5, l_6\}\}$	Decide $\neg l_5$
$[\neg l_1^d, l_2, \neg l_5^d]$	$\{\{F, T\}, \{l_3, l_4\}, \{F, l_3, \neg l_4\}, \{F, l_6\}\}$	Propagate l_6
$[\neg l_1^d, l_2, \neg l_5^d, l_6]$	$\{\{F, T\}, \{l_3, l_4\}, \{F, l_3, \neg l_4\}, \{F, T\}\}$	Decide $\neg l_3$
$[\neg l_1^d, l_2, \neg l_5^d, l_6, \neg l_3^d]$	$\{\{F, T\}, \{F, T\}, \{F, F, \neg l_4\}, \{F, T\}\}$	Propagate l_4
$[\neg l_1^d, l_2, \neg l_5^d, l_6, \neg l_3^d, l_4]$	$\{\{F, T\}, \{F, T\}, \{F, F, F\}, \{F, T\}\}$	**Conflict**

Figure 2.5: This example illustrates conflict-driven backtracking and clause learning. Chronological backtracking, as used in classic DPLL, will need to backtrack to the first decision, through the irrelevant clause $l_5 \vee l_6$, in order to resolve the conflict. CDCL can use the conflict to learn the lemma $l_1 \vee l_3$, and thus backtrack immediately to the first decision on l_1.

in the third clause $\neg l_2 \vee l_3 \vee \neg l_4$, and arises because of the propagation of l_2 after the decision $\neg l_1^d$. From this the procedure can infer that the clause set entails $l_1 \vee l_3$ because $\neg l_1^d$ is incompatible with $\neg l_3^d$ and its unit propagation l_4. This is called a *backjump clause*, and if it were in the original clause set, it would have facilitated the propagation of l_3 immediately after the first decision, thus avoiding the conflict. Thus, backtracking proceeds to this level, adds the unit-propagated literal immediately to M, and inserts the conflict clause as a lemma in F. We will not go into further detail on lemma learning; for more information see (Nieuwenhuis et al., 2006) for an excellent overview.

We model CDCL by the function $\mathsf{BackjumpLemma}(F, M)$. Given a clause set F that is in conflict under M, BackjumpLemma:

1. finds a split M_1, M_2 such that $M_1 \| M_2 = M$ and M_2 begins with a decision literal,

2. finds a backjump clause $G \vee l$ such that $F \models G \vee l$, $\mathsf{Conflict}(G, M_1)$, l is undefined in M_1, and l or $\neg l$ appears in F,

3. returns a new trail $M' = [M_1, l]$ (l is appended as an implied literal) and a new clause set F' corresponding to $F \wedge (G \vee l)$

It is always possible to find such a split and backjump clause (Nieuwenhuis et al., 2006).

Abstract Decision Procedures

It is common to specify decision procedures abstractly using transition rules (de Moura and Jovanović, 2013; Nieuwenhuis et al., 2006, 2005). This approach separates key assumptions of the procedure from the details of how they operate efficiently, facilitating cleaner and more direct formal reasoning about the procedure's correctness properties.

We describe DPLL in this way as a *transition system* over states comprising pairs $\langle M, F \rangle$ of trails and clause sets as well as the terminal states Sat and Unsat. The core of the system is the relation $\mapsto$ which we write using infix notation, e.g., $\langle M, F \rangle \mapsto \langle M', F' \rangle$ to denote the fact that a procedure in the pre-state $\langle M, F \rangle$ can transition to the post-state $\langle M', F' \rangle$. A sequence of transitions is a *derivation*. The transition system is comprised of a set of rules, and each rule is specified in the form:

$$\langle M, F \rangle \mapsto \langle M', F' \rangle \quad \textbf{if} \quad \Phi(\langle M, F \rangle)$$

for some condition Φ over the pre-state. Such a rule means that a procedure currently in state $\langle M, F \rangle$ *may* transition to $\langle M', F' \rangle$ if $\Phi(\langle M, F \rangle)$ is true. Note that if another rule's condition is also true in $\langle M, F \rangle$, then the procedure can arbitrary select which rule to apply. If there is no rule in the system whose condition applies to the current state, then that is the *final* state and the procedure terminates.

The DPLL procedure given in Figure 2.7 corresponds to the transition system in Figure 2.8. Each rule corresponds to a key component of the procedure. The simplest rule is Sat, which transitions to the terminal Sat state whenever F is satisfied under M. Similarly, Unsat transitions to Unsat

Subroutine	*Description*
Conflict(F, M)	Returns True if F is in conflict under the assignments in M, and False otherwise.
Satisfied(F, M)	Returns True if F is satisfied under the assignments in M, and False otherwise.
Decide(F, M)	Return an unassigned literal (or its negation) from F.
Propagate(F, M)	Apply unit propagation on F repeatedly until no further clauses are unit. Return an updated trail.
BackjumpLemma(F, M)	When F is in conflict under M, generate a lemma and backtrack in M to the appropriate point. Return the clause set with the new lemma and the trail after backtracking.

Figure 2.6: Subroutines used by DPLL

Algorithm DPLL(F, M)

Input: $F = \text{Clauses}(\phi)$ for a CNF ϕ,
M a trail of assignments to literals in F

Output: Sat if ϕ is satisfiable, Unsat otherwise

```
1.  M ← Propagate(F, M)
2.  If Conflict(F, M):
3.      If M contains no decision literals:
4.          Return Unsat
5.      Else:
6.          (F', M') ← BackjumpLemma(F, M)
7.          Return DPLL(F', M')
8.  Else if Satisfied(F, M):
9.      Return Sat
10. Else:
11.     l ← Decide(F, M)
12.     Append l to M as a decision literal
13.     Return DPLL(F, M)
```

Figure 2.7: DPLL algorithm with conflict-driven clause learning

DECIDE		
$\langle M, F\rangle \mapsto \langle [M, l^d], F\rangle$	**if**	l or $\neg l$ appears in F l is unassigned in M
PROPAGATE		
$\langle M, F\rangle \mapsto \langle [M, l], F\rangle$	**if**	F contains a clause $\{l_1, \ldots, l_n, l\}$ $M \models \neg(l_1 \vee \cdots \vee l_n)$ l is unassigned in M
BACKJUMPLEMMA		
$\langle [M_1, l^d, M_2], F\rangle \mapsto \langle [M_1, l'], F \cup G\rangle$	**if**	F contains a clause $\{l_1, \ldots, l_n\}$ $[M_1, l^d, M_2] \models \neg(l_1 \vee \ldots \vee l_n)$ There exists $G = \{l'_1, \ldots, l'_m, l'\}$ $F \models G$ $M_1 \models \neg(l'_1 \vee \cdots \vee l'_n)$ l' is unassigned in M_1 l' or $\neg l'$ occurs in F
SAT		
$\langle M, F\rangle \mapsto \mathsf{Sat}$	**if**	$\mathsf{Satisfied}(F, M)$
UNSAT		
$\langle M, F\rangle \mapsto \mathsf{Unsat}$	**if**	$\mathsf{Conflict}(F, M)$ M has no decided literals

Figure 2.8: Transition system for DPLL with conflict-driven clause learning and backjumping

whenever F is in conflict under M and there are no further opportunities for backjumping. PROPAGATE applies unit propagation: if F contains a clause $\{l_1, \ldots, l_n, l\}$ where the first n literals are unsatisfied and the last literal l is unassigned, then l is added to the end of the trail as an implied literal. The most complex rule corresponds to conflict-driven backjumping and clause learning, BACKJUMPLEMMA. The clause $\{l_1, \ldots, l_n\}$ is the conflict clause, as it evaluates to false under the pre-state trail $[M_1, l^d, M_2]$. The lemma $G = (\neg(l'_1 \vee \cdots \vee l'_n) \implies l')$ is entailed by F, so the new trail becomes $[M_1, l']$ because $M_1 \models \neg(l'_1 \vee \cdots \vee l'_n)$, thus implying l'.

DPLL(T): DPLL Modulo Theories

We now turn to the problem of SMT for a first-order theory Γ over language L. Here we assume that restricted formulas, consisting only of conjunctions of atomic formulas or their negations, are decidable in the quantifier-free fragment of Γ, and that we already have a procedure $\mathsf{Sat}_\Gamma(\phi)$ which does this for us.

DPLL(T) refers to a class of procedures that are based on the classic DPLL algorithm. These procedures exploit the fact that a first-order formula ϕ cannot be satisfied if its *Boolean skeleton*, denoted $\mathbf{B}(\phi)$, is not also satisfied. $\mathbf{B}(\phi)$ is defined by replacing each atomic formula in ϕ (or its negation) with a unique propositional literal. For a skeleton $\phi_{\mathbf{B}}$, we write $\Gamma_{\mathbf{B}}(\phi_{\mathbf{B}})$ to refer to its corresponding L_{qf} formula. For example, the first-order linear arithmetic formula $\phi \stackrel{\text{def}}{=} x = z \wedge (x = y \vee y = z) \wedge (x < y \vee y < z)$ would have the skeleton $\mathbf{B}(\phi) = l_1 \wedge (l_2 \vee l_3) \wedge (l_4 \vee l_5)$, so $\Gamma_{\mathbf{B}}(l_1) = (x = z)$, $\Gamma_{\mathbf{B}}(l_2) = (x = y)$, $\Gamma_{\mathbf{B}}(l_3) = (y = z)$, $\Gamma_{\mathbf{B}}(l_4) = x < y$, and $\Gamma_{\mathbf{B}}(l_5) = y < z$. We abuse notation and write $\Gamma_{\mathbf{B}}(M)$ for a trail M to denote the conjunction of each literal in M under $\Gamma_{\mathbf{B}}(\cdot)$.

Observe that when $\mathbf{B}(\phi) = (l_{1,1} \vee \cdots \vee l_{1,n_1}) \wedge \cdots \wedge (l_{m,1} \vee \cdots \vee l_{m,n_m})$ is a CNF formula satisfied by a trail $M = [l_{1,i_1}, \ldots, l_{m,i_m}]$, then ϕ is Γ-satisfiable when the conjunction $\Gamma_{\mathbf{B}}(M) = \bigwedge_{j=1..m} \Gamma_{\mathbf{B}}(l_{j,i_j})$ is Γ-satisfiable. DPLL(T) relies on this to drive its basic functionality: begin deciding the propositional formula $\mathbf{B}(\phi)$, and use Sat_Γ to check whether the assignments in trails that satisfy $\mathbf{B}(\phi)$ yield Γ-satisfiable conjunctions of atoms from ϕ. This is often called the *lazy propositional encoding* approach for deciding SMT, because at each step of the procedure, an equisatisfiable propositional encoding of the Γ-formula is constructed until it is rich enough to allow safe termination.

DPLL(T) can thus be described as a transition system that is very similar to that given in Figure 2.8, but a few modifications are in order. First, Satisfied and Conflict must be updated to account for the semantics of Γ

Γ-Propagate		
$\langle M, F\rangle \mapsto \langle [M, l], F\rangle$	**if**	$\Gamma_{\mathbf{B}}(M) \models_\Gamma \Gamma_{\mathbf{B}}(l)$ F contains either l or $\neg l$ l is unassigned in M
Γ-BackjumpLemma		
$\langle [M_1, l^d, M_2], F\rangle \mapsto \langle [M_1, l'], F \cup G\rangle$	**if**	F contains a clause $\{l_1, \ldots, l_n\}$ $[M_1, l^d, M_2] \models \neg(l_1 \vee \ldots \vee l_n)$ There exists $G = \{l'_1, \ldots, l'_m, l'\}$ $\Gamma_{\mathbf{B}}(F) \models_\Gamma \Gamma_{\mathbf{B}}(G)$ $M_1 \models \neg(l'_1 \vee \cdots \vee l'_n)$ l' is unassigned in M_1 l' or $\neg l'$ occurs in F
Γ-Sat		
$\langle M, F\rangle \mapsto \mathsf{Sat}$	**if**	$\mathsf{Satisfied}_\Gamma(F, M)$
Γ-Unsat		
$\langle M, F\rangle \mapsto \mathsf{Unsat}$	**if**	$\mathsf{Conflict}_\Gamma(F, M)$ M has no decided literals

Figure 2.9: Modified transition rules for DPLL(T)

as described above. We define $\mathsf{Satisfied}_\Gamma(F, M)$ to return true whenever $\mathsf{Satisfied}(F, M)$ is true *and* $\mathsf{Sat}_\Gamma(\Gamma_{\mathbf{B}}(M)) = \mathsf{Sat}$. Similarly, $\mathsf{Conflict}_\Gamma(F, M)$ is true whenever $\mathsf{Conflict}(F, M)$ is true *or* $\mathsf{Sat}_\Gamma(\Gamma_{\mathbf{B}}(M)) = \mathsf{Unsat}$. We must also modify the propagation rule to account for theory entailment, so we can add a literal l to the trail whenever it is a consequence of Γ and the current trail M. Finally, BackjumpLemma is modified to use theory entailment to define valid lemmas. Specifically, G should be Γ-entailed by $\Gamma_{\mathbf{B}}(F)$, but still unit under M_1. The remaining rules for Decide and Propagate remain unchanged, as DPLL(T) effectively ignores the semantics of theory atoms and treats them syntactically as propositional variables. The new rules are given in Figure 2.9.

Example 2.5. *Consider the formula from before, where our theory* Γ *is the theory of linear arithmetic:*

$$\begin{aligned} & \phi \stackrel{\text{def}}{=} x = z \wedge (x = y \vee y = z) \wedge (x < y \vee y < z) \\ \textit{where} \quad & \mathbf{B}(\phi) = l_1 \wedge (l_2 \vee l_3) \wedge (l_4 \vee l_5) \end{aligned}$$

Our initial clause set $F = \{\{l_1\}, \{l_2, l_3\}, \{l_3, l_4\}\}$*. We see right away that* l_1 *is unit, so the initial trail becomes:* $M = [l_1]$*. The solver must then decide which literal to assert next. Supposing the solver picks* l_2*, we see that an opportunity arises to apply* Γ-Propagate *twice, because:*

$$\begin{aligned} & M = [l_1, l_2^d],\ \Gamma_{\mathbf{B}}(M) = (x = z \wedge x = y) \\ \textit{so}, \quad & \Gamma_{\mathbf{B}}(M) \models_\Gamma \neg(x < y) \\ \textit{and} \quad & \Gamma_{\mathbf{B}}(M) \models_\Gamma \neg(y < z) \end{aligned}$$

The atoms $x < y$ *and* $y < z$ *appear in the last clause as* l_4 *and* l_5*, neither of which currently appear on the trail. The new trail becomes* $M = [l_1, l_2^d, \neg l_4, \neg l_5]$*. However, this trail is in conflict, as the last clause cannot be satisfied. We see that we can apply* Γ-BackjumpLemma*, if we let* $M_1 = [l_1]$*,* $M_2 = [\neg l_4, \neg l_5]$*, and* $l^d = l_2^d$*. Then our lemma* G *will be* $l_1 \implies \neg l_2$*. Notice that this is more concise than the naive lemma* $(l_1 \wedge l_4 \wedge l_5) \implies \neg l_2$*. However, it is valid because it is entailed by the original clause set under the theory of linear arithmetic. After applying* Γ-BackjumpLemma*, our new clause set is:*

$$F = \{\{l_1\}, \{l_2, l_3\}, \{l_3, l_4\}, \{\neg l_1, \neg l_2\}\}$$

and our new trail is $M = [l_1, \neg l_2]$*, which results in the unit propagation of* l_3 *to arrive at* $M = [l_1, \neg l_2, l_3]$*. However, just as when* l_2 *was decided before, this will result in two applications of* Γ-Propagate *giving us the conflicting trail* $M = [l_1, \neg l_2, l_3, \neg l_4, \neg l_5]$*. The difference now is that* M *contains no decided literals, so the procedure will return* Unsat*.*

2.3 Bounded Software Model Checking

Model checking refers to a set of techniques for verifying properties of state transition systems, typically by exhaustively searching their associated state space under the transition relation (Clarke et al., 1999). These techniques are among the most popular ways of automatically reasoning about the correctness of software. The properties that model checkers verify are expressed in a *temporal logic*, which is a class of formalisms that allow reasoning about the temporal ordering of events without making explicit reference to time.

Early approaches worked by creating explicit representations of a system's state transition graph and traversing them with graph algorithms. However, because of the *state-explosion problem*, which arises in most interesting systems that have too many states to explore individually, *symbolic* model checking techniques were developed in the early 1990's (Burch et al., 1990; Coudert et al., 1991; Pixley, 1991). Symbolic techniques attempt to mitigate the state-explosion problem by considering an *abstract* transition system (or simply *abstraction*) where each state corresponds to a set of states of the targeted system. The original symbolic model checking techniques used Binary Decision Diagrams (BDDs) (Bryant, 1986) to represent the abstraction, but most recent approaches use abstractions based on propositional logic or various first-order theories.

In this section, we discuss one class of symbolic techniques called *satisfiability-based bounded model checking* (BMC) (Biere et al., 1999). Intuitively, Sat-based BMC works by "unrolling" the first k steps of a transition system, encoding the resulting *bounded* system as a formula, and using a satisfiability procedure to verify that the required property holds over the system. While the approach is generally not complete for computations involving more than k steps, it sidesteps the difficult problem of finding inductive invariants that are needed to verify unbounded systems.

Preliminaries

A program $s_{prog} \in S$ is a *statement* from some language S. Statements are given unique labels that range over a set $Label = \{\ell_0, \ell_1, ...\}$, and for a statement s we write $\ell(s)$ as shorthand to refer to its label. Program states come from the set $\Sigma = S \times (X \mapsto D)$, so they are pairs (s, ρ) of statements and *valuations*, or mappings from variables (X) to some domain (D). The *semantics* of S is given by a relation $\rightarrow_S \subseteq \Sigma \times \Sigma$, for which we will use infix notation. Let L be a first-order language for a theory Γ_D whose domain is D. Given a valuation ρ and an L-formula ϕ with free variables from X, we abuse notation and write $\rho(\phi)$ to refer to the formula obtained by replacing free variables in ϕ with their image under ρ. We write $[\![\phi]\!]$ to refer to the valuations described by ϕ, i.e., $[\![\phi]\!] = \{\rho : \rho(\phi) \text{ is } \Gamma_D\text{−satisfiable}\}$.

We assume that L is expressive enough to specify any set of valuations, i.e., for any set of mappings $R \subseteq X \mapsto D$, there exists an L-formula ϕ where $[\![\phi]\!] = R$. Furthermore, we assume that L contains function and relation symbols corresponding to those allowed in expressions from S. Let X_i correspond to the set of variable symbols where each symbol from X is annotated with the subscript $_i$. Then we write $\phi(X_i)$ to refer to the formula obtained by replacing each variable from X appearing in ϕ with its counterpart in X_i. If R is a set of valuations, then we write $\beta_L(R)$ to refer to the L-formula where $R = [\![\beta_L(R)]\!]$.

We will focus on checking one particular type of temporal property called an *invariant* (Definition 11). Invariants are a type of *safety property* (Definition 12) (Lamport, 1977). Intuitively, safety properties stipulate that *something bad will never happen*, where "something bad" refers to a set of program states. An invariant stipulates that at all times during program execution, program variables always have valuations from a particular set. In other words, invariants are safety properties that apply at all times throughout execution of a program.

Definition 11. (Invariant Property). *An invariant property is a set* $\Delta \subseteq X \mapsto D$ *of valuations. A program* s_{prog} satisfies Δ *(alternatively,* s_{prog} models Δ*), written* $s_{prog} \models \Delta$*, if and only if for all states* (s, ρ) *in a finite computation of* s_{prog}*,* $\rho \in \Delta$*:*

$$s_{prog} \models \Delta \Leftrightarrow (\{\rho \mid (s_0, \rho_0) \rightarrow_S^* (s, \rho) \wedge \rho \notin \Delta\} = \varnothing)$$

where (s_0, ρ_0) *is the initial program state and* $\rightarrow_S^*$ *is the transitive closure of* $\rightarrow_S$*.*

Definition 12. (Safety Property). *A safety property is a set* $\Pi \subseteq \mathit{Label} \times (X \mapsto D)$ *of label, valuation pairs. A program* s_{prog} satisfies Π*, written* $s_{prog} \models \Pi$*, if and only if for all states* (s, ρ) *in a finite computation of* s_{prog}*, it is the case that* $(s, \rho) \in \Delta$*:*

$$s_{prog} \models \Pi \Leftrightarrow (\{(\ell(s), \rho) \mid (s_0, \rho_0) \rightarrow_S^* (s, \rho) \wedge (\ell(s), \rho) \notin \Pi\} = \varnothing)$$

where (s_0, ρ_0) *is the initial program state and* $\rightarrow_S^*$ *is the transitive closure of* $\rightarrow_S$*.*

In languages with guard statements (e.g., **if-then-else**), safety properties can be converted to invariants on programs obtained through a simple rewriting that we now describe informally. Given a program s_{prog} and safety property Π, let $v^\star$ be a variable not appearing in s_{prog}, and let $\mathbf{0}^\star$ and $\mathbf{1}^\star$ be two arbitrary elements of D. Then derive a new program s'_{prog} by adding the statement $v^\star = \mathbf{0}^\star$ to the beginning of s_{prog}, and replacing each statement $s \in s_{prog}$ where there exists $(\ell(s), \rho) \in \Pi$ with the statement: $\mathbf{if}(\beta_L(\rho))$ **then** s **else** $v^\star = \mathbf{1}^\star$; s. Then the corresponding invariant property is described by the formula $\Delta = [\![v^\star = \mathbf{0}^\star]\!]$, and $s_{prog} \models \Pi \leftrightarrow s'_{prog} \models \Delta$.

To facilitate symbolic model checking, we introduce a *symbolic post-state operator* $\mathsf{Post}_i(s, \phi)$, which gives an L-formula specifying the valuations that can result from executing a statement s starting in a valuation de-

Assign	
$(x = a, \rho) \to_S (\bot, \rho[x \leftarrow a'])$	**if** $x \in X$, a is a numeric expression $\tau_D(a)$ evaluates to a' under ρ
Sequence$_1$	
$(s_1; s_2, \rho) \to_S (s_1'; s_2, \rho')$	**if** $(s_1, \rho) \to_S (s_1', \rho')$
Sequence$_2$	
$(\bot; s_2, \rho) \to_S (s_2, \rho)$	
If	
(**if** b **then** s_1 **else** $s_2, \rho) \to_S (s', \rho')$	**if** $\tau_B(b)$ evaluates to b' under ρ $(s_1, \rho) \to_S (s', \rho')$ when $b' = \mathsf{True}$ $(s_2, \rho) \to_S (s', \rho')$ when $b' = \mathsf{False}$
While$_1$	
(**while** b **do** $s, \rho) \to_S (s;$ **while** b **do** $s, \rho)$	**if** $\tau_B(b)$ evaluates to True under ρ
While$_2$	
(**while** b **do** $s, \rho) \to_S (\bot, \rho)$	**if** $\tau_B(b)$ evaluates to False under ρ

(a) Structural operational semantics of While

$$
\begin{aligned}
\mathsf{Post}_i(x = a, \phi) &= \phi \wedge \left(\textstyle\bigwedge_{v \in X - \{x\}} v_i = v_{i-1}\right) \wedge x_i = \tau_D(a)[V_{i-1}] \\
\mathsf{Post}_i(s_1; s_2, \phi) &= \mathsf{Post}_{i+1}(s_2, \mathsf{Post}_i(s_1, \phi)) \\
\mathsf{Post}_i(\bot, \phi) &= \phi \wedge \left(\textstyle\bigwedge_{v \in X} v_i = v_{i-1}\right) \\
\mathsf{Post}_i(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2, \phi) &= \begin{array}{l}(\tau_B(b)[V_{i-1}] \wedge \mathsf{Post}_i(s_1, \phi)) \\ \vee\ (\neg\tau_B(b)[V_{i-1}] \wedge \mathsf{Post}_i(s_2, \phi))\end{array} \\
\mathsf{Post}_i(\textbf{while } b \textbf{ do } s, \phi) &= \begin{array}{l}(\tau_B(b)[V_{i-1}] \wedge \mathsf{Post}_i(s; \textbf{while } b \textbf{ do } s, \phi)) \\ \vee\ (\neg\tau_B(b)[V_{i-1}] \wedge \mathsf{Post}_i(\bot, \phi))\end{array}
\end{aligned}
$$

(b) Symbolic post-state operator Post_i **for** While

Figure 2.10: Operational semantics and symbolic Post_i operator for a simple language While. Because the semantics have no rules whose pre-state corresponds to the empty statement ϕ, a While-program terminates after entering it. τ_D and τ_B transform numeric and Boolean program expressions into L-terms and formulas, respectively. See Example 2.6 for details.

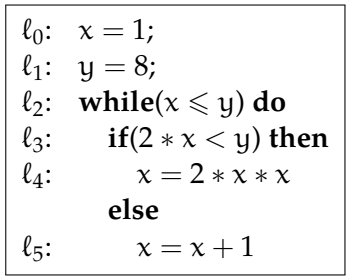
scribed by ϕ:

$$\mathsf{Post}_i(s, \phi) = \bigvee_{\phi' \in \Psi} \phi'(V_i)$$
$$\textit{where } \Psi = \{\beta_L(\rho') \mid \exists \rho \in [\![\phi(V_{i-1})]\!].(s, \rho) \to_S (s', \rho')\}$$

Notice that $\mathsf{Post}_i(s, \phi)$ is a *two-vocabulary formula*, as it is defined over two sets of variables V_{i-1} and V_i. We use this convention to denote valuations from the pre-state, annotated with the subscript $_{i-1}$, from those in the post state, annotated with $_i$. Note that in languages with recursive syntax, $\mathsf{Post}_i(s, \phi)$ may contain additional vocabularies corresponding to *intermediate* states, but these are less essential as they can be removed using quantifiers to arrive at an equisatisfiable two-vocabulary formula.

Example 2.6. (Simple While language). *Consider a simple structured language whose statements can be assignments, sequences of statements,* ***if-then-else*** *statements, and* ***while*** *statements. The domain over which variables range is the integers, and arithmetic expressions can use addition, subtraction, and multiplication. Boolean expressions can use the normal connectives, in addition to the inequality relation $\leqslant$. A suitable choice for Γ_D is then the quantifier-free theory of integer arithmetic, and L must have function symbols corresponding to $\{+, -, \times\}$ and a relation symbol corresponding to $\leqslant$.*

The semantics of these statements is given in Figure 2.10(a). Note the use of the transformers τ_D and τ_B. τ_D transforms numeric expressions from While *into terms in L by replacing program operators with their counterparts in L. τ_B does the same for Boolean* While *expressions. Figure 2.4(b) gives the definition of Post_i for this language. Notice that the two-vocabulary structure of Post_i is obvious in the definition of the non-compound assignment and termination statements: the constraints on variables in the post-state are given entirely in terms of variables from the pre-state, whose subscript annotations differ by one. For the compound statements, subscripts between pre- and post-states may differ*



$$
\begin{array}{lll}
 & & (\ell_0; \ell_1; \ell_2, \varnothing) \\
(\textsc{Assign}, \textsc{Sequence}_2) & \to_S & (\ell_1; \ell_2, [x \mapsto 1]) \\
(\textsc{Assign}, \textsc{Sequence}_2) & \to_S & (\ell_2, [x \mapsto 1, y \mapsto 8]) \\
(\textsc{While}_1) & \to_S & (\ell_3; \ell_2, [x \mapsto 1, y \mapsto 8]) \\
(\textsc{If}, \textsc{Sequence}_1) & \to_S & (\ell_4; \ell_2, [x \mapsto 1, y \mapsto 8]) \\
(\textsc{Assign}, \textsc{Sequence}_2) & \to_S & (\ell_2, [x \mapsto 2, y \mapsto 8]) \\
(\textsc{While}_1) & \to_S & (\ell_3; \ell_2, [x \mapsto 2, y \mapsto 8]) \\
(\textsc{If}, \textsc{Sequence}_1) & \to_S & (\ell_4; \ell_2, [x \mapsto 2, y \mapsto 8]) \\
(\textsc{Assign}, \textsc{Sequence}_2) & \to_S & (\ell_2, [x \mapsto 8, y \mapsto 8]) \\
(\textsc{While}_1) & \to_S & (\ell_3; \ell_2, [x \mapsto 8, y \mapsto 8]) \\
(\textsc{If}, \textsc{Sequence}_1) & \to_S & (\ell_5; \ell_2, [x \mapsto 8, y \mapsto 8]) \\
(\textsc{Assign}, \textsc{Sequence}_2) & \to_S & (\ell_2, [x \mapsto 9, y \mapsto 8]) \\
(\textsc{While}_2) & \to_S & (\bot, [x \mapsto 9, y \mapsto 8])
\end{array}
$$

Figure 2.11: Sample While program and its execution under the semantics from Figure 2.10(a). Note that we usually apply the Sequence rules simultaneously with others to avoid tedious redundancies in the derivation. Sometimes we use $\textsc{Sequence}_1$ without writing it in the derivation, instead writing the rule that applies immediately after it, when the necessary application of $\textsc{Sequence}_1$ is obvious.

by more than one.

To demonstrate how these components work, Figure 2.11 shows an example While *program and its execution under the operational semantics. Note that when writing instances of the operational semantics rules, we substitute labels for statements in order to make the text fit on a single line. The first two steps are straightforward assignments, bringing the valuation from* $\varnothing$ *to* $[x \mapsto 1, y \mapsto 8]$*. In the third step,* $\textsc{While}_1$ *is applied because* $\tau_B(x \leqslant y) = x \dot{\leqslant} y$ *and* $\rho = [x \mapsto 1, y \mapsto 8]$*, so* $\rho(\tau_B(x \leqslant y)) = 1 \leqslant 8 = \mathsf{True}$*. This leads to an update to* x *in the fifth step, where* $\tau_D(2 * x * x) = \dot{2} \dot{\times} x \dot{\times} x$ *and* $\rho(x) = 1$ *giving the post-state valuation* $\rho' = [x \mapsto 2, y \mapsto 8]$*.*

Satisfiability-Based Bounded Software Model Checking

We are now ready to describe bounded software model checking of invariant properties. We begin by defining a slight modification to the symbolic post-state operator, $\mathsf{Post}_i^{\leqslant k}$, that effectively terminates for computations deeper than the bound k. We achieve this by replacing any rule defining an instance of Post_i with one that behaves normally whenever $i \leqslant k$, and simply returns the input condition ϕ otherwise:

$$\mathsf{Post}_i^{\leqslant k}(s, \phi) = \begin{cases} \mathsf{Post}_i(s, \phi) & \text{if } i \leqslant k \\ \phi & \text{otherwise} \end{cases} \tag{2.1}$$

$\mathsf{Post}_i^{\leqslant k}$ gives us a way to derive an L-formula $\phi^k_{s_{prog}}$ that is satisfiable if and only if s_{prog} violates a given invariant Δ. Specifically,

$$\phi^k_{s_{prog}} = \mathsf{Post}_1^{\leqslant k}(s_{prog}, \mathsf{True}) \wedge \left(\bigvee_{i=1}^{k} (\neg \beta_L(\Delta))[V_i] \right) \tag{2.2}$$

The second clause corresponds to an *unrolling* of the invariant across each intermediate state. This is achieved by stipulating that for at least one vocabulary corresponding to the i*th* step of the computation (i.e, V_i), the negation of the invariant should hold (i.e., $\neg(\beta_L(\Delta))$). Thus if any literal from this clause holds, *and* the symbolic postconditions for each intermediate state can be satisfied (i.e., $\mathsf{Post}_1^{\leqslant k}(s_{prog}, \mathsf{True})$ is Γ_D-satisfiable), then the invariant can be violated by some sequence of valuations and statements in the program. We then perform bounded model checking by using a decision procedure for Γ_D to check the satisfiability of $\phi^k_{s_{prog}}$. If the procedure returns Unsat, then we know that $s_{prog} \models \Delta$ in k steps. Otherwise, a Sat result indicates the presence of a bug that violates Δ, and we can derive a complete trace of the bug from a satisfying assignment to $\phi^k_{s_{prog}}$.

Let us see through an example. Figure 2.12(a) shows the WHILE program from Figure 2.11, and Figure 2.12(b) shows a version that has been

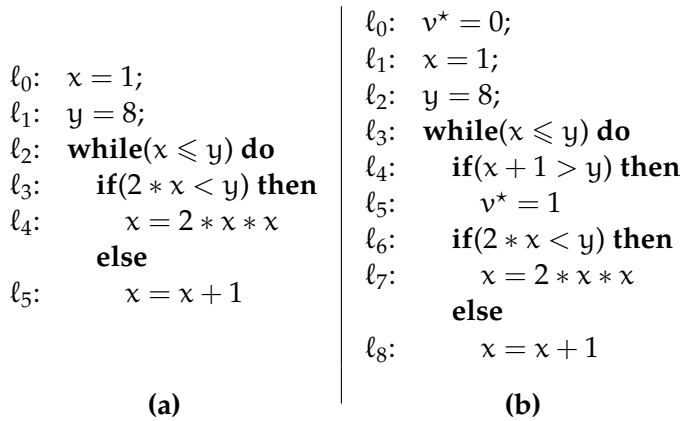


Figure 2.12: Program from Figure 2.11 transformed to allow checking the safety property $\Pi = \{(\ell_3, \rho) \mid \rho \in [\![x + 1 \leqslant y]\!]\}$ by checking the invariant $\Delta = [\![v^\star = 0]\!]$.

modified to allow checking the safety property $\Pi = \{(\ell_3, \rho) \mid \rho \in [\![x + 1 \leqslant y]\!]\}$ via the invariant $\Delta = [\![v^\star = 0]\!]$. We will verify that the program in Figure 2.12(b) satisfies Δ on all traces with at most 4 steps. In the following, we will simplify formulas that result from the application of $\mathsf{Post}_i^{\leqslant k}$ liberally to keep the presentation clear.

To begin, we set $k = 4$ and apply $\mathsf{Post}_1^{\leqslant 4}$ to the sequence statement $\ell_0; \ell_1; \ell_2; \ell_3$ and the formula $v^\star \neq 0$:

$$\begin{aligned}\mathsf{Post}_1^{\leqslant 4}(\ell_0; \ell_1; \ell_2; \ell_3, \mathsf{True}) &= \mathsf{Post}_2^{\leqslant 4}(\ell_1; \ell_2; \ell_3, \mathsf{Post}_1^{\leqslant 4}(\ell_0, \mathsf{True})) \\ &= \mathsf{Post}_2^{\leqslant 4}(\ell_1; \ell_2; \ell_3, v_1^\star = 0)\end{aligned}$$

Observe the simplification we made by omitting conjuncts for x_1 and y_1, which have not yet been assigned. The full result would actually be:

$$\mathsf{Post}_2^{\leqslant 4}(\ell_1; \ell_2; \ell_3, v_1^\star = 0 \wedge x_1 = x_0 \wedge y_1 = y_0)$$

As we work through the example, we will simplify formulas liberally to draw out important changes and reduce clutter in our derivation. In this

case, we simplified by removing a trivially-satisfiable subformula, but we will also frequently propagate equalities. However, we will not simplify parts of the formula that could change the final result, such as updates to $v^\star$. Continuing, the statements labeled ℓ_1, ℓ_2 are also assignments, so,

$$\begin{aligned}\mathsf{Post}_2^{\leqslant 4}(\ell_1; \ell_2; \ell_3, v_1^\star = 0) &= \mathsf{Post}_3^{\leqslant 4}(\ell_2; \ell_3, v_1^\star = 0 \wedge v_2^\star = 0 \wedge x_2 = 1) \\ &= \mathsf{Post}_4^{\leqslant 4}(\ell_3, v_1^\star = 0 \wedge v_2^\star = 0 \wedge v_3^\star = 0 \wedge \\ &\qquad x_3 = 1 \wedge y_3 = 8)\end{aligned}$$

The next statement at ℓ_3 is a **while** whose body is a sequence of **if** statements labeled ℓ_4 and ℓ_6. The resulting symbolic postcondition introduces the first disjunction in order to distinguish valuations that enter the loop body from those that pass over it. In the following, let

$$\phi^3_{s_{\text{prog}}} = (v_1^\star = 0 \wedge v_2^\star = 0 \wedge v_3^\star = 0 \wedge x_3 = 1 \wedge y_3 = 8)$$

We have:

$$\begin{aligned}&\mathsf{Post}_4^{\leqslant 4}(\ell_3, \phi^3_{s_{\text{prog}}}) = \\ &\qquad (x_3 \leqslant y_3 \wedge \mathsf{Post}_4^{\leqslant 4}(\ell_4; \ell_6, \phi^3_{s_{\text{prog}}})) \vee (x_3 > y_3 \wedge \mathsf{Post}_4^{\leqslant 4}(\bot, \phi^3_{s_{\text{prog}}}))\end{aligned}$$

To complete this formula, we are obliged to derive $\mathsf{Post}_4^{\leqslant 4}(\ell_4; \ell_6, \phi^3_{s_{\text{prog}}})$ and $\mathsf{Post}_4^{\leqslant 4}(\bot, \phi^3_{s_{\text{prog}}})$. The latter is simply:

$$\mathsf{Post}_4^{\leqslant 4}(\bot, \phi^3_{s_{\text{prog}}}) = \phi^3_{s_{\text{prog}}} \wedge x_4 = x_3 \wedge y_4 = y_3 \wedge v_4^\star = v_3^\star$$

Note that this formula is unsatisfiable when conjoined with $x_3 > y_3$, because $x_3 = 1, y_3 = 8$. This means that,

$$\mathsf{Post}_4^{\leqslant 4}(\ell_3, \phi^3_{s_{\text{prog}}}) = x_3 \leqslant y_3 \wedge \mathsf{Post}_4^{\leqslant 4}(\ell_4; \ell_6, \phi^3_{s_{\text{prog}}})$$

Continuing on:

$$\begin{aligned}\mathsf{Post}_4^{\leqslant 4}(\ell_4;\ell_6,\phi^3_{s_{prog}}) = \mathsf{Post}_5^{\leqslant 4}(\ell_6,\ &(x_3+1>y_3 \wedge \mathsf{Post}_4^{\leqslant 4}(\ell_5,\phi^3_{s_{prog}})) \quad \vee \\ &(x_3+1\leqslant y_3 \wedge \mathsf{Post}_4^{\leqslant 4}(\bot,\phi^3_{s_{prog}})))\end{aligned}$$

Which leads us to:

$$\begin{aligned}\mathsf{Post}_4^{\leqslant 4}(\ell_5,\phi^3_{s_{prog}}) &= \phi^3_{s_{prog}} \wedge v_4^\star = 1 \wedge x_4 = x_3 \wedge y_4 = y_3\\ \mathsf{Post}_4^{\leqslant 4}(\bot,\phi^3_{s_{prog}}) &= \phi^3_{s_{prog}} \wedge v_4^\star = v_3^\star \wedge x_4 = x_3 \wedge y_4 = y_3\end{aligned}$$

Notice that $\mathsf{Post}_4^{\leqslant 4}(\ell_5,\phi^3_{s_{prog}})$ is unsatisfiable, because $x_3+1>y_3$ is not consistent with $\phi^3_{s_{prog}}$, so

$$\begin{aligned}\mathsf{Post}_4^{\leqslant 4}(\ell_4;\ell_6,\phi^3_{s_{prog}}) &= \mathsf{Post}_5^{\leqslant 4}(\ell_6, x_3+1\leqslant y_3 \wedge \mathsf{Post}_4^{\leqslant 4}(\bot,\phi^3_{s_{prog}}))\\ &= x_3+1\leqslant y_3 \wedge \phi^3_{s_{prog}} \wedge v_4^\star = v_3^\star\end{aligned}$$

We removed the final updates to x_4, y_4, as they are irrelevant to the invariant we are checking. Going back to $\mathsf{Post}_4^{\leqslant 4}(\ell_3,\phi^3_{s_{prog}})$, we can plug this in to arrive at the final result:

$$\begin{aligned}\mathsf{Post}_4^{\leqslant 4}(\ell_3,\phi^3_{s_{prog}}) &= x_3\leqslant y_3 \wedge x_3+1\leqslant y_3 \wedge \phi^3_{s_{prog}} \wedge v_4^\star = v_3^\star\\ &= x_3+1\leqslant y_3 \wedge x_3 = 1 \wedge y_3 = 8 \wedge\\ &\quad v_1^\star = 0 \wedge v_1^\star = 0 \wedge v_3^\star = 0 \wedge v_4^\star = 0\end{aligned}$$

Now we add the "unrolled" invariant condition,

$$\bigvee_{i=1}^{4} \neg(\beta_L(\Delta))[V_i] = (v_1^\star \neq 0 \vee v_2^\star \neq 0 \vee v_3^\star \neq 0 \vee v_4^\star \neq 0)$$

Clearly, this is inconsistent with $\mathsf{Post}_1^{\leqslant 4}(\ell_0;\ell_1;\ell_2;\ell_3,\mathsf{True}) = \mathsf{Post}_4^{\leqslant 4}(\ell_3,\phi^3_{s_{prog}})$, so we conclude that the program in Figure 2.12(b) does not violate $\Delta = [\![v^\star = 0]\!]$ in four steps.

2.4 Parametric Lattice-Point Counting

We now consider the problem of enumerating *lattice points* in linear systems. We will focus on the d-dimensional integer lattice, which we denote $\mathbb{Z}^d$, with points corresponding to d-tuples of integers. The linear systems that we consider are *rational polyhedra* (Definition 13), and correspond to the set of all points in d-dimensional Euclidean space that satisfy a given set of constraints.

Definition 13. (Rational polyhedron). *A rational polyhedron* $P \subset \mathbb{R}^d$ *is the set of solutions of a finite system of* n *linear inequalities with integer coefficients:*

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

where $A \in \mathbb{Z}^{n \times d}$, *and* $\gamma \in \mathbb{Z}^n$ *are constants.*

Given a rational polyhedron P, we are interested in counting the number of lattice points in $P \cap \mathbb{Z}^d$. Barvinok presented a constructive proof that this can be done in time polynomial in n, the number of inequalities, when the dimension d of the polyhedron is fixed in advance (Barvinok, 1994). His proof is based on computing a generating function for the set of lattice points in P. Definition 14 gives the lattice-point generating function for a rational polyhedron.

Definition 14. (Lattice-point generating function (Barvinok, 1994)). *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* $\mathbf{x} = (x_1, \ldots, x_d)$,

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

Observe that the generating function contains one monomial for each lattice point in P, so $f(P, [1, \ldots, 1]) = |P \cap \mathbb{Z}^d|$, because each monomial will take the value 1. Thus, the lattice-point generating function provides

a way to count the number of lattice points in P. Note, however, that even deciding whether $|P \cap \mathbb{Z}^d| > 0$ is NP-Hard, so computing f is not polynomial if d is allowed to change.

The details of Barvinok's proof are beyond the scope of this section, but we present his main result in Theorem 2.7.

Theorem 2.7. (Barvinok, 1994) *There is an 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 $f(P_i; \mathbf{x})$ 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 $f(P; \mathbf{x}) = 0$.*

This algorithm runs in time $L^{O(d)}$, where L is the size of the the constraints $A \in \mathbb{Z}^{n \times d}$, and $\gamma \in \mathbb{Z}^n$ used to specify P (see Definition 13 for details).

Property *(3)* of Theorem 2.7 is perhaps the most important takeaway: in polynomial time for fixed dimension, we can compute the generating function of a rational polyhedron. Property *(1)* implies that the inclusion-exclusion principle holds, so geometric compositions of polyhedra translate to similar compositions of their valuations. Property *(2)* says that geometric translations of P correspond to adding the translation vector to the exponent of each term in f, as expected. *(4)* states that unbounded polyedra have a constant-zero valuation, so the theorem is compatible with our finite-base counting operations.

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* (Verdoolaege et al., 2007).

Definition 15. (Rational Parametric Polyhedron). *A rational parametric polyhedron* $P_{\mathbf{p}}$ *is the set of solutions to a set of* m *linear inequalities with integer coefficients of the form*

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

where $\mathbf{p} \in \mathbb{Z}^{d'}$ *is the* parameter set, $A \in \mathbb{Z}^{m \times d}$, $B \in \mathbb{Z}^{m \times d'}$, *and* $\gamma \in \mathbb{Z}^m$ *are constant vectors.*

Verdoolaege *et al.* give an algorithm for computing valuations $f_{\mathbf{p}}(P_{\mathbf{p}}; \mathbf{x})$ over rational parametric polytopes, and evaluating them at $\mathbf{x} = \mathbf{1} = (1, \ldots, 1)$ (Verdoolaege et al., 2007). The result is a *parametric enumerator* (Definition 16), that represents the number of lattice points in $P_{\mathbf{p}}$ for a given configuation of $\mathbf{p}$.

Definition 16. (Parametric Enumerator). *The* parametric enumerator *for parametric polyhedron* $P_{\mathbf{p}}$ *is a piecewise quasi-polynomial*

$$f_{\mathbf{p}}(P_{\mathbf{p}}) = \begin{cases} p_1(\mathbf{p}), & \text{if } \phi_1(\mathbf{p}) \\ \vdots & \\ p_n(\mathbf{p}), & \text{if } \phi_n(\mathbf{p}) \end{cases}$$

where each chamber polynomial $p_i(\mathbf{p})$ *is a quasi-polynomial with rational coefficients, and each* chamber constraint $\phi(\mathbf{p})$ *is a conjunction of linear inequalities. For a fixed set of parameters, the parametric enumerator is equivalent to the Barvinok valuation (Theorem 2.7) evaluated at* $\mathbf{1}$.

To illustrate the structure of a quasi-polynomial, consider counting the parametric polytope $P_{a,b} = \{(x, y) \in \mathbb{R}^2 \ : \ 0 <= y <= a \wedge y <= x <= b\}$

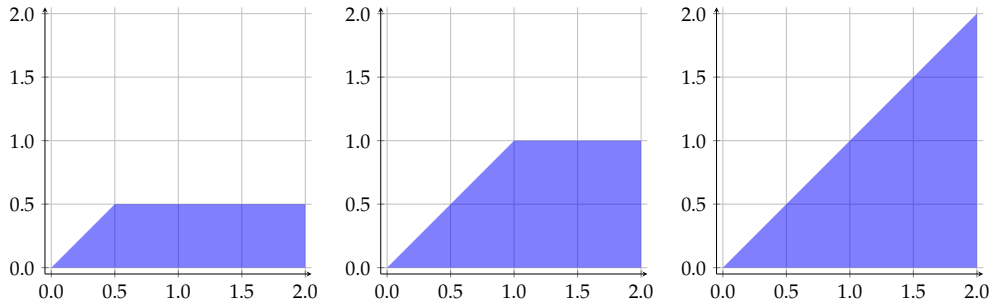


Figure 2.13: The parametric polytope $P_{a,b} = \{(x, y) \in \mathbb{R}^2 \ : \ 0 <= y <= a \wedge y <= x <= b\}$, evaluated at $a = \frac{1}{2}, b = 2$ (left), $a = 1, b = 2$ (middle), and $a = 2, b = 2$ (right). The first two parameter values reside in a different chamber than the last, so they do not share the same set of vertices and require different polynomials to characterize their lattice-point cardinalities (see Equation 2.3). Notice that changing parameter values within a chamber, as shown in the left and middle images, translates the vertices but does not change their relationship to one another.

which is shown in Figure 2.13 for three settings of the parameters a and b. The parametric enumerator is given by:

$$f_s(P_s) = \begin{cases} 1 + \frac{1}{2}a - \frac{1}{2}a^2 + ab + b & \text{if } 0 \leqslant a < b \\ 1 + \frac{3}{2}b + \frac{1}{2}b^2 & \text{if } 0 \leqslant b \leqslant a \end{cases} \tag{2.3}$$

Each piece of this enumerator corresponds to a *chamber* of P_s. The value taken by the enumerator corresponds to the *chamber polynomial*, and the corresponding condition is the *chamber constraint*. The chambers subdivide the parameter space into a polynomial number of convex subspaces where the vertices of the parametric polytope are invariant. Within each chamber, the location of the vertices may change, but their relationship to eachother (i.e., the faces with which they intersect) remains the same. This is demonstrated in Figure 2.13, where an additional vertex appears in the right-most image, but the vertices in the two left-most images remain the same modulo translation.

3 THE TENSION BETWEEN PRIVACY AND UTILITY

This chapter examines the seemingly fundamental tension between concerns of privacy and utility in data-centric applications that use Machine Learning (ML). Rather than equating privacy with a parameter in an abstract definition, Section 3.1 considers at an approach called Model Inversion (MI) that measures the risk of successful inference on selected variables by taking into account the semantics of an ML model. Section 3.2 then uses MI in concert with a set of detailed, application-specific utility metrics to illustrate a method for better understanding this tension, with an emphasis on the role of Differential Privacy in establishing a suitable privacy-utility balance.

3.1 Model Inversion

This section uses the notation for databases, rows, and distributions given in Section 2.1. Let D_t and D_v be two databases containing n and m rows, respectively, each i.i.d. sampled from the same distribution $\mathbb{D}$. We call D_t the *training set* and D_v the *validation set*. Assume that $X_1, \ldots, X_d, Y$ are all discrete-valued domains. We simply write X to refer to the product domain $X_1 \times \cdots \times X_d$ and X_{-i} for the domain $X_1 \times \cdots \times X_{i-1} \times X_{i+1} \times \cdots \times X_d$. For an index set $S \subset \{1, \ldots, d\}$ where $S = \{S_1, \ldots, S_{|S|}\}$, we write X_S to refer to the domain $X_{S_1} \times \cdots \times X_{S_{|S|}}$. For an element $x \in X$, we write x_i to refer to its i*th* element, x_{-i} for $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_d$, and x_S for $x_{S_1}, \ldots, x_{S_{|S|}}$.

Let $\mathbb{D}_1, \ldots, \mathbb{D}_d, \mathbb{D}_y$ correspond to the *first-order marginal distributions* of $\mathbb{D}$. Let p correspond to the probability density function of $\mathbb{D}$, p_i correspond to that of the i*th* marginal, and p_y correspond to the response marginal,

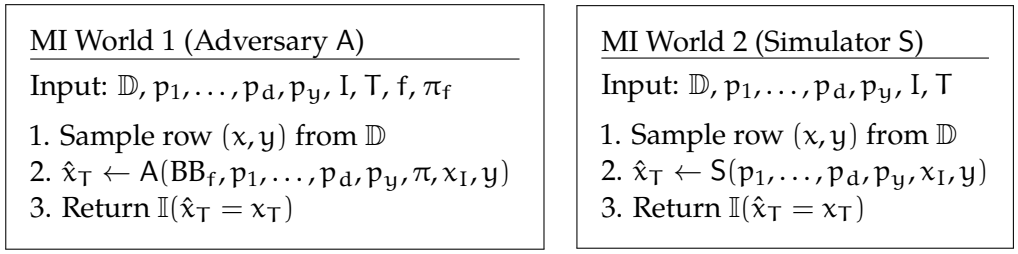


Figure 3.1: Model inversion game. In the first world, the adversary is given the marginals of $\mathbb{D}$, black-box access to the model f, f's performance statistics π, the response of a target sample y, and a subset of the sample's features x_I. In the second world, the simulator is given only the marginals and the same information about the target sample. In both worlds, the agent attempts to guess a subset of the target sample's features x_T, and wins when the game returns 1.

i.e.,

$$p_i(z_i) = \Pr_{x_i \leftarrow \mathbb{D}_i}[x_i = z_i] = \sum_{z \in X_{-i} \times Y} \Pr[x_1 = z_1, \ldots, x_d = z_d, y = z_y]$$

where the randomness in the last probability is taken over the draw of sample $(x_1, \ldots, x_d, y)$ from $\mathbb{D}$.

Let $f : X_1 \times \cdots \times X_d \mapsto Y$ be a deterministic *model* "trained on" D_t.[1] We say that an entity who can provide an input $x_1, \ldots, x_d$ to f and learn its output $y = f(x_1, \ldots, x_d)$ has *black-box access to* f. We model this type of access by providing a set $BB_f = \{(x, f(x)) \ : \ x \in X\}$. Let the *performance statistic* $\pi_f : X_1 \times \cdots \times X_d \times Y \mapsto \mathbb{R}$ be a function describing some aspect of f's ability to predict response from features.[2] Let I and T be disjoint subsets of $\{1, \ldots, d\}$, and $U = \{1, \ldots, d\} - (I \cup T)$. A *model inversion adversary* is a deterministic function that is given BB_f, π_f, $p_1, \ldots, p_d, p_y$, x_I, and y for some row $(x_1, \ldots, x_d, y)$, and attempts to guess the targeted features

[1] That f is trained on any particular data is not formally significant in what follows, but we make the distinction to clarify the connection our intended application.

[2] We discuss specific examples of π_f in later sections.

x_T. A *model inversion simulator* is a deterministic function that is given $p_1, \ldots, p_d, p_y, x_I$, and y, and attempts to guess the targeted features x_T.

The MI game is shown in Figure 3.1. The adversary A's world corresponds to the scenario described above, where A is given information about the model, the marginal distributions, and selected features of a row sampled from $\mathbb{D}$. The simulator S's world differs only in that S is not given any information about the model. Either agent wins when the game returns 1. Then the adversary's advantage then corresponds to the predictive power given by having information about f over just having information about the marginals of $\mathbb{D}$. This is made precise in Definition 17.

Definition 17. (Model Inversion Advantage) *An adversary* A*'s advantage against simulator* S *on* $\mathbb{D}, I, T, f, \pi_f$ *is defined as:*

$$\Pr\left[A(BB_f, p_1, \ldots, p_d, p_y, \pi_f, x_I, y) \text{ wins}\right] - \Pr\left[S(p_1, \ldots, p_d, p_y, x_I, y) \text{ wins}\right]$$

where the probability is taken over the sampling of (x, y) *from* $\mathbb{D}$.

Example 3.1. *Consider a setting where the two-dimensional feature space* $X = X_1 \times X_2 = \{0,1\}^2$*, the response space* $Y = \{0,1\}$*, and* $\mathbb{D}$ *has the pdf:*

$$p(x_1, x_2, y) = \begin{cases} \frac{1}{4} & \text{if } y = 1 - x_1 \\ 0 & \text{otherwise} \end{cases}$$

In other words, x_1 *and* y *are strongly correlated (i.e., they are functions of each other), and* x_2 *is independent of the other components. It is easily verified that the marginals are all uniform:*

$$p_1(x_1) = p_2(x_2) = \begin{cases} \frac{1}{2} & \text{if } x_{1,2} = 0 \\ \frac{1}{2} & \text{if } x_{1,2} = 1 \end{cases} \qquad p_y(y) = \begin{cases} \frac{1}{2} & \text{if } y = 0 \\ \frac{1}{2} & \text{if } y = 1 \end{cases}$$

Suppose that we are interested in a model f *that captures this distribution per-*

fectly:

$$f(x_1, x_2) = 1 - x_1$$

Then $BB_f = \{(0,0) \mapsto 1, (0,1) \mapsto 1, (1,0) \mapsto 0, (1,1) \mapsto 0\}$. *Furthermore, let* π_f *reflect the fact that* f *is a perfect predictor of* y *from* (x_1, x_2)*:*

$$\begin{aligned}\pi_f(\hat{x}_1, \hat{x}_2, \hat{y}) &= \Pr_{(x_1,x_2,y)\leftarrow\mathbb{D}}[y = \hat{y} \mid (x_1, x_2) = (\hat{x}_1, \hat{x}_2) \wedge f(\hat{x}_1, \hat{x}_2) = \hat{y}] \\ &= \begin{cases} 1 & \text{if } \hat{y} = f(\hat{x}_1, \hat{x}_2) \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

Now, suppose we want to play the MI game from Figure 3.1, where $T = \{1\}$ *and* $I = \{2\}$*, i.e., the adversary (or simulator) is given* x_2 *and* y*, and wishes to infer* x_1*. In this case, an intelligent adversary might determine from* BB_f *and* π_f *that* x_2 *is independent of* x_1 *and* y*, and* f *always gives the correct response, and decide to use the strategy:*

$$A(BB_f, p_1, p_2, p_y, \pi_f, x_2, y) = \arg\min_{x_1} |y - f(x_1, x_2)| = 1 - y$$

Likewise, an intelligent simulator would realize that from its perspective (i.e., given only the marginals), x_1 *is distributed uniformly, so it might as well return an arbitrary value:*

$$S(p_1, p_2, p_y, x_2, y) = 1$$

Now let us see what happens when this adversary and simulator play a round of the MI game. Say that in each player's game, the first step draws the row $(x_1 = 0, x_2 = 1, y = 1)$ *from* $\mathbb{D}$*. Then the adversary will infer* $\hat{x}_1 = 1 - y = 0$*, and the simulator will infer* 1*, leading the adversary's game to return* $\mathbb{I}(0 = 0) = 1$ *and the simulator's game to return* $\mathbb{I}(1 = 0) = 0$*. Thus, the adversary wins this round, and the simulator does not. In fact, the adversary has a considerable advantage in this case. We see that,*

$$\Pr\left[A(BB_f, p_1, \ldots, p_d, p_y, \pi_f, x_I, y) \text{ wins}\right] = \Pr\left[x_1 = 1 - y\right] = 1$$

On the other hand, the simulator's chance of winning is no better than an unbiased coin flip:

$$\Pr\left[\mathsf{S}(p_1, \ldots, p_d, p_y, x_I, y) \text{ wins}\right] = \Pr\left[x_1 = 1\right] = \frac{1}{2}$$

Thus, the adversary's advantage is $\frac{1}{2}$, which is the greatest possible advantage an adversary can achieve in this case against a Bayes- optimal simulator (this topic is discussed further in the following section).

Optimal Strategies

We now present strategies for both the MI simulator and adversary, and show that they are optimal. We first show that *strategies which minimize expected 0-1 loss maximize advantage* (Definition 17) when taken in expectation over the information x_I, y given to the adversary and simulator. This connection is demonstrated in Theorem 3.2. We then go on to show that our strategies for the simulator (Figure 3.2) and adversary (Figure 3.3) both minimize expected 0-1 loss (Theorems 3.3-3.6) whenever the adversary's initial beliefs correspond to a *maximum-entropy distribution*.

The notion of optimality for which we aim assumes that the simulator and adversary are Bayesian agents, who begin with an initial understanding of the "world" given by the indpendent marginal distributions of $\mathbb{D}$, update this understanding by performing a set of experiments on f using the instance information x_I, y, and eventually make a prediction about x_T. We show that our strategies embody the *Bayes decision rule*, and are thus optimal predictors for x_T in that they return an estimate $\hat{x}_T$ that minimizes the agent's *expected loss*, where we measure loss by misprediction of x_T. This corresponds to 0-1 loss:

$$\ell(x_T, \hat{x}_T) = \mathbb{I}(x_T \neq \hat{x}_T) = \begin{cases} 1 & \text{if } x_T \neq \hat{x}_T \\ 0 & \text{otherwise} \end{cases}$$

The expectation that we minimize is taken over a posterior distribution on x_T conditioned on the agent's observations, namely x_I and y:

$$\rho(\hat{x}_T \mid x_I, y) = \mathbf{E}_{\Pr[x_T \mid x_I, y]}[\ell(x_T, \hat{x}_T)] = \sum_{x_T \in X_T} \Pr[x_T \mid x_I, y]\, \ell(x_T, \hat{x}_T) \quad (3.1)$$

Notice that this is an appropriate objective for the MI adversary, as minimizing expected 0-1 loss on each instance corresponds in expectation to maximizing the adversary's chance of winning the MI game. This is established in Theorem 3.2.

Theorem 3.2. *An adversary who chooses $\hat{x}_T$ with the objective of minimizing the expected loss given in Equation 3.1 , i.e.:*

$$\hat{x}_T = \arg\min_{\hat{x}_T^*} \rho(v_T^* \mid x_I, y)$$

has maximal advantage (as given in Definition 17) in expectation over x_I, y.

Proof. For a fixed simulator S and distribution $\mathbb{D}$, Pr [S wins] is a constant, so the adversary maximizes her expected advantage by maximizing Pr [A wins] regardless of the choice of simulator. We show that a strategy that minimizes loss taken in expectation over observations x_I and y, i.e., $\mathbf{E}_{\Pr[x_I, y]}\mathbf{E}_{\Pr[x_T \mid x_I, y]}[\ell(x_T, \hat{x}_T)]$, maximizes Pr [A wins]:

$$\begin{aligned}
\mathbf{E}_{\Pr[x_I, y]}\mathbf{E}_{\Pr[x_T \mid x_I, y]}[\ell(x_T, \hat{x}_T)] &= \sum_{x_I, y} \Pr[x_I, y]\, \mathbf{E}_{\Pr[x_T \mid x_I, y]}[\ell(x_T, \hat{x}_T)] \\
&= \sum_{x_I, y} \Pr[x_I, y] \sum_{x_T} \mathbb{I}(\hat{x}_T \neq x_T) \Pr[x_T \mid x_I, y] \\
&= \sum_{x, y} \mathbb{I}(\hat{x}_T \neq x_T) \Pr[x, y] \\
&= \Pr[\text{A loses}]
\end{aligned}$$

Minimizing the probability of losing maximizes the probability of winning, so the proof is complete.
□

Strategy for the Simulator

Our goal in finding an optimal simulator strategy is to characterize an upper- bound on how predictable x_T is without being given access to the model f, which will allow us to determine how much additional predictability f lends an adversary via the experiments described later in Section 3.2. In this sense, our null hypothesis is that the simulator is as good at predicting x_T as the adversary, who has been informed of the model. Theorem 3.2 gives us an approach for designing an optimal strategy. However, computing the conditional expectation $\rho(\hat{x}_T \mid x_I, y)$ requires reasoning about the prior distribution $\mathbb{D}$, and the simulator is only given access to indpendent marginals of $\mathbb{D}$.

To derive a best estimate for $\mathbb{D}$ that incorporates this information, we apply the *principle of maximum entropy*. This principle states that when selecting a distribution, we should select the one that makes the fewest assumptions (i.e., is the *least biased*) subject to the constraints imposed by our previous knowledge. As argued by Jaynes (Jaynes, 1982), this distribution has maximum Shannon entropy among those that satisfy the given constraints. For the simulator, the relevant constraints require that the distribution has marginals that match those given in the MI game, $p_1, \ldots, p_d, p_y$. Theorem 3.3 shows that the maximum entropy prior in this case corresponds to the independent product of those marginals.

Theorem 3.3. *The maximum entropy distribution $\mathbb{D}_{p_1,\ldots,p_d}$ over X having first-order marginals $p_1, \ldots, p_d$ has the probability distribution function:*

$$p(x_1, \ldots, x_d) = \prod_{i=1}^{d} p_i(x_i)$$

In other words, it is the independent product of $p_1, \ldots, p_d$ *evaluated component-wise at* $x_1, \ldots, x_d \in X$.

Proof. First we verify that that p has first-order marginals distributed according to $p_1, \ldots, p_d$:

$$\begin{aligned} \Pr[x_i = z_i] &= \sum_{z' \in \mathbf{X}_{-i}} \Pr[x_1 = z'_1, \ldots, x_i = z_i, \ldots, x_d = z'_d] \\ &= p_i(z_i) \sum_{z' \in \mathbf{X}_{-i}} p_i(z'_1) \cdots p_i(z'_d) \\ &= p_i(z_i) \end{aligned} \tag{3.2}$$

The last step follows from the law of total probability, as we sum the marginal on X_{-i} over its entire domain.

Now we show that p has maximum entropy among distributions whose marginals are given by $p_1, \ldots, p_d$. In the following, we treat the marginals $p_i(x_i)$ evaluated as constants, as they are known in advance. The p.d.f. for this distribution corresponds to the solution for the following constrainted maximization problem:

$$\begin{aligned} & p^* = \arg\max_p - \textstyle\sum_{x \in \mathbf{X}} p(x) \log(p(x)) \\ \textit{subject to} \quad & p_i(x_i) = \textstyle\sum_{x' \in \mathbf{X}_{-i}} p(x'_1, \ldots, x_i, \ldots, x'_d), \forall 1 \leqslant i \leqslant d \\ & 1 = \textstyle\sum_{x \in \mathbf{X}} p(x) \end{aligned}$$

In the following, assume an ordering on the elements of each domain $X_1, \ldots, X_d$, and let $X_{i,j}$ refer to the j*th* element of the i*th* feature domain. We will construct a Lagrangian for this problem, encoding the marginal constraints with a series of functions c_{ij}:

$$c_{ij}(p) = p_i(X_{i,j}) - \sum_{x \in \mathbf{X}_{-i}} p(x_1, \ldots, X_{i,j}, \ldots, x_d)$$

for $1 \leqslant i \leqslant d, 1 \leqslant j \leqslant |X_i|$. The Lagrangian for this problem is:

$$L(p, \lambda) = -\sum_{x \in \mathbf{X}} p(x) \log(p(x)) - \lambda_0 \left(\sum_{x \in \mathbf{X}} p(x) - 1 \right) - \sum_{i=1}^{d} \sum_{j=1}^{|X_i|} \lambda_{i,j} c_{ij}(p)$$

Then we have:

$$\frac{\partial L}{\partial p(X_{1,i_1}, \ldots, X_{d,i_d})} = -\left(1 + \log(p(X_{1,i_1}, \ldots, X_{d,i_d})) + \lambda_0 + \sum_{j=1}^{d} \lambda_{j,i_j} \right)$$

Setting $\frac{\partial L}{\partial p(X_{1,i_1}, \ldots, X_{d,i_d})} = 0$ and solving the system obtained by substituting each valuation in X, we see that:

$$p(X_{1,i_1}, \ldots, X_{d,i_d}) = e^{-\left(1 + \lambda_0 + \sum_{j=1}^{d} \lambda_{j,i_j}\right)} \tag{3.3}$$

We will now solve for λ_0 and each $\lambda_{i,j}$ by setting $\frac{\partial L}{\partial \lambda_0} = 0$ as well as $\frac{\partial L}{\partial \lambda_{i,j}} = 0$, and substituting in Equation 3.3. We have:

$$\frac{\partial L}{\partial \lambda_0} = 1 - \sum_{x \in X} p(x) = 1 - \sum_{i_1=1}^{|X_1|} \cdots \sum_{i_d=1}^{|X_d|} e^{-\left(1 + \lambda_0 + \sum_{j=1}^{d} \lambda_{j,i_j}\right)} \tag{3.4}$$

$$\frac{\partial L}{\partial \lambda_{i,j}} = -c_{ij}(p) = \sum_{X_{1,k_1}, \ldots, X_{d,k_d} \in X_{-i}} e^{-\left(1 + \lambda_0 + \lambda_{ij} + \sum_{\substack{h=1 \\ h \neq i}}^{d} \lambda_{h,k_h}\right)} - p_i(X_{i,j}) \tag{3.5}$$

Solving this system, we arrive at:

$$\lambda_0 = \log(1) \qquad\qquad \lambda_{ij} = \log \frac{1}{p_i(X_{i,j})}$$

Substituting these solutions back into Equation 3.3 should give us an expression corresponding to the product of the independent marginals, if

the theorem is correct. We see that it holds:

$$\begin{aligned} p(X_{1,i_1}, \ldots, X_{d,i_d}) &= e^{-1-\lambda_0-\sum_{j=1}^{d}\lambda_{j,i_j}} = e^{-1}e^{-\log(1)}e^{-\sum_{j=1}^{d}\log\frac{1}{p_j(X_{j,i_j})}} \\ &= e^{-\sum_{j=1}^{d}\log\frac{1}{p_j(X_{j,i_j})}} \\ &= \prod_{j=1}^{d} p_j(X_{j,i_j}) \end{aligned}$$

For the sake of completeness, we can also verify that these solutions satisfy Equations 3.4 and 3.5. Plugging them into Equation 3.4, we have:

$$\begin{aligned} \sum_{i_1=1}^{|X_1|}\cdots\sum_{i_d=1}^{|X_d|} e^{-\left(1+\lambda_0+\sum_{j=1}^{d}\lambda_{j,i_j}\right)} &= \sum_{i_1=1}^{|X_1|}\cdots\sum_{i_d=1}^{|X_d|}\prod_{j=1}^{d} p_j(X_{j,i_j}) \\ &= \sum_{i_1=1}^{|X_1|} p_1(X_{1,i_1})\sum_{i_2=1}^{|X_2|}\cdots\sum_{i_d=1}^{|X_d|}\prod_{j=2}^{d} p_j(X_{j,i_j}) \\ &= \sum_{i_2=1}^{|X_2|}\cdots\sum_{i_d=1}^{|X_d|}\prod_{j=2}^{d} p_j(X_{j,i_j}) \\ &\cdots \\ &= 1 \end{aligned}$$

Here the intermediate steps that have been elided all follow from the law of total probability, as each sums one of the marginals over its full domain.

Lastly, Equation 3.5 follows from the fact that $p_i(X_{i,j})$ is a marginal of $\prod_{j=1}^{d} p_j(X_{j,i_j})$, as demonstrated in Equation 3.2.
□

Figure 3.2 shows a strategy for the simulator, who is only given x_I, y, and the marginal priors. This strategy follows the general form outlined in Theorem 3.2, where the adversary computes the posterior distribution $\Pr[x_T \mid x_I, y]$ by conditioning on the prior from Theorem 3.3. Because this

Strategy for MI simulator S

Input: $p_1, \ldots, p_d, p_y, x_I, y$

1. Let $\hat{x}_T \leftarrow \arg\max_{v_T} \prod_{i \in T} p_i(v_{T_i})$
2. Return $(\hat{x}_{T_1}, \ldots, \hat{x}_{T_{|T|}})$

Figure 3.2: Optimal strategy for the MI simulator S. This strategy selects the most likely value for each target feature according to the marginals $p_1, \ldots, p_d$, and is shown to minimize the adversary's expected misprediction rate in Theorem 3.4.

prior is feature-wise independent, x_I and y yield no useful information about x_T, so conditioning amounts to selecting the independent marginal for x_T. Thus, this strategy always predicts the mode for x_T from the independent marginal. Theorem 3.4 shows that this is the optimal strategy.

Theorem 3.4. *The strategy for the simulator given in Figure 3.2 minimizes expected 0-1 loss given* x_I, y*, i.e.,*

$$\arg\max_{v_T} \prod_{i \in T} p_i(v_{T_i}) = \arg\min_{v_T} \mathbf{E}_{\Pr[x_T \mid x_I, y]}[\ell(x_T, v_T)]$$

Proof. By our use of the maximal-entropy prior and Theorem 3.3, we have that $p(x_1, \ldots, x_d, y) = p_y(y) \prod_{i=1}^{d} p_i(x_i)$. Then recalling that $U = \{1, \ldots, d\} - (T \cup I)$,

$$\begin{aligned}
\Pr[x_T \mid x_I, y] &= \frac{\sum_{x'_U \in X_U} p(x_1, \ldots, x'_U, \ldots x_d, y)}{\Pr[x_I, y]} \\
&= \frac{\prod_{i \in T} p_i(x_i) \prod_{i \in I} p_i(x_i) p_y(y) \sum_{x'_U \in X_U} \Pr[x'_U]}{\prod_{i \in I} p_i(x_i) p_y(y)} \\
&= \prod_{i \in T} p_i(x_i)
\end{aligned}$$

Then let $v_T^* = \arg\max_{v_T} \prod_{i \in T} p_i(v_{T_i})$. We have,

$$\begin{aligned}
\mathbf{E}_{\Pr[x_T \mid x_I, y]}[\ell(x_T, v_T)] &= \Pr[x_T \neq v_T^* \mid x_I, y] \\
&= 1 - \Pr[x_T = v_T^* \mid x_I, y] \\
&= 1 - \prod_{i \in T} p_i(v_{T_i}^*)
\end{aligned}$$

This last quantity is minimized by our choice of v_T^*.
□

Strategy for the Adversary

We move now to a strategy for the adversary. Like the simulator, the adversary is given first-order marginals $p_1, \ldots, p_d$, along with two additional pieces of information:

- BB_f: black-box information about the model f's behavior.
- π_f: statistics regarding the model f's ability to predict the response feature y given x, when (x, y) is sampled from $\mathbb{D}$.

Recall that we model BB_f with the set $\{(x, f(x)) \mid x \in X\}$, so the adversary can see all input-output pairs admitted by f[3] For π_f, we assume that the adversary is given probabilities for each type of error that arises when f is evaluated at a given point x, i.e., $\Pr[y \mid f(x) = y']$. However, f is deterministic and the adversary has BB_f, so she can always look up $(x, f(x))$ pairs, which allows us to state this probability more directly:

$$\pi_f(x_1, \ldots, x_d, y) = \Pr[y \mid x_1, \ldots, x_d]$$

In other words, π_f gives the adversary information about the conditional probability of the response feature Y given a value x for the remaining

[3]We make no assumptions about the adversary's computational power, i.e., she is allowed to use as much time and space as necessary as long as she eventually terminates.

features. This adds a new constraint to the adversary's knowledge about the prior distribution $\mathbb{D}$, which yields a different maximum-entropy prior from the one used by the simulator. The new prior is given in Theorem 3.5, Equation 3.8.[4]

Theorem 3.5. *For a given function* $f : X_1 \times \cdots \times X_{d-1} \mapsto X_d$*, the maximum-entropy distribution* $\mathbb{D}_{p_1,\ldots,p_d,\pi_f}$ *satisfying the following constraints on its p.d.f.* p*:*

$$p_i(x_i) = \sum_{x' \in X_{-i}} p(x'_1, \ldots, x_i, \ldots, x'_d), \forall 1 \leqslant i \leqslant d \tag{3.6}$$

$$\pi_f(x_1, \ldots, x_d) = \Pr[x_d \mid x_1, \ldots, x_{d-1}] = \frac{p(x_1, \ldots, x_d)}{p_{-d}(x_1, \ldots, x_{d-1})} \tag{3.7}$$

where p_{-d} *is the* $d-1$*th-order marginal over* X_{-d}*, is given by:*

$$p(x_1, \ldots, x_d) = \pi_f(x_1, \ldots, x_d) \prod_{i=1}^{d-1} p_i(x_i) \tag{3.8}$$

Proof. First, we will verify that the given p.d.f. satisfies Equations 3.6 and 3.7. Starting with 3.7, as it is needed to prove 3.6:

$$\begin{aligned}
\frac{p(x_1, \ldots, x_d)}{\sum_{x'_d \in X_d} p(x_1, \ldots, x_{d-1}, x'_d)} &= \frac{\pi_f(x_1, \ldots, x_d) \prod_{i=1}^{d-1} p_i(x_i)}{\sum_{x'_d \in X_d} \pi_f(x_1, \ldots, x_{d-1}, x'_d) \prod_{i=1}^{d-1} p_i(x_i)} \\
&= \frac{\pi_f(x_1, \ldots, x_d)}{\sum_{x'_d \in X_d} \pi_f(x_1, \ldots, x_{d-1}, x'_d)} \\
&= \pi_f(x_1, \ldots, x_d)
\end{aligned}$$

[4]Note that in Theorem 3.5, we assume for notational simplicity that f is a function from $X_1, \ldots, X_{d-1}$ to X_d, whereas when we invoke the theorem, we assume f takes $X_1, \ldots, X_d$ to Y. The result is the same, but the subscripts will change.

The last step follows because we assume that π_f is a distribution on X_d. Now for 3.6, starting with the case $i \neq d$:

$$\begin{aligned}
\sum_{x' \in X_{-i}} p(x'_1, \ldots, x_i, \ldots, x'_d) &= p_i(x_i) \sum_{x' \in X_{-i}} \pi_f(x'_1, \ldots, x_i, \ldots, x'_d) \prod_{\substack{j=1 \\ j \neq i}}^{d-1} p_j(x'_j) \\
&= p_i(x_i) \sum_{x' \in X_{-i}} \pi_f(x'_1, \ldots, x_i, \ldots, x'_d) \\
&= p_i(x_i) \sum_{x' \in X_{-1}} \frac{p(x'_1, \ldots, x_i, \ldots, x'_d)}{\sum_{x''_d \in X_d} p(x'_1, \ldots, x_i, \ldots, x'_{d-1}, x''_d)} \\
&= p_i(x_i)
\end{aligned}$$

The second step follows because the sum marginalizes the factor $\prod_{\substack{j=1 \\ j \neq i}}^{d-1} p_j(x'_j)$ over all values, and becomes 1 because each p_j is a distribution. Now when $i = d$:

$$\begin{aligned}
\sum_{x' \in X_{-d}} p(x'_1, \ldots, x_{d-1}, x_d) &= \sum_{x' \in X_{-d}} \pi_f(x'_1, \ldots, x'_{d-1}, x_d) \prod_{j=1}^{d-1} p_j(x'_j) \\
&= \sum_{x' \in X_{-d}} \frac{p(x'_1, \ldots, x'_{d-1}, x_d)}{\sum_{x''_d \in X_d} p(x'_1, \ldots, x'_{d-1}, x''_d)} \\
&= \sum_{x' \in X_{-d}} \frac{\Pr\left[x'_1, \ldots, x'_{d-1}, x_d\right]}{\Pr\left[x'_1, \ldots, x'_{d-1}\right]} \\
&= p_d(x_d)
\end{aligned}$$

Now we know that if the p.d.f. for the maximum-entropy prior takes the form of Equation 3.8, then it satisfies all the needed constraints. We now show that $\mathbb{D}_{p_1, \ldots, p_d, \pi_f}$ does indeed take this form. The Lagrangian is:

$$L(p, \lambda) = -\sum_{x \in \mathbf{X}} p(x) \log(p(x)) - \lambda_0 \left(\sum_{x \in \mathbf{X}} p(x) - 1 \right)$$

$$
\begin{aligned}
&- \sum_{i=1}^{d} \sum_{j=1}^{|X_i|} \lambda_{i,j} \left(p_i(X_{i,j}) - \sum_{x \in X_{-i}} p(x_1, \ldots, X_{i,j}, \ldots, x_d) \right) \\
&- \sum_{x \in X} \lambda_{x_1,\ldots,x_d} \left(p(x) - \pi_f(x) p_{-d}(x_1, \ldots, x_{d-1}) \right)
\end{aligned}
$$

Setting $\frac{\partial L}{\partial p(X_{1,i_1},\ldots,X_{d,i_d})} = 0$ and solving for $p(X_{1,i_1}, \ldots, X_{d,i_d})$, we have:

$$
p(X_{1,i_1}, \ldots, X_{d,i_d}) = e^{-1-\lambda_0-\lambda_{X_{1,i_1},\ldots,X_{d,i_d}} - \sum_{j=1}^{d} \lambda_{j,i_j}} \tag{3.9}
$$

Plugging this into the constraints given in Equations 3.6 and 3.7, we arrive at the following solutions for the Lagrange coefficients:

$$
\lambda_0 = \log e^{-1} \quad \lambda_{X_{1,i_1},\ldots,X_{d,i_d}} = \log \frac{1}{\pi_f(X_{1,i_1},\ldots,X_{d,i_d})} \quad \lambda_{ij} = \begin{cases} \log \frac{1}{p_i(X_{i,j})} & i < d \\ \log(1) & i = d \end{cases}
$$

Now we show that substituting these coefficients into the solution for $p(x)$ given in Equation 3.9 gives the required form:

$$
\begin{aligned}
p(X_{1,i_1}, \ldots, X_{d,i_d}) &= e^{-1-\lambda_0-\lambda_{X_{1,i_1},\ldots,X_{d,i_d}} - \sum_{j=1}^{d} \lambda_{j,i_j}} \\
&= e e^{-\log \frac{1}{e}} e^{-\log \frac{1}{\pi_f(X_{1,i_1},\ldots,X_{d,i_d})}} e^{-\sum_{j=1}^{d-1} \log \frac{1}{p_i(X_{j,i_j})}} \\
&= \pi_f(X_{1,i_1}, \ldots, X_{d,i_d}) e^{-\sum_{j=1}^{d-1} \log \frac{1}{p_i(X_{j,i_j})}} \\
&= \pi_f(X_{1,i_1}, \ldots, X_{d,i_d}) \prod_{j=1}^{d-1} p_i(X_{j,i_j})
\end{aligned}
$$

As demonstrated earlier, this expression satisfies all of the constraints we have have on $p(x)$, so this completes the proof.
□

With the new prior given in Theorem 3.5, the adversary follows the general approach given in Theorem 3.2 of calculating the posterior and finding

Strategy for MI adversary A

Input: $p_1, \ldots, p_d, p_y, x_I, y, f, \pi_f$

1. Find the feasible set $\hat{X} \subseteq X$:

$$\hat{X} = \{x' \in X \mid x'_I = x_I\}$$

1. Let $\hat{x}_T \leftarrow \arg\max_{v_T} \sum_{x' \in \hat{X} : x'_T = v_T} \pi_f(x'_1, \ldots, x'_d, y) \prod_{1 \leqslant i \leqslant d} p_i(x'_i)$
2. Return $\hat{x}_T$

Figure 3.3: Optimal strategy for the MI adversary A. This strategy selects among samples matching x_I the most likely value for x_T according to the marginals $p_1, \ldots, p_d$ and performance statistics π_f, and is shown to minimize the adversary's expected misprediction rate in Theorem 3.6.

its mode. However, finding the mode is now slightly more involved. Candidate samples that match what is known about the target sample are run through π_f[5] and its output is used with the first-order marginals to weight each candidate. The configuration of target features with the greatest weight, computed by marginalizing the other unknown attributes, is returned as the mode. The strategy is shown in Figure 3.3, and its optimality is proved in Theorem 3.6.

Theorem 3.6. *The strategy for the adversary given in Figure 3.3 minimizes expected 0-1 loss given* x_I, y*, i.e.,*

$$\arg\min_{v_T} \mathbf{E}_{\Pr[x_T \mid x_I, y]}[\ell(x_T, v_T)] = \arg\max_{v_T} \sum_{x' \in \hat{X} : x'_T = v_T} \pi_f(x'_1, \ldots, x'_d, y) \prod_{1 \leqslant i \leqslant d} p_i(x'_i)$$

[5]For practical reasons, this usually involves consulting BB_f, i.e., evaluating f on the candidate samples.

Proof. By the prior given in Theorem 3.5, we have that

$$p(x_1, \ldots, x_d, y) = \pi_f(x_1, \ldots, x_d, y) \prod_{i=1}^{d} p_i(x_i)$$

Then recalling that $U = \{1, \ldots, d\} - (T \cup I)$,

$$\begin{aligned} \Pr[x_T \mid x_I, y] \propto \Pr[x_T, x_I, y] &= \sum_{x'_U \in X_U} p(x_1, \ldots, x'_U, \ldots, x_d, y) \\ &= \sum_{x' \in \hat{X} : x'_T = x_T} \pi_f(x'_1, \ldots, x'_d, y) \prod_{i=1}^{d} p_i(x'_i) \end{aligned}$$

The last step follows because $\hat{X}$ will contain all configurations of X_U. Now let $v_T^* = \arg\max_{v_T} \sum_{x' \in \hat{X} : x'_T = v_T} \pi_f(x'_1, \ldots, x'_d, y) \prod_{i=1}^{d} p_i(x'_i)$, as it is chosen in Figure 3.3. v_T^* minimizes $\mathbf{E}_{\Pr[x_T \mid x_I, y]}[\ell(x_T, v_T)]$ by an identical derivation to the one given in the proof of Theorem 3.4.
□

Discussion. We have argued that the strategy in Figure 3.3 minimizes the expected misclassification rate on the maximum-entropy prior given the available information (the model and marginals). However, it is not hard to specify distributions for which the marginals convey little useful information, so the expected misclassification rate diverges substantially from the true rate. In these cases, the strategy may perform poorly, and more background information is needed to accurately predict model inputs.

There is also the possibility that the model itself does not contain enough useful information about the correlation between certain input features and the output. For illustrative purposes, consider a model taking one input attribute, that discards nearly all information about that attribute,

e.g., it performs an equality comparison with a fixed constant c:

$$f(x) = \begin{cases} 1 & \text{if } x = c \\ 0 & \text{otherwise} \end{cases}$$

Suppose that the input feature x comes from some large domain with 2^m elements, and that $\mathbb{D}$ is uniform on that domain. Then the adversary's advantage is negligible, $1/(2^m - 1)$: the simulator, guessing according to the marginal, will win with probability $1/2^m$, whereas the adversary will win with probability $1/2^m + 1/(2^m - 1)$. In general, understanding how susceptible a function is to model inversion *a priori* is challenging, and warrants further consideration in future work. Thus, determining how well a model allows one to predict sensitive inputs generally requires further analysis. This topic is explored in the context of a real model in the next section.

Another important matter is the complexity of the strategies given in Figures 3.2 and 3.3. Depending on how the marginal distributions are presented, the simulator's strategy (Figure 3.2) might be efficient, although in general it requires time $O(|X_T|)$. The adversary's strategy (Figure 3.3) is worse yet: the set $\hat{X}$ has size $|X_{\bar{I}}|$, and the adversary must iterate through each element, apply π_f, and take the product of the marginals. Thus, this strategy is only feasible when $|X_{\bar{I}}| = |X_{U \cup T}|$. Fortunately, as discussed in the next section, this is the case in the application that we study. In cases where this is not true, approximation techniques are necessary. In Chapter 5, Section 5.1, we briefly discuss using model inversion to recover image data, and describe one such approximation approach that relies on gradient descent to address the complexity issue.

3.2 A Case Study: Personalized Medicine

We now turn to a study of how MI can be used as an application-specific measure of inference vulnerability to illustrate the dynamic between privacy and utility, and the effect that differential privacy has on this dynamic. We use an application from the field of *personalized medicine*—pharmacogenetic warfarin dosing—as a vehicle to drive this study, and we attempt to find empirical evidence that can provide answers to the following questions:

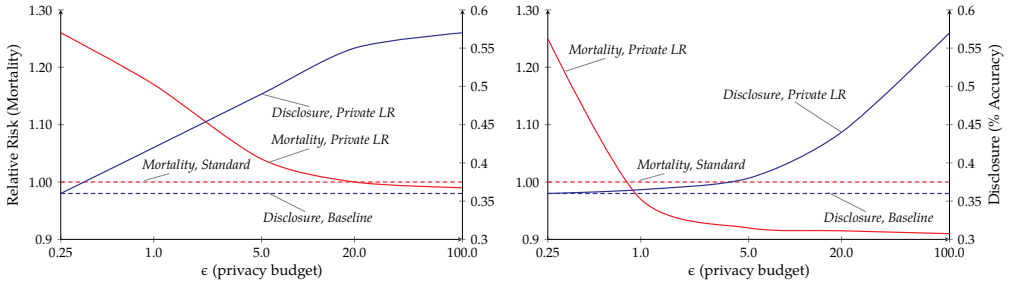
1. Do pharmacogenetic warfarin dosing models pose a threat to the genomic privacy of the individuals who are involved in this application, i.e., individuals who either contribute their data to help build the models, or those who eventually use them?

2. If these models do pose a threat, can differential privacy be successfully applied to help mitigate the threat? In short, is there a privacy budget (i.e., ϵ setting) that allows state-of-the-art DP regression techniques to prevent the inference attack described in Section 3.1?

3. If we can configure a DP mechnism to successfully prevent MI in this case, then does it stand in the way of utility—does it introduce an unacceptable increase in the likelihood of adverse health outcomes for patients who use the DP warfarin dosing models?

To summarize our findings, we find that warfarin dosing models pose a measurable threat to the genomic privacy of patients, giving the adversary a substantial (22%) advantage over the simulator in one case. While DP can negate this advantage when $\epsilon \leqslant 1$, privacy budgets in this range also negate the benefits of pharmacogenetic dosing, and in fact make certain adverse events more likely than if genotype were ignored entirely.

This situation is shown in Figure 3.4(a), against a hypothetical rendition of what "acceptable" results might look like in Figure 3.4(b). In these



(a) Actual results

(b) "Acceptable" results

Figure 3.4: Key results from our end-to-end study (a) and hypothetical "acceptable" results (b). The blue curves correspond to MI disclosure risk of the VKORC1 genotype, with the dashed line corresponding to the simulator's performance and the solid line corresponding to that of the adversary. The red curves correspond to patients' risk (relative to a fixed-dose protocol) for mortality when given initial warfarin doses using a DP model at the specified privacy budget level (note that smaller budgets imply more privacy). The dashed line corresponds to the risk using the fixed-dose model, and the solid line corresponds to the use of the private model.

results, we measure *1)* an adversary's ability to predict a patient's VKORC1 genotype from a differentially-private linear regression model, and *2)* patients' mortality risk when given warfarin doses using private models *relative to* the mortality risk of standard dosing practices. Notice that in the acceptable case, there is a range of privacy budgets around $\epsilon = 1$ where the private dosing model begins to provide benefits over standard dosing methods, but the MI adversary's advantage remains small—ideally, not statistically-significant. This range is not present in the actual results, where the MI adversary's advantage grows quickly as ϵ increases, and the model fails to provide benefits—or even break even with standard dosing practices—until ϵ is fairly large. This, in short, tells us that differential privacy is not the right countermeasure to use against inference attacks in this particular setting.

Application Background

Warfarin, also known in the United States by the brand name Coumadin, is a widely-prescribed anticoagulant medication. It is used to treat patients suffering from cardiovasvular problems, including atrial fibrillation (a type of irregular heart beat) and heart valve replacement. By reducing the tendency of blood to clot, at appropriate dosages it can reduce risk of adverse clotting events, particularly stroke. Unfortunately, warfarin is also very difficult to dose: proper dosages can differ by an order of magnitude between patients, and this has led to warfarin's status as one of the leading causes of drug-related adverse events in the United States (Kim et al., 2009). Underestimating the dose can result in failure to prevent the condition the drug was prescribed to treat. Overestimating the dose can, just as seriously, lead to uncontrolled bleeding events because the drug interferes with clotting. Because of these risks, in existing clinical practice patients starting on warfarin are given a fixed initial dose but then must visit a clinic many times over the first few weeks or months of treatment in order to determine the correct dosage which gives the desired therapeutic effect.

Stable dose is assessed clinically by measuring the time it takes for blood to clot, called prothrombin time. This measure is standardized between different manufacturers as an international normalized ratio (INR). Based on the patient's indication for warfarin, a clinician determines a target INR range. After the fixed initial dose, later doses are modified until the patient's INR is within the desired range and maintained at that level. INR in the absence of anticoagulation therapy is approximately 1, while the desired INR for most patients in anticoagulation therapy is in the range 2–3 (Brace, 2001).

Genetic variability among patients is known to play an important role in determining the proper dose of warfarin (Kamali and Wynne, 2010). Polymorphisms in two genes, VKORC1 and CYP2C9, are associated with the mechanism with which the body metabolizes the drug, which in

turn affects the dose required to reach a given concentration in the blood. Warfarin works by interfering with the body's ability to recycle vitamin K, which is used to regulate blood coagulation. VKORC1, part of the vitamin K epoxide reductase complex, is a component of the vitamin K cycle. CYP2C9 encodes for a variant of cytochrome P450, a family of proteins which oxidize a variety of medications.

Taken together with age and height, Sconce *et al.* (Sconce et al., 2005) demonstrated that CYP2C9 and VKORC1 account for 54% of the total warfarin dose requirement variability. In turn, a large literature (over 50 papers as of early 2013) has sought pharmacogenetic algorithms that predict proper dose by taking advantage of patient genetic markers for CYP2C9 and VKORC1, together with demographic information and clinical history (e.g., current medications). These typically involve learning a simple predictive model of stable dose from previously obtained outcomes. We focus on the IWPC algorithm (International Warfarin Pharmacogenetic Consortium, 2009), a study resulting in production of a linear regression model that, when used to predict the initial dosage, has been shown to provide improved outcomes in simulated clinical trials using the IWPC dataset discussed below. Interestingly, linear regression performed as well or better than a wide variety of other, more complex machine learning techniques. Some pharmacogenetic algorithms for warfarin are currently also undergoing (real) clinical trials (National Heart, Lung, and Blood Institute).

Dataset

The IWPC (International Warfarin Pharmacogenetic Consortium, 2009) collected data on patients who were prescribed warfarin from 21 sites in 9 countries on 4 continents. The data was curated by staff at the Pharmacogenomics Knowledge Base (Whirl-Carrillo et al., 2012), and each site obtained informed consent to use de-identified data from patients prior to

the study. Because the dataset contains no protected health information, and the Pharmacogenomics Knowledge Base has since made the dataset publicly available for research purposes, it is exempt from institutional review board review. However, the type of data contained in the IWPC dataset is equivalent to many other medical datasets that have not been released publicly (Sconce et al., 2005; Hamberg et al., 2007; Anderson et al., 2007; Carlquist et al., 2006), and are considered private.

Each patient was genotyped for at least one SNP in VKORC1, and for variants of CYP2C9. Since each person has two copies of each gene, there are several combinations of variants possible. Following the IWPC paper (International Warfarin Pharmacogenetic Consortium, 2009), we represent VKORC1 polymorphisms by single nucleotide polymorphism (SNP) rs9923231, which is either G (common variant) or A (uncommon variant), resulting in three combinations G/G, A/G, or A/A. Similarly, CYP2C9 variants are *1 (most common), *2, or *3, resulting in 6 combinations. In addition, other information such as age, height, weight, race, and other medications was collected. The outcome variable is the stable therapeutic dose of warfarin, defined as the steady-state dose that led to stable anticoagulation levels. The patients in our dataset were restricted to those with target INR in the range 2–3 (the vast majority of patients), as is standard practice with most studies of warfarin dosing efficacy (Fusaro et al., 2013; Anderson et al., 2007).

We divided the data into two cohorts based on those used in IWPC (International Warfarin Pharmacogenetic Consortium, 2009). The first (training) cohort was used to build a set of pharmacogenetic dosing algorithms. The second (validation) cohort was used to test privacy attacks as well as draw samples for the clinical simulations. To make the data suitable for regression we removed all patients missing at least one feature, normalized the data to the range [-1,1], and converted all nominal attributes into binary-valued numeric attributes using one-hot encoding. Our eventual

training cohort consisted of 3966 patients, and our validation cohort of 853 patients, and corresponds to the same training-validation split used by IWPC (but without the missing values used in the IWPC split).

Adversary Model

We assume an adversary with the basic capabilities outlined in Section 3.1. We assume that the model is linear, and was trained over a set of samples (or *rows*) D drawn i.i.d. from $\mathbb{D}$. In this setting, $X_T = \{\text{VKORC1}, \text{CYP2C9}\}$, and the response feature Y corresponds to warfarin dose. We study two configurations of features known to the adversary, x_I:

- *Basic demographics*: a subset of demographic features contained in the IWPC dataset, including age, race, height, and weight (denoted $x_{\text{age}}, x_{\text{race}}, \ldots$). All of the numeric features were binned by the authors of the original IWPC model (International Warfarin Pharmacogenetic Consortium, 2009).
- *All background*: all of features except CYP2C9 or VKORC1 genotype.

We compare the adversary's performance against two strategies:

- *Simulator*: the "baseline" simulator corresponds to the strategy given in Figure 3.2 based on predicting the mode of the marginal prior for each genotype feature.
- *Ideal predictor*: in order to determine how how well one can predict these genotypes in an *ideal* setting, we compare against a multinomial logistic regression model for each genotype from the IWPC dataset. This allows us to compare the performance of the adversary against "best-possible" results achieved using standard linear machine learning techniques.

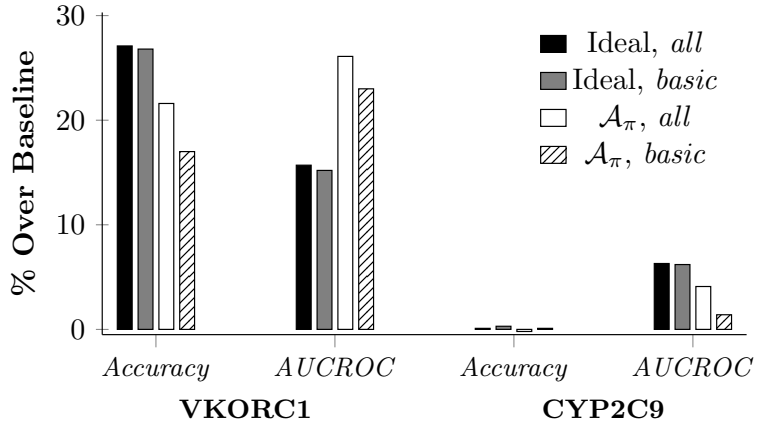


Figure 3.5: Model inversion performance, as improvement over baseline guessing from marginals, given a linear model derived from the training data. Available background information specified by *all* and *basic* as discussed above.

Empirical Analysis of Model Inversion

Standard Linear Regression

A linear regression model assumes that the response is a linear function of the attributes, i.e., there exists a coefficient vector $w \in \mathbb{R}^d$ and random *residual error* δ such that $y = w^T x + w_0 + \delta$ for some bias term b. A linear regression model f_L is then an estimate $(\hat{w}, \hat{b})$ of w and the bias term, which operates as: $f_L(x) = \hat{b} + \hat{w}^T x$. It is typical to assume that δ has a fixed Gaussian distribution $\mathcal{N}(0, \sigma^2)$ for some variance σ. Most regression software estimates σ^2 empirically from training data, so it is often published alongside a linear regression model. Using this the adversary can derive an estimate of π,

$$\hat{\pi}(y, y') = \mathbf{Pr}_{\mathcal{N}(0,\sigma^2)}[y - y']$$

All of our results in Section 3.1 assume that the features are discrete, whereas linear models operate more generally over continuous features.

Fortunately, in the case we consider, the continuous variables have natural discretizations, as they correspond to attributes such as age and weight. Warfarin doses discretize on half-milligram increments, as finer increments are not commonly manufactured.

Results. To evaluate adversary A, we split the IWPC dataset into a training and validation set, D_T and D_V respectively, use D_T to derive a linear model f, and then run A on every row in D_T with either of the two background information types (*all* or *basic*) to predict both genotypes. We repeated this for the simulator (*baseline*) and the ideal predictor (*ideal*).

We measure performance both in terms of *accuracy*, which is the percentage of samples for which the algorithm correctly predicted genotype, and *AUCROC*, which is the multi-class area under the ROC curve defined by Hand and Till (Hand and Till, 2001). While accuracy is generally easier to interpret, it can give a misleading characterization of predictive ability for *skewed* distributions—if the predicted attribute takes a particular value in 75% of the samples, then a trivial algorithm can easily obtain 75% accuracy by always guessing this value. AUCROC does not suffer this limitation, and so gives a more balanced characterization of how well an algorithm predicts both common and rare values.

The results are given in Figure 3.5, which shows the performance of A and "ideal" multinomial regression predicting VKORC1 and CYP2C9 on the training set. The numbers are given relative to the baseline performance obtained by the simulator–36% accuracy on VKORC1, 75% accuracy on CYP2C9, and 0.5 AUCROC for both genotypes. We see that A comes close to ideal accuracy on VKORC1 (5% less accurate with all background information), and actually exceeds the ideal predictor in terms of AUCROC. This means that A does a better job predicting *rare* genotypes than the ideal model, but does slightly worse overall, and may be a result of the ideal model avoiding overfitting to uncommon data points.

The results for CYP2C9 are quite different. Neither A or the ideal strategy were able to predict this genotype more accurately than baseline. This indicates that CYP2C9 is difficult to predict using linear models, and because we use a linear model to run A in this case, it is no surprise that it inherits this limitation. Both the ideal predictor and A slightly outperform baseline prediction in terms of AUCROC, and A comes very close to ideal performance (within 2%). In one case A does slightly worse (0.2%) than baseline accuracy; this may be due to the fact that the marginals and $\hat{\pi}$ used by A are approximations to the true marginals and error distribution π.

We also evaluated A on the validation set (using a model f derived from the training set). We found that both genotypes are predicted more accurately on the training set than validation. For VKORC1, A was 3% more accurate and yielded an additional 4% AUCROC. The difference was less pronounced with CYP2C9, which was 1.5% more accurate with an additional 2% AUCROC. Although these differences are not as large as the absolute gain over baseline prediction, they persist across other training/validation splits. We ran 100 instances of cross-validation, and measured the difference between training and validation performance. We found that we were on average able to better predict the training cohort.

Differentially-Private Mechanisms

In the last section, we saw that linear models trained on private datasets leak information about patients in the training cohort. In this section, we explore the issue on models and datasets for which differential privacy has been applied.

As in the previous section, we take the perspective of the adversary, and attempt to infer patients' genotype given differentially-private models and different types of background information on the targeted individual. As such, we use the same attack model, but rather than assuming the

adversary has access to f, we assume access to a *differentially private version of* the original dataset D or f. We use two published differentially-private mechanisms with publicly-available implementations: *private projected histograms* (Vinterbo, 2012) and the *functional mechanism* (Zhang et al., 2012) for learning private linear regression models. Although full histograms are typically not published in pharmacogenetic studies, we analyze their privacy properties here to better understand the behavior of differential privacy across algorithms that implement it differently.

Differentially-private histograms. We first investigate a mechanism for creating a differentially-private version of a dataset via the private projected histogram method (Vinterbo, 2012). DP datasets are appealing because an (untrusted) analyst can operate with more freedom when building a model; she is free to select whichever algorithm or representation best suits her task, and need not worry about finding differentially-private versions of the best algorithms.

Because the numeric attributes in our dataset are too fine-grained for effective histogram computation, we first discretize each numeric attribute into equal-width bins. In order to select the number of bins, we use a heuristic given by Lei (Lei, 2011) and suggested by Vinterbo (Vinterbo, 2012), which says that when numeric attributes are scaled to the interval $[0, 1]$, the bin width is given by $(\log(n)/n)^{1/(d+1)}$, where $n = |D|$ and d is the dimension of D. In our case, this is implies two bins for each numeric attribute. We validated this parameter against our dataset by constructing 100 differentially-private datasets at $\epsilon = 1$ with $2, 3, 4$, and 5 bins for each numeric attribute, and measured the accuracy of a dose-predicting linear regression model over each dataset. The best accuracy was given for $k = 2$, with the difference in means for $k = 2$ and $k = 3$ not attributable to noise. When the discretized attributes are translated into a private version of the original dataset, the median value from each bin is used to create numeric

values.

To infer the private genomic attributes given a differentially-private version D_ϵ of a dataset, we compute an empirical approximation $\hat{p}$ to the joint probability p by counting the frequency of tuples in D_ϵ. A minor complication arises due to the fact that numeric values in D_ϵ have been discretized and re-generated from the median of each bin. Therefore, the likelihood of finding a row in D_ϵ that matches any row in D_T or D_V is low. To account for this, we transform each numeric attribute in the background information to the nearest median from the corresponding attribute used in the discretization step when generating D_ϵ. We then use $\hat{p}$ to directly compute a prediction of the genotype x_t that maximizes $\Pr_{\hat{p}}[x_T|x_I, y]$.

Differentially-private linear regression. We also investigate use of the functional mechanism (Zhang et al., 2012) for producing differentially-private linear regression models. The functional mechanism works by adding Laplacian noise to the coefficients of the objective function used to drive linear regression. This technique stands in contrast to the more obvious approach of directly perturbing the output coefficients of the regression training algorithm, which would require an explicit sensitivity analysis of the training algorithm itself. Instead, deriving a bound on the amount of noise needed for the functional mechanism involves a fairly simple calculation on the objective function (Zhang et al., 2012).

We produce private regression models on the IWPC dataset by centering each feature and then scaling into the interval $[-1, 1]$ This is described in the publicly-available implementation of the technique, and is necessary to ensure that sufficient noise is added to the objective function (i.e., the amount of noise needed is not scale-invariant). In order to inter-operate with the other components of our evaluation apparatus, we re-implemented the algorithm in R by direct translation from the Matlab implementation released by Zhang et al.

Applying model inversion to the functional mechanism is straightforward, as our adversary A makes no assumptions about the internal structure of the regression model or how it was derived. However, care must be taken with regards to data scaling, as the functional mechanism classifier is trained on scaled data. When calculating $\hat{X}$, all input variables must be transformed by the same scaling function used on the training data, and the predicted response must be transformed by the inverse of this function.

Results on private models

We evaluated our inference algorithms on both mechanisms discussed above at a range of ϵ values: 0.25, 1, 5, 20, and 100. For each algorithm and ϵ, we generated 100 private models on the training cohort, and attempted to infer VKORC1 and CYP2C9 for each individual in both the training and validation cohort. All computations were performed in R. All tests of statistical significance discussed below are one-tailed t-tests, and we consider a result significant whenever it reaches a significance level at or below 0.05. Figure 3.6 shows our results in detail. Our key findings are summarized as follows:

- Some ϵ values effectively protect genomic privacy for DP linear regression. For $\epsilon \leqslant 1$, A could not predict VKORC1 better on the training set than the validation set either in terms of accuracy or AUCROC. The same result holds on CYP2C9, but only when measured in terms of AUCROC. A's *absolute* performance for these ϵ is not much better than the baseline either: VKORC1 is predicted only 5% better at $\epsilon = 1$, and CYP2C9 sees almost no improvement.

- "Large"-ϵ DP mechanisms offer little genomic privacy. When $\epsilon \geqslant 5$, both DP mechanisms see a statistically-significant increase in training

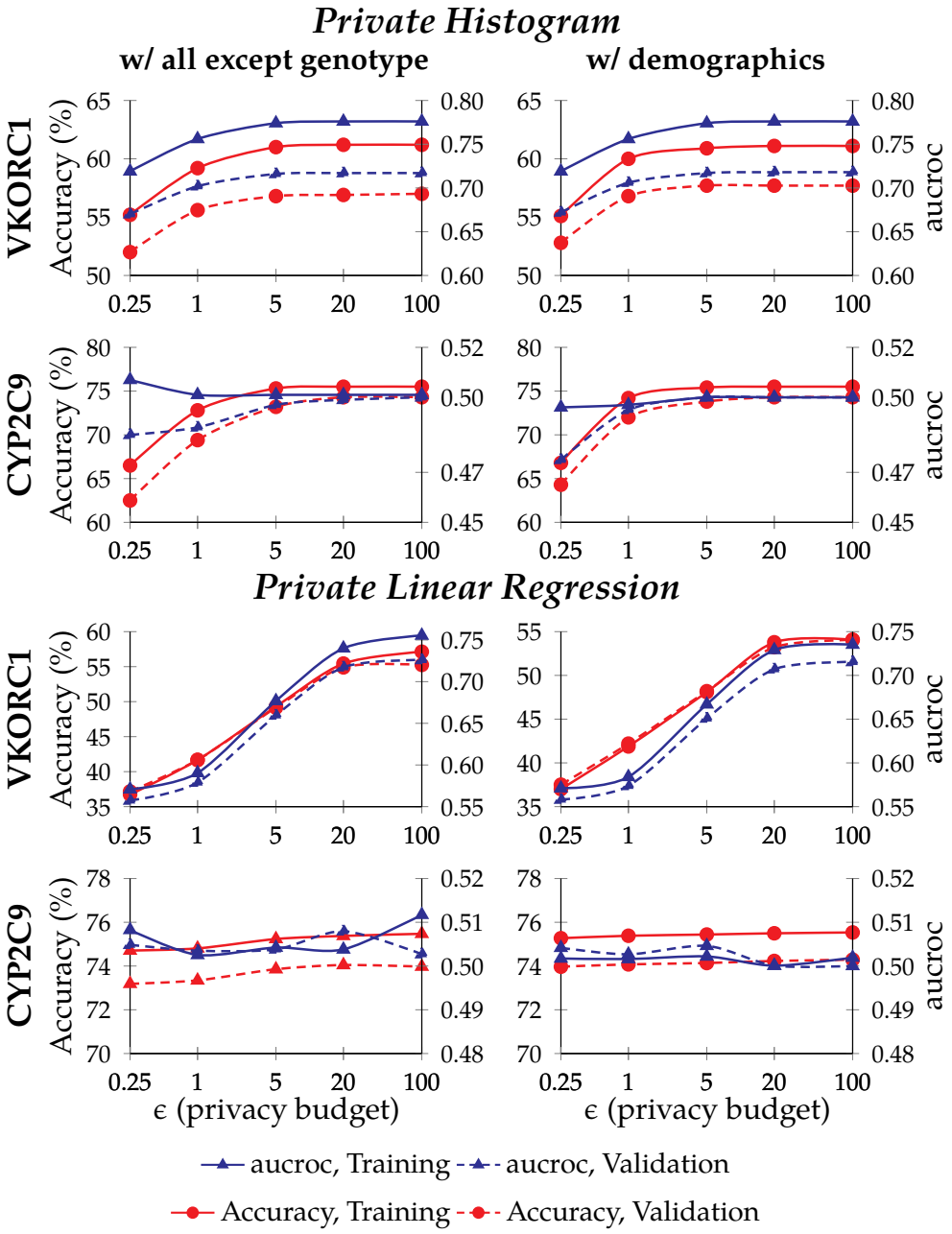


Figure 3.6: Inference performance for genomic attributes over IWPC training and validation set for private histograms (**left**) and private linear regression (**right**), assuming both configurations for background information. Dashed lines represent accuracy, solid lines area under the ROC curve (AUCROC). Smaller ϵ settings correspond to more privacy, and larger ones to less. The baseline accuracy for VKORC1 is 35%, and 75% for CYP2C9. The baseline AUCROC is always 0.5.

set performance over validation, and as ϵ approaches 20 there is little difference from non-private mechanisms (between 3%-5%).

- Private histograms disclose significantly more information about genotype than private linear regression, even at identical ϵ values. At all tested ϵ, private histograms leaked more on the training than validation set. *This result holds even for non-private regression models*, where the AUCROC gap reached 3.7% area under the curve, versus the 3.9% - 5.9% gap for private histograms. This demonstrates that the relative nature of differential privacy's guarantee can lead to meaningful concerns.

Our results indicate that understanding the implications of differential privacy for pharmacogenomic dosing is a difficult matter—even small values of ϵ might lead to unwanted disclosure in many cases. In the following, we discuss several aspects of these results in greater detail.

Private Histograms vs. Linear Regression. We found that private histograms leaked significantly more information about patient genotype than private linear regression models. The difference in AUCROC for histograms versus regression models is statistically-significant for VKORC1 at all ϵ. As Figure 3.6 indicates, the magnitude of the difference from baseline is also higher for histograms when considering VKORC1, nearly reaching 0.8 AUCROC and 63% accuracy, while regression models achieved at most 0.75 AUCROC and 55–60% accuracy. The AUCROC performance for VKORC1 was greater than the baseline for all ϵ. However, for CYP2C9 this result only held when assuming all background information except genotype, and only for $\epsilon \leqslant 5$; when we assumed only demographic information, there was no significant difference between baseline and private histogram performance.

Disclosure from Overfitting. In nearly all cases, we were able to better infer genotype for patients in the training set than those in the validation set. For private linear regression, this result holds for VKORC1 at $\epsilon \geqslant 5.0$ for AUCROC. The fact that the difference at certain ϵ values is not statistically-significant is evidence that private linear regression is effective at preventing genotype disclosure at these ϵ. For private histograms, this result held for VKORC1 at all ϵ, and CYP2C9 at $\epsilon < 5$ with all background information but genotype.

Differences in Genotype. For both private regression and histogram models, performance for CYP2C9 is strikingly lower than for VKORC1. Private regression models performed no differently from the baseline, achieving essentialy no gain in terms of accuracy and at most 1% gain in AUCROC. We observe that this also held in the non-private setting, and the ideal model achieved the same accuracy as baseline, and only 7% greater AUCROC. This indicates that CYP2C9 is not well-predicted using linear models, and A performed nearly as well as is possible.

Utility Analysis of Private Warfarin Models

In this section, we evaluate the potential medical consequences of using a differentially-private regression algorithm to make dosing decisions in warfarin therapy. Specifically, we estimate the increased risk of stroke, bleeding, and fatality resulting from the use of differentially-private warfarin dosing at several privacy budget settings. This approach differs from the normal methodology used for evaluating the utility of differentially-private data-mining techniques. Whereas evaluation typically ends with a comparison of simple predictive accuracy against non-private methods, we actually simulate the application of a privacy-preserving technique to its domain-specific task, and compare the outcomes of that task to those achieved without the use of private mechanisms.

These comparisons form an empirical basis that allows us to evaluate the central null hypothesis for this part of the study: *the ϵ values that protect genomic privacy introduce no additional risk for stroke, bleeding, and patient mortality over a non-genomic dose assignment protocol.* We compare to the non-genomic fixed-dose protocol because it uses no individual patient information, and is thus *private by default*. However, our results show that for privacy budgets including the "safe" range ($\epsilon \leqslant 1$) identified in the previous subsection, private pharmacogenetic dosing results in greater risk for stroke, bleeding, and mortality as compared to the fixed-dose protocol.

Simulated clinical trials. To evaluate the consequences of private genomic dosing algorithms, we simulate a clinical trial designed to measure the effectiveness of pharmacogenomic warfarin dosing regimens. The practice of simulating clinical trials is well-known in the medical research literature (Fusaro et al., 2013; Holford et al.; Bonate, 2000; Holford et al., 2000), where it is used to estimate the impact of various decisions before initiating a costly real-world trial involving human subjects. Our clinical-trial simulation follows the design of the CoumaGen clinical trials for evaluating the efficacy of pharmacogenomic warfarin-dosing algorithms (Anderson et al., 2007), which is the largest completed real-world clinical trial to date for evaluating these algorithms. At a high level, we train a non-private pharmacogenomic warfarin-dosing algorithm and a set of private pharmacogenomic warfarin dosing algorithms on the training set. The simulated trial draws random patient samples from the validation set, and for each patient, applies three dosing algorithms to determine the simulated patient's starting dose: the current standard clinical algorithm, the non-private pharmacogenomic algorithm, and the private pharmacogenomic algorithm whose performance we wish to measure. We then simulate the patient's physiological respose to the doses given by each

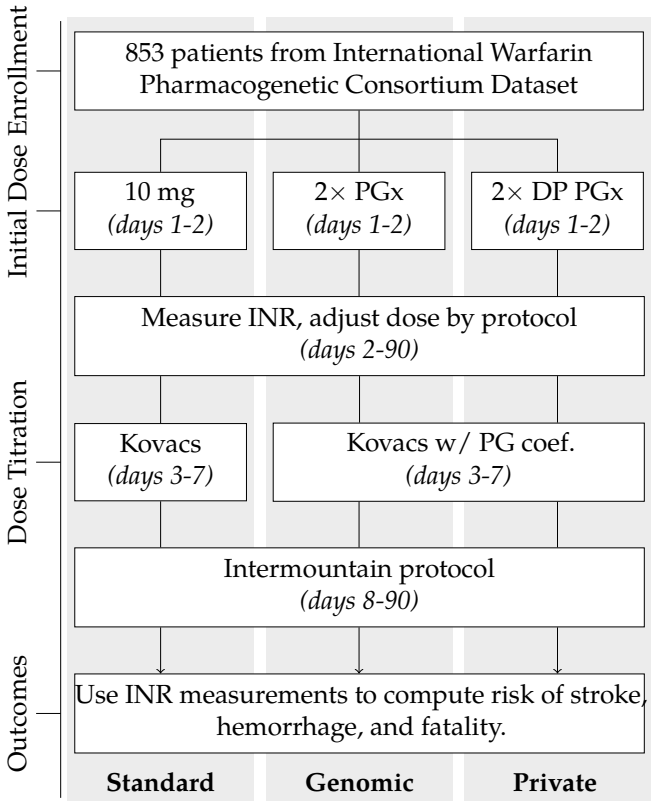


Figure 3.7: Overview of the Clinical Trial Simulation. *PGx* signifies the pharmacogenomic dosing algorithm, and *DP* differential privacy. The trial consists of three arms differing primarily on initial dosing strategy, and proceeds for 90 days. Details of Kovacs and Intermountain protocol are given below.

algorithm using a dose *titration* (i.e., modification) protocol defined by the original CoumaGen trial.

In more detail, our trial simulation defines three parallel *arms* (see Figure 3.7), each corresponding to a distinct method for assigning the patient's initial dose of warfarin:

1. *Standard:* the current standard practice of initially prescribing a fixed 10mg/day dose.

2. *Genomic:* Use of a genomic algorithm to assign the initial dose.

3. *Private:* Use of a differentially-private genomic algorithm to assign initial dose.

Within each arm, the trial proceeds for 90 simulated days in several stages, as depicted in Figure 3.7:

1. *Enrollment:* A patient is sampled from the population distribution, and their genotype and demographic characteristics are used to construct an instance of a *Pharmacokinetic/Pharmacodynamic (PK/PD) Model* that characterizes relevant aspects of their physiological response to warfarin (i.e., INR). The PK/PD model contains random variables that are parameterized by genotype and demographic information, and are designed to capture the variance observed in previous population-wide studies of physiological response to warfarin (Hamberg et al., 2007).

2. *Initial Dosing:* Depending on which arm of the trial the current patient is in, an initial dose of warfarin is prescribed and administered for the first two days of the trial.

3. *Dose Titration:* For the remaining 88 days of the simulated trial, the patient administers a prescribed dose every 24 hours. At regular intervals specified by the titration protocol, the patient makes "clinic visits" where INR response to previous doses is measured, a new dose is prescribed based on the measured response, and the next clinic visit is scheduled based on the patient's INR and current dose. This is explained in more detail later in this section.

4. *Measure Outcomes:* The measured responses for each patient at each clinic visit are tabulated, and the risk of negative outcomes is computed.

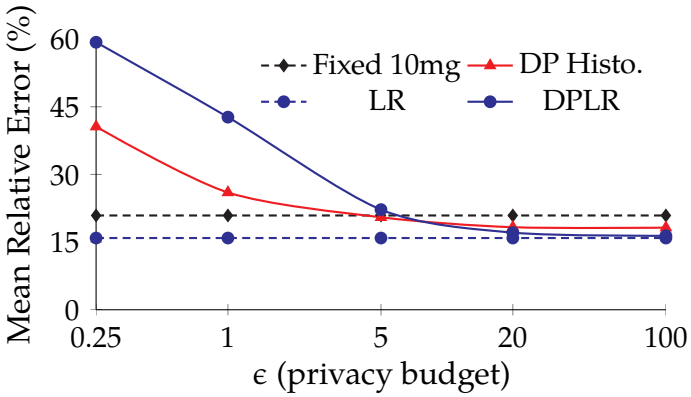


Figure 3.8: Pharmacogenomic warfarin dosing algorithm performance measured against clinically-deduced ground truth in IWPC dataset.

Pharmacogenomic Warfarin dosing algorithms. To build the non-private regression model, we use regularized least-squares regression in R, and obtained 15.9% average absolute error (see Figure 3.8). To build differentially-private models, we use two techniques: the functional mechanism of Zhang *et al.* (Zhang et al., 2012) and regression models trained on Vinterbo's private projected histograms (Vinterbo, 2012).

To obtain a baseline estimate of these algorithms' performance, we constructed a set of regression models for various privacy budget settings ($\epsilon = 0.25, 1, 5, 20, 100$) using each of the above methods. The average absolute predictive error, over 100 distinct models at each parameter level, is shown in Figure 3.8. Although the average error of the private algorithms at low privacy-budget settings is quite high, it is not clear how that will affect our simulated patients. In addition to the magnitude of the error, its *direction* (i.e., whether it under- or over-prescribes) matters for different types of risk. Futhermore, because the patient's initial dose is subsequently titrated to more appropriate values according to their INR response, it may be the case that a poor guess for initial dose, as long as the error is not too significant, will only pose a risk during the early portion

of the patient's therapy, and a negligible risk overall. Lastly, the accuracy of the standard clinical and non-private pharmacogenomic algorithms are moderate (about 15% and 20% error, respectively), and these are the best known methods for predicting initial dose. The difference in accuracy between these and the private algorithm is not extreme (e.g., greater than an order of magnitude), so lacking further information about the correlation between initial dose accuracy and patient outcomes, it is necessary to study their use in greater detail. Removing this uncertainty is the goal of our simulation-based evaluation.

Dose assignment and titration. To assign initial doses and control the titration process in our simulation, we follow the protocol used by the CoumaGen clinical trials on pharmacogenomic warfarin dosing algorithms (Anderson et al., 2007). In the standard arm, patients are given 10-mg doses on days 1 and 2, followed by dose adjustment according to the Kovacs protocol (Kovacs et al., 2003) for days 3 to 7, and final adjustment according to the Intermountain Healthcare protocol (Anderson et al., 2007) for days 8 to 90. Both the Kovacs and Intermountain protocols assign a dose and next appointment time based on the patient's current INR, and possibly their previous dose.

The genomic arm differs from the standard arm for days 1-7. The initial dose for days 1-2 is predicted by the pharmacogenomic regression model, and multiplied by two[6] (Anderson et al., 2007). On days 3-7, the Kovacs protocol is used, but the prescribed dose is multiplied by a coefficient C_{pg} that measures the ratio of the predicted pharmacogenomic dose to the standard 10mg initial dose: $C_{pg} = (2 \times \textit{Initial Pharmacogenomic Dose})/(10\ \text{mg})$. On days 8-90, the genomic arm proceeds identically to the standard arm.

[6]Note that the mean response of most pharmacogenomic models is approximately 5mg per day, so this protocol is roughly equivalent to the initial 10mg/day protocol used in the standard arm.

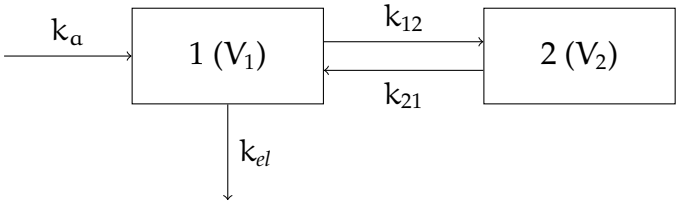


Figure 3.9: Two-compartment PK model with parameters k_a, k_{el}, V_1, V_2.

The private arm is identical to the genomic arm, but the pharmacogenomic regression model is replaced with a differentially-private model.

To simulate realistic dosing increments, we assume any combination of three pills from those available at most pharmacies: 0.5, 1, 2, 2.5, 3, 4, 5, 6, 7, and 7.5 mg. The maximum dose was set to 15 mg/day, with possible dose combinations ranging from 0 to 15 mg in 0.5 mg increments.

PK/PD model for INR response to Warfarin. We adopted a previously-developed PK/PD INR model to predict each patient's physiological (INR) response to previous dosing choices (Hamberg et al., 2007). The PK component of the model is a *two-compartment model with first-order absorption*. A two-compartment model assumes an abstract representation of the body as two discrete sections: the first being a *central* compartment into which a drug is administered and a *peripheral* compartment into which the drug eventually distributes. This is depicted in Figure 3.9. The central compartment (assumed to have volume V_1) represents tissues that equilibrate rapidly with blood (e.g., liver, kidney, *etc.*), and the peripheral (volume V_2) those that equilibrate slowly (e.g., muscle, fat, *etc.*). Three *rate constants* govern transfer between the compartments and elimination: k_{12}, k_{21}, for the central-peripheral and peripheral-central transfer, and k_{el} for elimination from the body, respectively. V_1, V_2, k_{12}, and k_{21} are related by the following equality:

$$V_1 k_{12} = V_2 k_{21}$$

The *absorption rate* k_a governs the rate at which the drug enters the central compartment. In the model used in our simulation, each of these parameters is represented by a random variable whose distribution has been fit to observed population measurements of Warfarin absorption, distribution, metabolism, and elimination (Hamberg et al., 2007). The elimination-rate constant k_{el} is parameterized by the patient's CYP2C9 genotype.

Given a set of PK parameters, the Warfarin concentration in the central compartment over time is calculated using standard two-compartment PK equations for oral dosing. Concentration in two-compartment pharmacokinetics diminishes in two distinct phases with differing rates: the α ("distribution") phase, and β ("elimination") phase. The expression for concentration C over time assuming doses $D_1, \ldots, D_n$ administered at times $t_{D_1}, \ldots, t_{D_n}$ has another term corresponding to the effect of oral absorption:

$$C(t) = \sum_{i=1}^{n} D_i (A e^{-\alpha t_i} + B e^{-\beta t_i} - (A + B) e^{-k_a t_i})$$

with $t_i = t - t_{D_i}$ and α, β satisfying

$$\alpha\beta = k_{21} k_{el} \qquad \alpha + \beta = k_{el} + k_{12} + k_{21}$$

and

$$A = \frac{k_a}{V_1} \frac{k_{21} - \alpha}{(k_a - \alpha)(\beta - \alpha)} \quad B = \frac{k_a}{V_1} \frac{k_{21} - \beta}{(k_a - \beta)(\alpha - \beta)}$$

The model of Hamberg et al. contains an error term with a zero-centered log-normal distribution whose variance depends on whether or not steady-state dosing has occurred.

The PD model used in our simulations is an *inhibitory sigmiod-*E_{max} *model*. Recall that the purpose of the PD model is to describe the physiological response E, in this case INR, to Warfarin concentration at a particular time. E_{max} represents the maximal response, i.e., the maximal

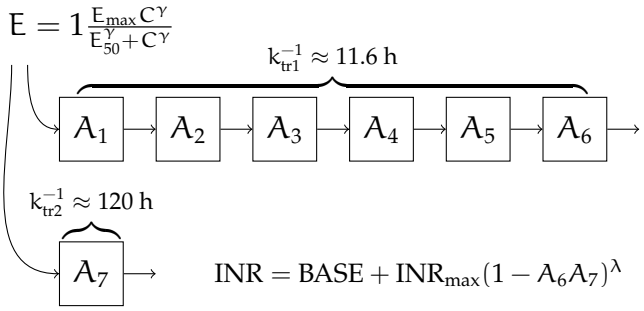


Figure 3.10: Overview of transit-compartment PD model (Hamberg et al., 2007). The total transit time of each chain corresponds to the inverse of the transfer coefficient. The mean transit time of the "long" chain is approximately 11.6 hours, and the "short" chain 120 hours.

inhibition of coagulation, and E_{50} the concentration of Warfarin producing half-maximal inhibition. E_{max} is fixed to 1, and E_{50} is a patient-specific random variable that is a function of the patient's VKORC1 genotype. A sigmoidocity factor γ is used to model the fact that the concentration-effect response of Warfarin corresponds to a sigmoid curve at lower concentrations. The basic formula for calculating E at time t from concentration is:

$$1 - \frac{E_{max}C(t)^\gamma}{E_{50}^\gamma + C(t)^\gamma}$$

However, Warfarin exhibits a delay between exposure and anticoagulation response (Kuruvilla and Gurk-Turner, 2001). To characterize this feature, Hamberg et al. showed that extending the basic E_{max} model with a *transit compartment model* with two parallel chains is adequate, as shown in Figure 3.10. The delay between exposure and concentration is modeled by assuming that the drug travels along two parallel *compartment chains* of differing lengths and turnover rates. The transit rate between compartments on the two chains is given by two constants k_{tr1} and k_{tr2}. The first chain consists of six compartments, and the second a single compartment. The first transit constant is a random zero-centered log-normal

variable, whereas empirical data did not reliably support variance in the second (Hamberg et al., 2007). The amount in a given compartment i, A_i, at time t is described by a system of coupled ordinary differential equations:

$$\frac{dA_1}{dt} = k_{tr1}\left(1 - \frac{E_{max}C(t)^\gamma}{E_{50}^\gamma + C(t)^\gamma}\right) - k_{tr1}A_1$$
$$\frac{dA_n}{dt} = k_{tr1}(A_{n-1} - A_n), n = 2, 3, 4, 5, 6$$
$$\frac{dA_7}{dt} = k_{tr2}\left(1 - \frac{E_{max}C(t)^\gamma}{E_{50}^\gamma + C(t)^\gamma}\right) - k_{tr2}A_7$$

The final expression for INR at time t is given by solving for A_6 and A_7 starting from initial conditions $A_i = 1$, and calculating the expression:

$$\log(\text{INR}) = \log(\text{Base} + \text{INR}_{max}(1 - A_6A_7)^\lambda) + \epsilon_{INR}$$

In this expression, Base is the patient's baseline INR, INR_{max} is the maximal INR (assumed to be 20 (Hamberg et al., 2007)), λ is a scaling factor derived from empirical data (Hamberg et al., 2007), and ϵ_{INR} is a zero-centerd, symmetrically-distributed random variable with variance determined from empirical data (Hamberg et al., 2007).

Calculating patient risk. INR levels correspond to the coagulation tendency of blood, and thus to the risk of adverse events. Sorensen *et al.* performed a pooled analysis of the correlation between stroke and bleeding events for patients undergoing warfarin treatment at varying INR levels (Sorensen et al., 2009). We use the probabilities for various events as reported in their analysis, which are shown in Table 3.1. We calculate each simulated patient's risk for stroke, intra-cranial hemorrhage, extra-cranial hemorrhage, and fatality based on the predicted INR levels produced by the PK/PD model. At each 24-hour interval, we calculated INR and the

Description	**Detail**	Value
Prior stroke		18%
Ischemic stroke probabilities		
w/No prior stroke		4.5%
w/Prior stroke		10.9%
RR INR in-range vs. all INRs	INR 2-3	0.46
RR INR outside range vs. in range	INR < 2.0	5.14
	INR > 3.0	0.73
Fatality resulting from stroke		
	INR $\geqslant 2.0$	8.1%
	INR < 2.0	17.5%
Bleeding probabilities		
ICH		
INR in-range	INR 2-3	0.25%
RR INR outside range	INR < 2.0	0.92
	INR > 3.0	4.31
Fatality from ICH	All INRs	51.6%
Major ECH		
Baseline probability	No treatment	0.2%
RR on warfarin	INR 2-3	2.22
RR INR outside range	INR < 2.0	2.14
	INR > 3.0	5.88
Fatality from Major ECH	All INRs	1.47%

Table 3.1: Annual cerebrovascular event probabilities for INR ranges. RR stands for relative risk, ICH for intra-cranial hemorrhage, ECH for extra-cranial hemorrhage. *Source:* Sorensen *et al.* (Sorensen et al., 2009).

corresponding risk for these events. The sum total risk for each event across the entire trial period is the endpoint we use to compare the arms. We also calculated the mean *time in therapeutic range* (TTR) of patients' INR response for each arm. We define TTR as any INR reading between 1.8–3.2, to maintain consistency with previous studies (Fusaro et al., 2013; Anderson et al., 2007).

The results are presented in Figure 3.11 in terms of relative risk (defined as the quotient of the patient's risk for a certain outcome when using a particular algorithm versus the fixed dose algorithm). The results are

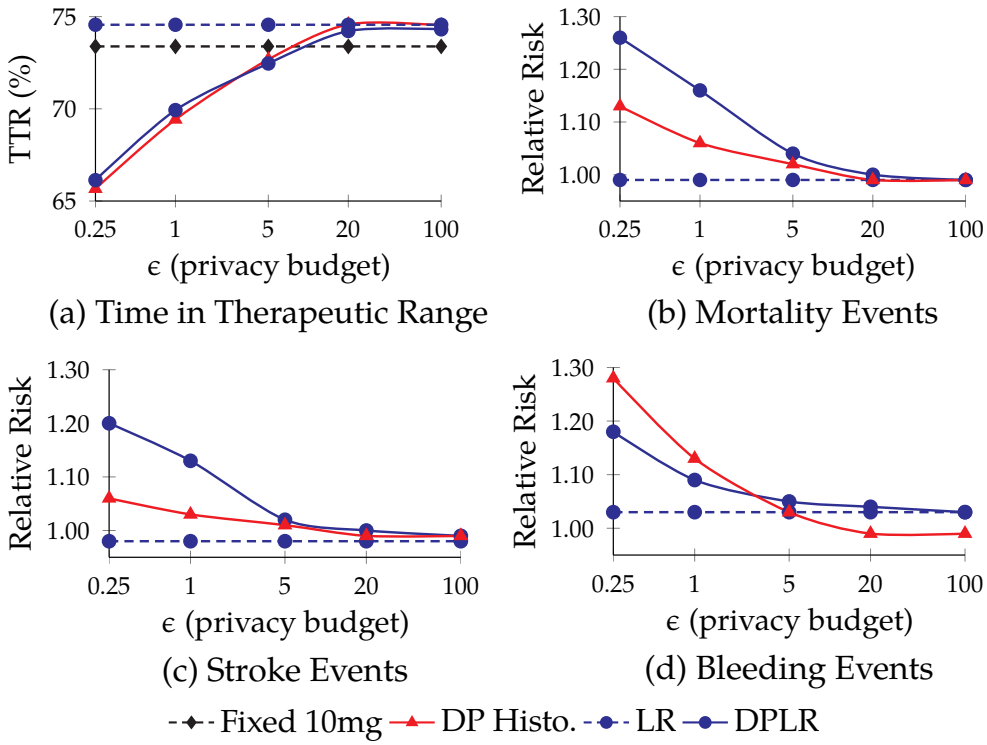


Figure 3.11: Trial outcomes for fixed dose, non-private linear regression (LR), differentially-private linear regression (DPLR), and private histograms. Notice that in subfigures (b)-(d), the fixed 10mg protocol is implicitly represented by the curve at Relative Risk $= 1$, which is not shown to avoid clutter in the plots. Horizontal axes represent ϵ.

striking: for reasonable privacy budgets ($\epsilon \leqslant 5$), private pharmacogenomic dosing results in greater risk for stroke, bleeding, and fatality events as compared to the fixed dose protocol. The increased risk is statistically significant for both private algorithms up to $\epsilon = 5$ and all types of risk (including reduced TTR), except for private histograms, for which there was no significant increase in bleeding events with $\epsilon > 1$.[7]

On the positive side, there is evidence that both algorithms may reduce

[7] Although the mean relative risk for bleeding events was greater than 1 as shown in Figure 3.11 (d), it was not statistically-significant at the $p \leqslant 0.05$ level for private histograms.

all types of risk at certain privacy levels. Differentially-private histograms performed slightly better, improvements in all types of risk at $\epsilon \geqslant 20$. Private linear regression seems to yield lower risk of stroke and fatality and increased TTR at $\epsilon \geqslant 20$. However, the difference in bleeding risk for DPLR was not statistically significant at any $\epsilon \geqslant 20$. *These results lead us to conclude that there is evidence that differentially-private statistical models may provide effective algorithms for predicting initial warfarin dose, but only at low settings of* $\epsilon \geqslant 20$ *that yield little privacy (see Section 3.2).*

Summary and Conclusion

We conducted an end-to-end case study of the use of differential privacy in a medical application, exploring the tradeoff between privacy and utility that occurs when existing DP algorithms are used to guide dosage levels in warfarin therapy. Using a new technique called *model inversion*, we repurpose pharmacogenetic models to infer patient genotype. We showed that models used in warfarin therapy introduce a threat to patients' genomic privacy. When models are produced using state-of-the-art differential privacy mechanisms, genomic privacy is protected for small $\epsilon(\leqslant 1)$, but as ϵ increases towards larger values this protection vanishes.

We evaluated the *utility* of differential privacy mechanisms by simulating clinical trials that use private models in warfarin therapy. This type of evaluation goes beyond what is typical in the literature on differential privacy, where raw statistical accuracy is the most common metric for evaluating utility. We show that differential privacy substantially interferes with the main purpose of these models in personalized medicine: for ϵ values that protect genomic privacy, which is the central privacy concern in our application, the risk of negative patient outcomes increases beyond acceptable levels.

4 SATISFIABILITY MODULO COUNTING

This chapter introduces a problem called *satisfiability modulo counting*, and shows 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 *parametric model-counting operations*. The object is to find an assignment to the model-counting parameters that satisfies the given constraints, or to demonstrate unsatisfiability. Section 4.1 presents a logic for expressing these problems, and Section 4.2 gives an abstract decision procedure fashioned after CDCL-based SMT procedures for solving them. The abstract procedure encapsulates 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. Section 4.3 describes an implementation of this procedure for counting over linear-integer arithmetic, including an effective strategy for generating lemmas. Section 4.4 concludes the chapter by applying this decision procedure to the verification of privacy properties over programs taken from a well-known privacy-preserving compiler called *Fairplay* (Ben-David et al., 2008; Malkhi et al., 2004).

4.1 A Logic with Parametric Counting Terms

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 primitives: a *satisfiablity oracle* $\mathcal{O}_{\mathsf{sat}}$, and a *model counting oracle* $\mathcal{O}_{\mathsf{cnt}}$. The satisfiability oracle $\mathcal{O}_{\mathsf{sat}}(\phi)$ is a function taking a formula $\phi \in \mathsf{L}$ to $\{\mathsf{true}, \mathsf{false}\}$, and returns true iff ϕ is satisfiable in L. The model counting oracle $\mathcal{O}_{\mathsf{cnt}}(V, \phi)$ is a function taking a formula $\phi \in \mathsf{L}$ and a set of variables V appearing in ϕ,

which returns the number of distinct models of ϕ:

$$\mathcal{O}_{\mathsf{cnt}}(\{v_1, \ldots, v_n\}, \phi) = |\{v_1, \ldots, v_n \ : \ \mathcal{O}_{\mathsf{sat}}(\phi[v_1 \mapsto v_1, \ldots, v_n \mapsto v_n])\}|$$

We make no assumptions about the computability of $\mathcal{O}_{\mathsf{sat}}$ and $\mathcal{O}_{\mathsf{cnt}}$, and only require that they can be defined meaningfully for $\mathcal{L}_{\#}(\mathsf{L})$.

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^s is a parameter (or *shared*, denoted by the superscript s) in the linear-integer formula $0 \leqslant y^s \leqslant 5 \wedge \mathsf{count}(x, 1 \leqslant x \leqslant y^s) \leqslant 2$, then the formula describes all values for the parameter y^s that lie in the interval $[0, 5]$ and result in at most two solutions for the variable x when $1 \leqslant x \leqslant y^s$. The meaning of a $\mathcal{L}_{\#}(\mathsf{L})$ formula is defined by the set of models taken by counting parameters. Notice that a count term whose second argument contains a parameter y^s corresponds to a function $f : D \mapsto \mathbb{Z}_+$ (where D is the domain of y^s), rather than a constant value from $\mathbb{Z}_+$. The presence of parameters in a $\mathcal{L}_{\#}(\mathsf{L})$ formula means that counting terms cannot be reduced to integer 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 4.1 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 ($\leqslant, <, =$) 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 $\mathsf{SharedFormula}$). 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 $\mathsf{count}(x, 0 \leqslant x \wedge y \leqslant 3 \wedge x + y = 5)$ the same interpretation as $\mathsf{count}(x, \exists y : 0 \leqslant x \wedge y \leqslant 3 \wedge x + y = 5)$. Aside from this type of implicit quantification in the target of count terms, $\mathcal{L}_{\#}(\mathsf{L})$ is quantifier-free.

Semantics. As this chapter is primarily concerned with satisfiability decision procedures for $\mathcal{L}_{\#}(\mathsf{L})$, we focus on the semantics of satisfiability. We define this semantics by giving a denotational translation $[\![\cdot]\!](\alpha)$ from $\mathcal{L}_{\#}(\mathsf{L})$ formulas and variable mappings to sentences in the quantifier-free

Base	BaseFormula	::= $\phi(V_{\text{share}}, V_{\text{base}}) \in L$
Counting	Const	::= c from $\mathbb{R}$
	Term	::= Const | Term $\{+, \times\}$ Term | count$(v_0, \ldots, v_n$, BaseFormula$)$ *where* $v_0, \ldots, v_n \subseteq V_{\text{base}}$ in BaseFormula
	Atom	::= Term $\{\leqslant, <, =\}$ Term
Outer	SharedFormula	::= $\phi(V_{\text{share}}) \in L$
	Formula	::= Atom | (SharedFormula) | (Formula) | $\neg$Formula | Formula $\wedge$ Formula

Figure 4.1: $\mathcal{L}_{\#}(L)$ grammar.

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 $\mathcal{L}_{\#}(L)$ in terms of the existence of a variable mapping α that results in a satisfiable QF_NRA formula $[\![\phi]\!](\alpha)$.

To ensure compatibility between our denotation operator and QF_NRA, which does not include ∞, 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 18). A formula is finite-base iff each count term is given a finite interpretation by $\mathcal{O}_{\text{cnt}}$ for any assignment to its parameters.

Definition 18. *(*Finite-base formula*). A formula* $\phi_L \in \mathcal{L}_{\#}(L)$ *is* finite-base *iff for each instance of*

$$\text{count}(\{x_1, \ldots, x_n\}, \phi_L(x_1, \ldots, x_n, y_1^s, \ldots, y_m^s))$$

that appears in ϕ*, then for all assignments* α *to the parameters* $y_1^s, \ldots, y_m^s$*,* $\mathcal{O}_{\text{cnt}}(\{x_1, \ldots, x_n\}, \alpha(\phi_L))$ *is finite.*

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

Figure 4.2: Semantic denotation operator $[\![\cdot]\!](\alpha)$. $\mathcal{L}_{\#}(\mathsf{L})$ formulas are transformed into QF_NRA formulas recursively by replacing assignments under α, and then applying the satisfiability and counting oracles, $\mathcal{O}_{\mathsf{sat}}$ and $\mathcal{O}_{\mathsf{cnt}}$, respectively.

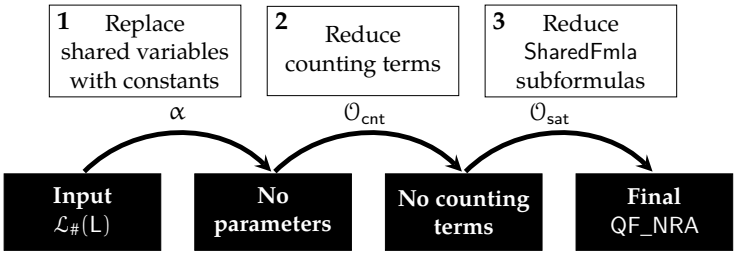


Figure 4.3: High-level workflow for the denotational transformation from $\mathcal{L}_{\#}(\mathsf{L})$ to QF_NRA.

The denotational translation $[\![\cdot]\!](\alpha)$ on which our semantics is based works in three phases as shown in Figure 4.3. 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 counting oracle $\mathcal{O}_{\mathsf{cnt}}(\{x_1, \ldots, x_n\}, \phi)$. Lastly, each SharedFormula subformula is reduced to a Boolean constant using $\mathcal{O}_{\mathsf{sat}}$. The final step is always possible because SharedFormula 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' whose semantics we equate with those of $\alpha(\phi)$.

A full definition of the denotational tranformation operator $[\![\cdot]\!]$ is given by the set of reduction rules shown in Figure 4.2. We use $[\![\cdot]\!](\alpha)$ to define the satisfiability of a $\mathcal{L}_{\#}(\mathsf{L})$-formula, given in Definition 19.

Setup: $\alpha = [y^s \mapsto 1]$
Original $\mathcal{L}_{\#}(L)$ **:** $0 \leqslant y^s \leqslant 5 \wedge \mathsf{count}(x, 1 \leqslant x \leqslant y^s) \leqslant 2$

(1)	$[\![0 \leqslant y^s \leqslant 5 \wedge \mathsf{count}(x, 1 \leqslant x \leqslant y^s) \leqslant 2]\!]([y^s \mapsto 1])$ $\rightarrow$ $[\![0 \leqslant 1 \leqslant 5 \wedge \mathsf{count}(x, 1 \leqslant x \leqslant 1 \leqslant 2)]\!]([y^s \mapsto 1])$
(2)	$[\![0 \leqslant 1 \leqslant 5 \wedge \mathsf{count}(x, 1 \leqslant x \leqslant 1) \leqslant 2]\!]([y^s \mapsto 1])$ $\rightarrow$ $[\![0 \leqslant 1 \leqslant 5 \wedge 1 \leqslant 2]\!]([y^s \mapsto 1])$
(3)	$[\![0 \leqslant 1 \leqslant 5 \wedge 1 \leqslant 2]\!]([y^s \mapsto 1]) \rightarrow \mathsf{True}$

Figure 4.4: Example reduction using the transformation given in Figure 4.2, using a formula from the theory of linear-integer arithmetic. Each numbered step corresponds to those given in the high-level workflow displayed in Figure 4.3.

Example 4.1. *The semantic transformation is shown in more detail in Figure* **??**, *where we show how a* $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ *formula is reduced under the mapping* $\alpha = [y^s \mapsto 1]$. *In the first step, the shared variables are replaced with their image under* α *throughout the formula. This occurs inside of* count *terms as well as the outer formula, so for example* $\mathsf{count}(x, 1 \leqslant x \leqslant y^s)$ *becomes* $\mathsf{count}(x, 1 \leqslant x \leqslant 1)$. *In the second step, all counting atoms are replaced with atoms from* QF_NRA. *We use* $\mathcal{O}_{\mathsf{cnt}}$ *to arrive at the solutions for the* count *terms:* 1, *because there is exactly one solution to* $1 \leqslant x \leqslant 1$. *In the third step, the top-level formulas in* QF_LIA, *which are always instances of the* SharedFormula *syntactic rule, are reduced to Boolean literals using* $\mathcal{O}_{\mathsf{sat}}$. *In this example, both* SharedFormula *atoms* $0 \leqslant 1 \leqslant 5$ *and* $1 \leqslant 2$ *are* true *in* QF_LIA. *After this transformation, the entire formula is equivalent to* True *in* QF_NRA, *which means that* α *is a satisfying assignment for the original* QF_LIA *formula.*

Definition 19. (Satisfiability of formulas in $\mathcal{L}_{\#}(L)$). *A finite-base* $\mathcal{L}_{\#}(L)$ *formula* ϕ *is satisfiable in* $\mathcal{L}_{\#}(L)$ *iff there exists a mapping* α *such that* $[\![\phi]\!](\alpha)$ *is*

valid in QF_NRA.

The choice of L affects the decidability of $\mathcal{L}_{\#}(\mathsf{L})$. Theorem 4.2 shows that using quantifier-free linear-integer arithmetic as the base logic leads to undecidability. However, $\mathcal{L}_{\#}(\mathsf{L})$ is decidable in several important cases, as discussed in Sections 4.2 and 4.3.

Theorem 4.2. $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$*, the theory of counting over the quantifier-free fragment of linear-integer arithmetic, is undecidable.*

Proof. 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

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

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 assumption simplifies the proof, because a count expression cannot return negative values. We show how to obtain an $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ formula $\phi(x_1^s, \ldots, x_n^s)$ 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 when q is encoded in binary, this increase in size is not a concern, because we only consider decidability.

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

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

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., $0 \leqslant x$ and $x < x^s$), and $0 \leqslant x \wedge x < x^s$ is in $\mathsf{QF\_LIA}$. 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 $\phi(x_1^s, \ldots, x_n^s)$ corresponds via the identity mapping to a solution for $p(x_1, \ldots, x_n)$. First, $\alpha(x_i^s)$ is an integer for all i. Notice that the left-hand side of $\phi(\alpha(x_1^s), \ldots, \alpha(x_n^s))$ evaluates to the same value as that in $p(\alpha(x_1^s), \ldots, \alpha(x_n^s))$. We can see this on a term-by-term basis:

1. In ϕ, each $\mathsf{count}(x, 0 \leqslant x < x^s)$ evaluates to $\alpha(x^s)$. In p, each occurrence of x also evaluates to $\alpha(x^s)$.

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

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

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. □

4.2 $\mathsf{sat}^{\#}$: An Abstract Decision Procedure

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, we give a small set of general requirements in the form of three *black-box* interface functions needed to make the decision procedure work: genparam (Definition 20), explain (Definition 21), and reduce (Definition 22).

$\mathsf{sat}^{\#}$ takes after CDCL satisfiability and SMT solvers (Nieuwenhuis et al., 2006), and borrows concepts from the model-constructing satisfiability calculus of de Moura and Jovanovíc (de Moura and Jovanović, 2013). 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 capability 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.

Abstract Procedure

We describe $\mathsf{sat}^{\#}$ as a transition system, as introducted in Section 2.2. Recall that 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. In addition

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**	$c' = (l_1 \vee \ldots \vee l_n \vee l) \in C$ $\mathsf{value}(l_i, M) = \mathsf{false}$ $\mathsf{value}(l, M) = \mathsf{undef}$
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
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

Figure 4.5: Clausal transition rules for $\mathsf{sat}^{\#}$. These rules are the same as those given in Section 2.2 (Figure 2.8), but updated to use the helper functions and definitions specific to $\mathsf{sat}^{\#}$.

C-DECIDE

$$\langle M, C\rangle \mapsto \langle [M, x \mapsto v], C\rangle \quad \textbf{if} \quad \begin{array}{l} x \in V_{\mathsf{share}}(C) \text{ not decided in } M \\ C' = \{c_1, \ldots, c_n\} \\ c_i, \ldots, c_n \in L(V_{\mathsf{share}}) \\ C' \cap M_L \cap C \text{ is maximal} \\ v = \mathsf{genparam}(C', x) \\ \mathsf{consistent}([M, x \mapsto v]) \end{array}$$

C-PROPAGATE

$$\langle M, C\rangle \mapsto \langle [M, c \to l], C\rangle \quad \textbf{if} \quad \begin{array}{l} l \in C, \mathsf{value}_b(l, M) = \mathsf{undef} \\ \mathsf{conflict}(C, [M, \neg l]) \\ c = \mathsf{explain}(l, M) \end{array}$$

C-MODEL-DECIDE

$$\langle M, C\rangle \mapsto \langle [M, l], C\rangle \quad \textbf{if} \quad \begin{array}{l} x \text{ is not decided in } M \\ \neg\mathsf{consistent}([M, x \mapsto v]) \\ c = \mathsf{explain}(\mathsf{false}, [M, x = v]) \\ c = l_1 \vee \cdots \vee l_n \vee l \\ \mathsf{value}(l, M) = \mathsf{undef} \end{array}$$

C-LEARN

$$\langle M, C\rangle \mapsto \langle M, C \cup \{c\}\rangle \quad \textbf{if} \quad \begin{array}{l} \mathsf{conflict}(C, M) \\ c = \mathsf{explain}(\mathsf{false}, M) \end{array}$$

C-BACKJUMP

$$\langle [M, N], C\rangle \mapsto \langle [M, c \to l], C\rangle \quad \textbf{if} \quad \begin{array}{l} \mathsf{conflict}(C, [M, N]) \\ \mathsf{explain}(\mathsf{false}, [M, N]) = c \\ c = l_1 \vee \cdots \vee l_n \vee l \\ \mathsf{value}(l_i, M) = \mathsf{false} \\ \mathsf{value}(l, M) = \mathsf{undef} \end{array}$$

C-BACKJUMP-DECIDE

$$\langle [M, x \mapsto v, N], C\rangle \mapsto \langle [M, l], C\rangle \quad \textbf{if} \quad \begin{array}{l} \mathsf{conflict}(C, [M, x \mapsto v, N]) \\ \mathsf{explain}(\mathsf{false}, [M, x \mapsto v, N]) = c \\ c = l_1 \vee \cdots \vee l_n \vee l \\ \exists l_i.\mathsf{value}(l_i, M) = \mathsf{undef} \\ \mathsf{value}(l, M) = \mathsf{undef} \end{array}$$

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

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

Figure 4.6: Counting theory transition rules for sat$^{\#}$. The clausal rules are the same as those given in Section 2.2. The helper function value is defined on the bottom.

to the decided and propagated literals discussed in Section 2.2, trails in $\mathsf{sat}^{\#}$ also have *model assignments* (appearing as $x \mapsto v$) that correspond to decisions that map a value to a counting-theory variable. 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. Note that model assignments are restricted to variables in V_{share}, so the values they take are drawn from L. We write M_{L} to denote the set of clauses consisting of the base-logic literals decided in or implied by M.

The set of model assignments on a trail M induces an interpretation α_M of shared variables; i.e., if $x \mapsto v$ appears on M, then $\alpha_M(x) = v$. Similarly, a trail M induces an interpretation of literals appearing in the clause set C. Following again de Moura and Jovanović (de Moura and Jovanović, 2013), 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 4.2) reduces to true (resp. false) under the interpretation α_M. Both are undefined (undef) otherwise. The full definition of these functions is given in Figure 4.6.

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 4.6. 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 $\mathsf{satisfied}$ 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 20), explain (Definition 21), and reduce (Definition 22). Each of these functions assumes computable versions of the oracles $\mathcal{O}_{\mathsf{cnt}}$ and $\mathcal{O}_{\mathsf{sat}}$ introduced in Section 4.1. genparam builds partial assignments for $\mathcal{L}_{\#}(L)$ 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 20. $\mathsf{genparam}(C, x)$. *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* $\mathcal{O}_{\mathsf{sat}}(c_1 \wedge \cdots \wedge c_n \wedge x = v)$ *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 that follow from $\mathcal{O}_{\mathsf{cnt}}$ and the base-logic semantics.

Definition 21. $\mathsf{explain}(l, M)$. *Given a consistent trail* M *and a literal* l *implied by* M, $\mathsf{explain}(l, M)$ *returns an* $\mathcal{L}_{\#}(L)$ *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' *implies* 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 call to reduce amounts to 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, because future decisions might make it unsatisfiable.

Definition 22. reduce(M, l). *Given a trail* M *and a literal* l, reduce(M, l) *returns a new literal* l' *derived from* l *by applying two changes:*

1. *If* l *contains a* 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* $l \in L$ *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* $\mathcal{O}_{\mathsf{sat}}(\alpha_M(l) \wedge M_L)$ *is* false, *and* undef *otherwise.*

Transition Rules. Given a set of clauses C_0, sat$^\#$ begins in the state $\langle [], C_0 \rangle$. Rules from Figures 4.5 and 4.6 are applied with the goal of entering either the sat or unsat state. The rules shown in Figure 4.5 are the clausal search and conflict rules, which are similar to those used by traditional CDCL solvers (see Section 2.2). The main modifications come from the use of helper functions specific to sat$^\#$ (e.g., value, satisfied, and conflict) to maintain continuity with the counting-theory rules in Figure 4.6, and the rules Learn, Forget, and Backjump. Whereas the traditional rules given in Section 2.2 combine lemma-learning with backjumping, here they are separated. This separation means that learning a new clause is optional, and that learned clauses can be "forgotten" when they are no longer useful.

The rules shown in Figure 4.6 are specific to counting theory. 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, to ensure that value remains well-defined. The clause set C' passed to genparam can contain 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 restriction 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 $[\mathsf{count}(\{x, y\}, 0 \leqslant x, y \leqslant y^s) = 121]$, $\mathsf{sat}^{\#}$ might pass the clause set $C' = \{6 \leqslant y^s \leqslant 12\}$ to genparam. $\mathsf{sat}^{\#}$ can obtain this sort of information through theory-specific mechanisms or approximation methods; an example is given in Section 4.3.

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. Such assumptions are 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 $[\mathsf{count}(\{x, y\}, 0 \leqslant x, y \leqslant y^s) = 120]$, where the literal has been propagated. The procedure cannot continue unless explain concludes that y^s cannot take a value that satisfies all clauses. This may not be possible because $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ is undecidable, but if it can use a hypothetical assignment $y^s \mapsto 9$ to give a lemma that implies $y^s > 9$, and subsequently produce a similar assignment-driven lemma that implies $y^s < 10$, then it can stop in unsat. This example illustrates how our explain for QF_LIA works by finding such contradictions, as we describe further in Section 4.3. C-Model-Decide incorporates these facts when they are needed.

The counting theory-specific learning rule, C-Learn, behaves like the lemma-learning component of the clausal BackjumpLemma rule, but it uses explain to produce a lemma. There are two backtracking rules specific to the counting theory. The first, C-Backjump, is similar to the clausal BackjumpLemma 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 $x \mapsto v$, where the normal clausal backjump rule does not apply because multiple literals in the explanation could 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 (de Moura and Jovanović, 2013).

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 4.3.

Theorem 4.3. (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.

Proof. 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:

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

This condition corresponds to the negation of the conjoined preconditions for the Sat and Unsat rules:

- $(\neg\mathsf{satisfied}(C, M) \wedge \neg\mathsf{conflict}(C, M))$ implies that at least one shared variable is not assigned in M or a literal from 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 (i.e., an explanation

in which $\mathsf{explain}(\mathsf{false}) = \mathsf{false}$) exists. Thus, because explain does not return false, one of the backjump and clause-learning rules can be applied to add a new clause to C.

- $(\neg\mathsf{satisfied}(C, M) \wedge M$ 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 in its Boolean skeleton. The second condition implies that the values of counting parameters given by the model assignments in M do not invalidate the Boolean satisfiability, i.e., the semantics of $\mathcal{L}_{\#}(L)$ 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 4.6c). For any l such that $\mathsf{value}_b(l, M) = \mathsf{true}$, the first condition implies that $\mathsf{reduce}(M, l) = \mathsf{true}$. This situation implies that α_M is total over $V_{\mathsf{share}}(l)$, and that $\mathcal{O}_{\mathsf{sat}}(\alpha_M(l)) = \mathsf{true}$. We can then conclude that C is satisfiable as a QF_NRA formula, so by Definition 19 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.

These conditions can arise for reasons that arise entirely because of conflicts in the base logic subformulas, or because the procedure has learned a set of lemmas that preclude further assignments to the shared variables. In the former case, Propagate and C-Propagate will 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. In the latter case, Propagate and C-Propagate will 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 $\phi \rightarrow l$, where l implies that $x^s \neq \alpha_M(x^s)$, for some ϕ and x^s assigned on M. It follows that C is unsatisfiable. We conclude that when sat$^\#$ enters Unsat, C is unsatisfiable.
□

There are certain cases where sat$^\#$ is complete as well. As long as the explain function can only return a finite number of clauses, then sat$^\#$ will always terminate. Borrowing from a similar argument of de Moura and Jovanovíc (de Moura and Jovanović, 2013), we can show this property by imposing an order on the states entered by sat$^\#$, and reasoning that the set of such states in any run progresses monotonically. When the space of possible lemmas is finite, this order has a maximal element, and completeness follows.

Theorem 4.4. (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.*

Proof. We begin with a brief sketch. This proof proceeds along the lines of that of Theorem 1 in (de Moura and Jovanović, 2013). We define a lexicographic partial order on states, and show that the transition rules in Figure 4.6 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 (de Moura and Jovanović, 2013), we define a lexicographic partial order on $\mathsf{sat}^{\#}$ states. It is based on a weight assignment w for trail elements:

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

Propagated literals are given the most weight, followed by decided literals, and finally model assignments. This weighting 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:

$$
\begin{array}{lllll}
\textit{1)} & [\,] \sqsubset M & \text{iff} & M \neq [\,] \\
\textit{2)} & [a, M_1] \sqsubset [b, M_2] & \text{if} & w(a) < w(b) \\
\textit{3)} & [a, M_1] \sqsubset [b, M_2] & \text{if} & w(a) = w(b) \wedge M_1 \sqsubset M_2 \\
\textit{4)} & \langle M_1, C_1 \rangle \prec \langle M_2, C_2 \rangle & \text{if} & M_1 \sqsubset M_2 \\
\textit{5)} & \langle M_1, C_1 \rangle \prec \langle M_2, C_2 \rangle & \text{if} & M_1 = M_2 \wedge |C_1| > |C_2|
\end{array}
$$

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 $|C_{\mathsf{univ}}|$ propagations and a total model assignment is maximal among those that will ever appear in a given application of the procedure. We now have 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, because their resulting states cause $\mathsf{sat}^{\#}$ to terminate immediately.
- Any rule that adds an element to the trail is immediately monotonic, because $\langle M, C_1 \rangle \prec \langle [M, a], C_2 \rangle$ by rules 1 and 3. This observation 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 $M_1 = [M, x^s \mapsto v, N]$ to $M_2 = [M, c \to l']$. Reasoning similar to that used with Backump and C-Backjump applies: $w(x^s \mapsto v) < w(c \to l')$, so rules 1 and 3 imply that $M_1 \prec M_2$.

- At first blush, Learn and C-Learn seem to pose a problem, because 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 clause set will reach its minimal configuration. When this condition occurs, Learn and C-Learn are no longer applicable, so a different rule must apply. Notice that at this point the state cannot change without raising the rank of the trail M on $\sqsubset$, because the only way to do so would be Forget, which cannot be applied on a conflict trail, 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, because it leads to immediate termination), one of the backjump rules, or one of the propagate rules. When this situation happens, the resulting state will be ranked higher than the current one 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 for $\prec$. 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.
□

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

We describe an implementation of $\mathsf{sat}^{\#}$ for $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ called countersat. It is based on Barvinok's algorithm (Barvinok, 1994) 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 (De Moura and Bjørner, 2008), `libbarvinok` (Verdoolaege et al., 2007), the Nomad optimization library (Abramson et al.), and Mathematica. Because Theorem 4.2 shows that $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ is undecidable, Theorem 4.4 obviously does not apply to our implementation. We describe a fragment of $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ for which our implementation of explain returns lemmas from a finite set, so that countersat is guaranteed to terminate with the correct answer.

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

Using Barvinok's algorithm (see Section 2.4 for background on this topic), 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 22), explain (Definition 21), and genparam (Definition 20) using Barvinok's algorithm and Z3. We describe these components in the context of an example formula:

$$s_1^{\mathbb{S}} = s_2^{\mathbb{S}} \wedge \mathsf{count}(\{x, y\}, 0 \leqslant x \leqslant s_1^{\mathbb{S}} \wedge 0 \leqslant y \leqslant s_2^{\mathbb{S}}) = 120$$

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:

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

Implementing reduce. Our implementation of reduce relies heavily on `libbarvinok`, which supports parametric counting of Presburger formulas that contain 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 $0 \leqslant x \leqslant s_1^s \wedge 0 \leqslant y \leqslant s_2^s$ yields a parametric enumerator with two chambers, the first corresponding to the positive quadrant of the plane:

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

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 $\mathsf{reduce}(l_2, [s_1^s \mapsto 5, s_2^s \mapsto 5])$ activates the first chamber, and yields the invalid formula $1 + (5) + (5) + (5)(5) = 120$. Because there are no shared variables remaining in this formula, we make a validity query to Z3, which yields the answer False.

Implementing explain. Our implementation of explain produces two types of lemmas: those arising from QF_LIA that do not involve counting the-

ory, and those arising due to counting-theory conflicts. To illustrate the first type, suppose that explain is given a trail resulting from the unit-propagation of l_1 and l_2, and the following constraints on shared variables:

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

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 $s_2^s = 5$, and the QF_LIA literal l_1. explain detects the conflict 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 $s_2^s \neq 5$:

$$s_1^s = 3 \wedge s_1^s = s_2^s \wedge s_2^s = 5$$

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: $s_1^s \neq 3 \vee \neg(s_1^s = s_2^s) \vee s_2^s \neq 5$.

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: $\mathsf{explain}(s_2^s \neq 10, [\rightarrow l_1, \rightarrow l_2, s_1^s \mapsto 10])$. Recalling again that the negation of $s_2^s \neq 10$ 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, because

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

explain can always return the naive-but-valid lemma $\neg l_2 \vee s_1^s \neq 10 \vee s_2^s \neq 10$; however, this approach often leads to non-terminating behavior, because 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,

$$p(s_1^s, s_2^s) = 1 + s_1^s + s_2^s + s_1^s s_2^s, \quad \phi \equiv s_1^s \geqslant 0 \wedge s_2^s \geqslant 0$$

We start with s_1^s, attempting to find its monotone regions in p. To do so, we apply *cylindrical algebraic decomposition* (Collins, 1975) (CAD) to the partial derivative $D_{s_1^s} p(s_1^s, s_2^s)$ of p over s_1^s, which produces a set of regions over which $D_{s_1^s} p(s_1^s, s_2^s)$ is sign-invariant. Care must be taken before applying CAD, because any floor terms in the quasi-polynomial must first be removed by adding fresh existentially-quantified variables (Pugh, 1994). 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 $s_1^s \geqslant 0$. Similarly, p is monotonic in s_2^s whenever $s_2^s \geqslant 0$. From these observations, 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:

$$\neg l_1 \vee \neg l_2 \vee \neg(s_1^s \geqslant 0) \vee \neg(s_2^s \geqslant 0) \vee s_1^s < 10 \vee s_2^s < 10 \qquad (4.1)$$

Notice that the nearest solution that satisfies all of the QF_LIA constraints, as well as this lemma, is $s_1^s \mapsto 9, s_2^s \mapsto 9$, which by the same reasoning

causes explain to return the lemma:

$$\neg l_1 \vee \neg l_2 \vee \neg(s_1^s \geqslant 0) \vee \neg(s_2^s \geqslant 0) \vee s_1^s > 9 \vee s_2^s > 9 \tag{4.2}$$

Clauses 4.1 and 4.2 quickly lead 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 condition occurs when the polynomial is one-dimensional.[1] Specialization yields weaker lemmas.

explain is complete on a practical fragment of $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$, termed the *monotone fragment* (Definition 23). All but one of our benchmarks (**diffpriv**) are monotone.

Definition 23. 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 an inequality 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. Equality relations are not allowed with* count *subterms.*

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.*

[1] Note that this is the worse case, as it implies that the maximal number of parameters have been removed, and the resulting lemma is thus the least-general.

Theorem 4.5. *Given a monotone clause set from* $\mathcal{L}_{\mathsf{count}}(\mathsf{QF\_LIA})$*, countersat terminates in a finite number of steps.*

Proof. To prove this result we invoke Theorem 4.4, showing that when the input clause set C_0 is monotone explain will only return lemmas from a finite set T_{C_0}. Furthermore, we only need to consider lemmas returned by explain when it is given a conflicting clause set containing at least one parametric count term; if explain generates lemmas for conflicts not involving counting theory, then it can use complete strategies for QF_LIA.

Consider a trail $[M, l]$ (or equivalently $[M, \neg l]$, but we focus on the former case w.l.o.g.) that conflicts with clause set C, and assume without loss of generality that the conflict arises due to l. Let α be the assignment implied by M. Assuming for the moment that there is only one count term in C, l must take the form: $\mathsf{count}(\ldots, \phi(x_1^s, \ldots, x_n^s)) \leqslant c$ for some constant c (see the first condition from Definition 23). Suppose that the constraint is satisfiable at some point $a_1, \ldots, a_n$, and that $\alpha(x_1) = v_1, \ldots, \alpha(x_n) = v_n$. Note that α will be total over the parameters $x_1, \ldots x_n$ appearing in ϕ, otherwise the conflict would not exist. explain, by its use of CAD, will produce a lemma of the form $\phi(x_1, \ldots, x_n) \implies \phi_1(x_1) \vee \cdots \vee \phi_n(x_n)$ where:

$$\phi_i(x_i) = \begin{cases} x_i > v_i & \text{if } v_i < a_i \\ x_i < v_i & \text{if } v_i > a_i \end{cases}$$

Because each chamber polynomial is monotone (see the second condition from Definition 23), this lemma will cause future model assignments to drive the valuation of the count term closer to c, i.e., for $1 \leqslant i \leqslant n$, $|x_i - a_i|$ will shrink in the next complete assignment to the parameters. This "shrinking" behavior will continue until $|x_i - a_i| = 0$, in which case the constraint is satisfied. In the worse case, explain will have generated lemmas corresponding for each integer between the initial v_i and a_i, for each parameter i, but this set of lemmas will always be a finite set.

On the other hand, if the term is not satisfiable, then one of two cases must hold:

1. The parametric enumerator takes a minimum at some value $c' > c$. In this case, explain will continue generating lemmas that drive the count term closer to c' (in the same manner as in the satisfiable case). Eventually after a finite number of intermediate lemmas (in the worst case, a similar sequence of lemmas that were generated in the satisfiable case), explain will generate a lemma that implies values for $x_1, \ldots, x_n$ such that $\mathsf{count}(\ldots, \phi(x_1^s, \ldots, x_n^s)) < c'$, but this will necessarily contradict all of the chamber constraints, leading to an UNSAT decision.

2. The constraints in M restrict $x_1, \ldots, x_n$ to a subspace over which the parametric enumerator takes some value $c' > c$. As in the previous case, explain will generate lemmas that drive the count term closer to c', generating a lemma that implies values for $x_1, \ldots, x_n$ such that $\mathsf{count}(\ldots, \phi(x_1^s, \ldots, x_n^s)) < c'$. This situation will result in a conflict, causing the procedure to backtrack to a point where the constraint becomes satisfiable (in which case explain will generate a finite sequence of lemmas leading to SAT), or terminating with UNSAT.

Finally, the case in which there are multiple count terms in the same constraint is simplified by the first and third constraints in Definition 23. Because the parameters in each count term are distinct, and the coefficient of their sign is the same, such a term can be rewritten as follows:

$$\mathsf{count}(\{y_1, \ldots, y_n\}, \phi(x_1^s, \ldots, x_n^s)) + \mathsf{count}(\{z_1, \ldots, z_m\}, \phi(x_1'^s, \ldots, x_n'^s)) \leqslant c$$
$$\implies \mathsf{count}(\{y_1, \ldots, y_n, z_1, \ldots, z_m\}, \phi(x_1^s, \ldots, x_n^s) \vee \phi(x_1'^s, \ldots, x_n'^s)) \leqslant c$$

Notice that the targets of the count $\{y_1, \ldots, y_n\}, \{z_1, \ldots, z_m\}$ might overlap. This situation is nothing more than a syntactic inconvenience, as these

variables can be rewritten arbitrarily without changing the meaning of the term (see Section 4.1 for a discussion of this). After rewriting, the results from the single-count term case apply, and the proof is complete. □

Implementing genparam. Implementing 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 genparam(C, x) using mesh-based black-box optimization (Abramson et al.). 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 (Luenberger and Ye, 2008) 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:

$$g(s_1^s, s_2^s) = (s_1^s - s_2^s)^2 + (b(s_1^s, s_2^s) - 120)^2$$

$b(s_1^s, s_2^s)$ 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: $7 \leqslant s_1^s, s_2^s \leqslant 13$. The size of the vicinity is heuristic; our implementation uses a region of size 7 in each dimension.

4.4 Case Study: Verifying Secure Multiparty Computations

In this section, we describe the application of countersat to verifying confidentiality properties in programs written for a secure multi-party computation (SMC) compiler called *Fairplay* (Ben-David et al., 2008). SMC is a cryptographic primitive, realized through a protocol, that allows a set of remote parties $P_1, \ldots, P_n$ to jointly compute a function f on their individual inputs, and learn its output $f(x_1, \ldots, x_n)$ without also learning the other parties' inputs. A secure multi-party compiler is an application that transforms programs written in a specialized high-level language into an executable SMC protocol. Lindell and Pinkas describe the properties common to most realizations of SMC in greater detail (Lindell and Pinkas, 2000):

1. *Privacy*: the only information that a given party learns about the other parties' inputs can be derived from knowledge of $f(x_1, \ldots, x_n)$.

2. *Correctness*: each party is guaranteed that they learn the correct value for $f(x_1, \ldots, x_n)$.

3. *Input independence*: the inputs provided by corrupted parties are independent of those provided by honest parties. For technical reasons having to do with the properties of certain cryptographic primitives, this property is a distinct requirement because it may not be guaranteed by privacy.

4. *Output delivery*: denial-of-service attacks are impossible, so corrupted parties cannot prevent honest parties from receiving $f(x_1, \ldots, x_n)$ when the protocol terminates.

5. *Fairness*: corrupted parties receive their outputs if and only if honest parties do.

These properties do not constitute a security definition, but are rather informal guidelines that SMC protocols should achieve. The security of an SMC protocol is formalized by requiring that it be *indistinguishable* from a functionally-equivalent protocol executed in an *ideal world*. In the ideal-world protocol, each party sends its input to an un-corruptable trusted third party, who computes $f(x_1, \ldots, x_n)$ and sends the result back to each party. An SMC protocol is said to be secure if, for any attack possible on the "real-world" protocol, there is an adversary who can mount the same attack on the ideal-world protocol. For a more detailed look at SMC protocols and their security, see (Cramer et al., 2012; Goldreich, 2004; Lindell and Pinkas, 2009).

Observe that this definition of security does not attempt to proscribe any specific inferences an adversary might make about the honest parties' inputs. While this observation might seem to make the definition so vague as to be of little practical use, it is in fact necessary to meaningfully define SMC over a general class of functions. The outputs given by some functions might inherently reveal a significant amount about their inputs, and by omitting a characterization of which functions are acceptable for SMC (or equivalently, what other parties should be allowed to learn of others' inputs aside from the output $f(x_1, \ldots, x_n)$), this definition achieves greater flexibility by leaving this matter to the user's better judgement.

However, as we have argued in this dissertation, it is not always straightforward to determine whether a function reveals too much about its inputs, or whether an implementation of a function correctly preserves an intended confidentiality property. To address this issue, we show how to use Bounded Software Model Checking (see Section 2.3 for background) alongside countersat to verify SMC programs against safety properties that characterize application-specific notions of confidentiality. For a given function, we assume that the compiler will ensure properties 1-5 above, and determine whether the function *as specified in the program text* models

```
program Millionaires {
    type int = Int<4>; // 4-bit integer
    type AliceInput = int;
    type BobInput = int;
    type AliceOutput = Boolean;
    type BobOutput = Boolean;
    type Output = struct {
        AliceOutput alice, BobOutput bob
    };
    type Input = struct {
        AliceInput alice,
        BobInput bob
    };
    function Output out(Input inp) {
        out.alice = inp.alice > inp.bob;
        out.bob = inp.bob > inp.alice;
    }
}
```

Figure 4.7: (From (Malkhi et al., 2004)) Example SFDL program for computing the Millionaires' problem between two parties. Each party provides an integer value representing their wealth as input, and learns whether they are wealthier than the other party.

an additional user-specific property (encoded as a safety property using $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$) that characterizes acceptable disclosure of the function's input values through its output.

The FairPlay Compiler

The Fairplay secure multi-party compiler (Ben-David et al., 2008; Malkhi et al., 2004) translates programs written in a specialized language called *Secure Function Definition Language* (SFDL) into a Boolean circuit representation, which can then be evaluated using a variant of Yao's garbled circuit protocol (Yao, 1982) extended to work for more than two parties (Beaver et al., 1990) to realize the SMC properties discussed above. SFDL supports variable-length integer, Boolean, and structured data types, basic arith-

```
0 input //alice$0
1 input //bob$0
2 gate arity 2 table [0110] inputs [0 1]
3 output gate arity 2 table [10] inputs [2] //alice$0
4 output gate arity 2 table [01] inputs [2] //bob$0
```

Figure 4.8: Circuit representation used by Fairplay. Each line represents a wire in the circuit, which is either an input wire or a Boolean gate of specified arity given by a truth table. This circuit returns the function $f(x_0, x_1) = \neg(x_0 \oplus x_1)$ to Alice (who inputs x_0) and $f(x_0, x_1) = x_0 \oplus x_1$ to Bob (who inputs x_1).

metic operations and relations, functions, assignments, conditionals, and loops. However, to ensure privacy of inputs, the number of loop iterations must be constant, so loops are just syntactic sugar for nested conditional statements, and recursive functions are not allowed. Similarly, the circuit produced by Fairplay will always execute both branches of a conditional. An example SFDL program is shown in Figure 4.7. This program computes a solution to Yao's famous Millionaires' problem (Yao, 1982), wherein two parties compare their numeric inputs using a simple inequality test.

Figure 4.8 shows an example of the circuit representation to which FairPlay compiles. This representation essentially gives a straight-line program over Boolean values, where each line specifies a distinct circuit wire corresponding to either an input wire, an internal gate, or an output gate. Wires are referenced by number, which is specified on the left of each line. Additionally, Fairplay annotates input and output wires with comments that signify their corresponding program variables. The gates correspond to Boolean functions whose arity is given on the corresponding line, and compute a value according to the table appearing after the arity specification. The table is interpreted by concatenating the gate's input wires, interpreting the resulting string as a little-endian binary number, and returning the table value at the corresponding index. For example, the gate numbered 2 in Figure 4.8 computes $x_0 \oplus x_1$, so if $x_0 = 0, x_1 = 1$, then we

look up the value at table index $(01)_2 = 2$ (remembering little-endianness), and return the value 1.

Model Checking Fairplay Programs

The uniformity of Fairplay's circuit representation makes it an attractive target for program analysis, as there are a small number of statement types for which we must define the symbolic post-state operator (see Section 2.3 for background on this topic). Additionally, the gates in Fairplay's circuits have natural representations as linear-integer constraints. To see why, consider the xor-gate (labeled 2) in Figure 4.8, which takes inputs x_0, x_1 and computes x_2. We will write a 0-1 linear-integer CNF formula $\phi(x_0, x_1, x_2)$ that is satisfiable only when $x_2 = x_0 \oplus x_1$ (assuming appropriate conversions between Boolean values and integers), starting with constraints that restrict each variable to take a value in $\{0, 1\}$:

$$\phi(x_0, x_1, x_2) \stackrel{\text{def}}{=} 0 \leqslant x_0 \leqslant 1 \wedge 0 \leqslant x_1 \leqslant 1 \wedge 0 \leqslant x_2 \leqslant 1$$

We now encode the truth table using a single clause for each entry. Observe that the first entry corresponds to the Boolean formula:

$$(\neg x_0 \wedge \neg x_1) \implies \neg x_2 \;\; \textit{or simply,} \;\; x_0 \vee x_1 \vee \neg x_2$$

We can encode this over 0-1 integers using addition, subtraction, and an inequality test. When we do this for each entry in the truth table, we arrive at:

$$\begin{aligned} \phi(x_0, x_1, x_2) \stackrel{\text{def}}{=} \;& 0 \leqslant x_0 \leqslant 1 \wedge 0 \leqslant x_1 \leqslant 1 \wedge 0 \leqslant x_2 \leqslant 1 \\ & \wedge\, x_0 + x_1 + (1 - x_2) \geqslant 1 \\ & \wedge\, x_0 + (1 - x_1) + x_2 \geqslant 1 \\ & \wedge\, (1 - x_0) + x_1 + x_2 \geqslant 1 \\ & \wedge\, (1 - x_0) + (1 - x_1) + (1 - x_2) \geqslant 1 \end{aligned}$$

Thus, we view Fairplay circuits as programs with variables that take values over the set $\mathbb{Z}_{0,1} = \{0, 1\}$. We also assume that each circuit ends with an empty statement $\perp$.

Defining Post_i**.** In the following, let $b(i, j)$ correspond to the j*th* bit of the little-endian binary representation of i. We associate a label ℓ_i with the i*th* line of a given circuit program, and a unique label $\ell_\perp$ to refer to the final statement. We can then define the symbolic-post state operator in four cases. The first corresponds to input wire statements:

$$\mathsf{Post}_i(n \texttt{ input}, \phi) = 0 \leqslant x_n \leqslant 1$$

Notice that the vocabulary index i is ignored in this definition, because Fairplay's circuit programs already have an implicit vocabulary structure suitable for model checking: each statement defines and (possibly) makes an assignment to exactly one variable, which is never re-assigned in later statements. Let s refer to the statement:

$$w_c \texttt{ gate arity } n \texttt{ table } [o_0 \dots o_{2^n-1}] \texttt{ inputs } [w_0 \dots w_{n-1}]$$

Generalizing the conversion from gate statements to linear-integer arithmetic outlined above, we have:

$$\mathsf{Post}_i(s, \phi) = 0 \leqslant w_c \leqslant 1 \ \wedge \phi_0 \wedge \dots \wedge \phi_{2^n-1}$$
$$\textit{where } \phi_j \stackrel{\text{def}}{=} 1 \leqslant 1 - o_j - w_c + 2o_j w_c + \sum_{k=0}^{n-1} b(j, k) + w_k - 2w_k b(j, k)$$

The operator for `output gate` statements is identical. Then the operator for the composition of statements $s_1 \dots s_n$ is:

$$\mathsf{Post}_i(s_1 \dots s_n, \phi) = \mathsf{Post}_{i+n}(s_n, \mathsf{Post}_{i+n-1}(s_{n-1}, \cdots \mathsf{Post}_i(s_1, \phi) \cdots))$$

Finally, we define the operator for the empty statement:

$$\mathsf{Post}_{\mathsf{i}}(\bot, \phi) = \phi$$

Notice that for each statement type, $\mathsf{Post}_{\mathsf{i}}$ will return a CNF formula, so these formulas can be used directly with countersat.

Defining Safety Properties with $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$. We check Fairplay programs against safety properties defined using $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$ formulas (see Section 2.3 for background on safety properties). Because Fairplay circuits output values once at the end of execution, and our properties concern the information that can be inferred from output values, these properties only ever pertain to $\ell_\bot$—the label corresponding to the final statement. Given an $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$ formula ϕ, we verify properties Π_ϕ of the form:

$$\Pi_\phi = \{(\ell_\bot, \rho) \mid \rho \in [\![\phi]\!]\}$$

where $[\![\phi]\!]$ uses the definition of satisfiability for $\mathcal{L}_{\#}(\mathsf{L})$ given in Definition 19.

Example 4.6. *Suppose that Alice wishes to verify non-interference (Denning and Denning, 1977) between her input x_0 in Figure 4.8 and Bob's input and output wires, which correspond to variables x_1 and x_4, respectively. Non-interference states that no information about the sensitive variable (in this case, x_0) is communicated to the selected "low-sensitivity" variables (in this case, the variables that Bob can see, x_1 and x_4). Let $\phi(x_0, x_1, x_2, x_3, x_4) = \mathsf{Post}_k(s_0 s_1 s_2 s_3 s_4, \mathsf{True})$ for a sufficiently large k.*[2] *We state non-interference as a $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$-safety property as follows:*

$$\Pi = \{(\ell_\bot, \rho) \mid \rho \in [\![\mathsf{count}(x_0, \phi(x_0, x_1^s, x_2, x_3, x_4^s)) \geqslant 2]\!]\} \tag{4.3}$$

[2]We discuss selecting the trace bound in the next subsection.

Notice that if when we are given values for x_1 and x_2 "observed" by Bob, there are fewer than 2 possible models for x_0, then Bob has learned something about Alice's sensitive variable. Conversely, if there are 2 possible values, then Bob has learned nothing, and non-interference holds. In the $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ formula describing this property, x_1 and x_4 are counting parameters to reflect the fact that Bob knows their values, so the corresponding counting-satisfiability problem is defined over observations that Bob can make that would result in a violation of non-interference.

The role of Bob's observations (i.e., the counting parameters) is crucial in this property. To see why, consider what would happen if we were to make the only parameter Bob's output x_4. Recalling that the formula for x_4 in Figure 4.8 is given by $x_4 \leftrightarrow x_0 \oplus x_1$, we see that the parametric enumerator for the count *term in the above property is the constant 2 everywhere because for both possible values of x_4, we can always find a value for x_1 that makes the formula* True *for* both *values of x_0:*

$$\forall x_4.\exists x_1.(x_4 \leftrightarrow (x_0 \oplus x_1)) \vee (x_4 \leftrightarrow (\neg x_0 \oplus x_1))$$

In other words, this would fail to capture the flow of information between x_0 and x_4 that clearly exists in the program given Bob's knowledge of his input. The property given in Equation 4.3 has no such issue: for any values of x_1, x_4, there is exactly one value of x_0, which reflects the fact that this example does not have non-interference.

Notice that in Example 4.6, the $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ formula that defines non-interference contains the symbolic post-state $\mathsf{Post}_k(s_0 s_1 s_2 s_3 s_4, \mathsf{True})$ as a sub-formula. This inclusion is necessary, because the number of models x_0 can take depends on the program semantics. This dependence is common to the confidentiality properties that we verify as $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$-safety properties. However, it means that these properties cannot be specified without making explicit reference to the symbolic post-state, which tends

BMC for $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$-safety properties

Input: Fairplay ciruit program $s_0 \ldots s_n$
Safety property $\Pi_\phi = \{(\ell_\perp, \rho) \mid \rho \in [\![\phi]\!]\}$

1. Let $k \leftarrow n$
2. Let $\phi_{\mathsf{Post}} \leftarrow (\lambda \mathbf{p}.\phi)(\mathsf{Post}_k(s_0 \ldots s_n, \mathsf{True}))$
3. Return $\mathsf{countersat}(\neg\phi_{\mathsf{Post}})$

Figure 4.9: Procedure for performing bounded software model checking of $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$-safety properties for Fairplay circuits. $\mathbf{p}$ is a placeholder for the symbolic post-state formula that must appear within count terms to specify confidentiality properties. $\mathbf{p}$ is replaced with the actual post-state after it is computed in step 2.

to make them large and difficult to understand. To address this issue, we adopt a convention where any reference to the symbolic post-state in a $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$-safety property is made with a placeholder $\mathbf{p}$, which can be replaced with the correct formula automatically later in the analysis.

Putting it all together. While bounded model checking is typically incomplete, this need not be the case with Fairplay ciruits because they always yield finite computations: there are no loops or other non-sequential control transfer, and only a finite number of statements. We can always select a value for k that results in sound and complete verification by ensuring that k is no smaller than the number of statements in the circuit program.

Figure 4.9 shows our procedure for performing bounded model checking in this setting. We set the depth bound k to the number of statements in the program, compute the k-step symbolic post-state, rewrite the property by replacing the placeholder $\mathbf{p}$ with the post-state formula, and check the negated result for satisfiability using countersat.

editdist	Two-party edit distance. Verify that result should not leak contents of a full character from party's string.
diffpriv	Compute the cardinality of a set, represented as a binary string, verifying differential privacy.
median	Median of two partys' lists. Verify that one party cannot learn elements of the other party's input that are not returned as the median.
auction	Vickrey auction. Verify that the result does not leak more a player's exact bid.
auction-circ	Same as above, but translated directly from circuit.
mill-circ	Millionaire's problem. Verify that the result does not leak a player's exact wealth.
manymill-circ	Multi-party variant of above.
keydb-circ	Keyed database lookup. Result should leak no information about unmatched rows, i.e., non-interference on unmatched rows.
voting-circ	Simple majority voting circuit. A coalition of fewer than half minus one should not learn another party's vote.

Figure 4.10: Benchmarks used to evaluate the performance of bounded software model checking with countersat. Names ending with "-circ" correspond to instances verified from a Fairplay circuit representation. The remaining benchmarks were encoded in $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ by hand.

Benchmarks

We evaluated countersat on nine benchmarks, the details of which are given in Figure 4.10. 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 (Ben-David et al., 2008). Each of the verification benchmarks suffixed with **-circ** was generated directly from the circuit program compiled by Fairplay.

The remaining verification benchmarks were translated from Fairplay by hand, without being first reduced to Boolean operations. We used hand-written symbolic post-states on some benchmarks because the process of compiling a program into a circuit introduces a large number of new variables and constraints into the symbolic post-state that are not present in the post-state computed directly from the high-level source code. By verifying both representations, we aim to measure the extent to which the number of intermediate variables affects model checking.

Two-party secure edit-distance (editdist). We verified a two-party edit distance program adapted from one written for comparing genome sequences (Jha et al., 2008). When compiled using Fairplay, the resulting circuit was too large for countersat to verify, so we hand-coded the symbolic post-state for the original SFDL program; without reducing integers to Boolean wires, countersat was able to verify the resulting formula. The safety property that we verified states that the second party (Bob) should be unable to learn the first party's (Alice) exact input unless they give the same input:

$$\mathrm{in}_{\mathrm{alice}} = \mathrm{in}_{\mathrm{bob}} \vee \mathsf{count}(\mathrm{in}_{\mathrm{alice}}, \mathbf{p}(\mathrm{in}_{\mathrm{alice}}, \mathrm{in}^{\mathbf{s}}_{\mathrm{bob}}, \mathrm{out}_{\mathrm{alice}}, \mathrm{out}^{\mathbf{s}}_{\mathrm{bob}})) \geqslant 1$$

Notice that Bob's input and output variables were made shared variables in the formula.

Differentially-private set cardinality (diffpriv). We wrote a simple DP program for counting the number of elements in a set. The set is represented as a bit string, with one bit corresponding to each element; however, it is important to note that the ordering of the bits does not matter for this program. Differential privacy is implemented by applying a transform to a uniformly-distributed integer input r. The $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ formula used to

encode the needed safety property is:

$$\neg(\mathrm{out}^{s} = \mathrm{out}'^{s} \wedge \sum \mathrm{in}^{s} - \sum \mathrm{in}'^{s} \leqslant 1)$$
$$\vee\ \mathsf{count}(r, \mathbf{p}(r, \mathrm{in}^{s}, \mathrm{out}^{s})) \leqslant \epsilon \times \mathsf{count}(r', \mathbf{p}(r'^{s}, \mathrm{in}'^{s}, \mathrm{out}'^{s}))$$

Note that this is an implication, and that the property is defined over two sets of program variables ("primed" and "unprimed"), as we have to reason about pairs of "neighboring" executions (see Section 2.1 for background on this topic). The antecedent states that the output produced in both executions matches, and that their inputs differ in only one position. The consequent asserts the DP probability bound, as the only random value used in the computation is the uniform-random r. The other variables are shared, so the $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ satisfiability problem corresponds to a search over neighboring inputs and their corresponding outputs that violate the bound.

Two party median (median). This benchmark accepts a list of integers from Alice and Bob, and computes the median of their combined list. We verify that Bob cannot learn any element from Alice's list that is not returned as the median. For the i*th* element of Alice's list, the corresponding formula is:

$$\mathrm{in}_{\mathrm{alice}}[i] = \mathrm{out}^{s}_{\mathrm{bob}} \vee \mathsf{count}(\mathrm{in}_{\mathrm{alice}}[i], \mathbf{p}(\mathrm{in}_{\mathrm{alice}}, \mathrm{in}^{s}_{\mathrm{bob}}, \mathrm{out}_{\mathrm{alice}}, \mathrm{out}^{s}_{\mathrm{bob}})) \geqslant N$$

where N is the number of possible values each element can take.

Vickrey auction (auction and auction-circ). This auction program is distributed with Fairplay. We verify that no other party can learn the i*th* party's exact bid:

$$\mathsf{count}(\mathrm{in}_{i}, \mathbf{p}(\mathrm{in}^{s}_{1}, \ldots, \mathrm{in}^{s}_{n}, \mathrm{out}^{s}_{1}, \ldots, \mathrm{out}^{s}_{n})) \geqslant N$$

countersat correctly found that this property does not hold, because there are corner cases that allow players to learn each others' bids:

- If the second-price bid is 0 (i.e., the lowest possible bid), then the winner knows the other parties' bids.
- If the second-price bid is the maximum bid and the winner is ordered first in the comparison used to find the winner, then the other parties learn that the first party made the maximum bid as well.

However, these conditions are easily excluded from the safety property.

Millionaire's Problem (mill-circ and manymill-circ). We verified Yao's famous two-party Millionaire's problem (Yao, 1982) and a three-party variant of it. This problem involves a simple inequality test, or in the three-party case, a sequence of inequality tests. We verify that no party learns another party's exact input, which makes the corresponding $\mathcal{L}_{\#}(\mathsf{QF\_LIA})$ formula the same as the one used for the Vickrey auction. However, as with the Vickrey auction, corner cases invalidate this property for extremal inputs.

Keyed Database Lookup (keydb-circ). In this benchmark, Alice provides a key value, and Bob provides a key-value store as input. The program returns the corresponding entry from Bob's store. We verify that entries from Bob's store that do not match Alice's key remain completely unknown to Alice. For the i*th* element of Bob's store, the corresponding formula is:

$$\mathrm{out}^{\mathsf{s}}_{\mathrm{alice}} = \mathrm{in}[\mathrm{i}].\mathrm{value}_{\mathrm{bob}} \vee \mathsf{count}(\mathrm{in}[\mathrm{i}]_{\mathrm{bob}}, \mathbf{p}(\mathrm{in}^{\mathsf{s}}_{\mathrm{alice}}, \mathrm{in}_{\mathrm{bob}}, \mathrm{out}^{\mathsf{s}}_{\mathrm{alice}}, \mathrm{out}_{\mathrm{bob}})) \geqslant \mathsf{N}$$

This property holds for the circuit compiled by Fairplay.

	With advice			No adv.	No CAD	NLSAT
	count	*advice*	***total***	***total***	***total***	***total***
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

Figure 4.11: Performance characteristics for countersat. All times are measured in seconds. – signifies timeout, NA signifies that the solver returned "unknown". *count* refers to the amount of time spent in Barvinok's algorithm, *advice* to the amount of time spent in genparam, **No adv.** to a configuration with genparam disabled, **No CAD** to a configuration with CAD lemmas disabled, and **NLSAT** to a configuration that produces QF_NRA instances and sends them directly to Z3 for a solution.

Multi-party election (voting-circ). This benchmark comes from a voting protocol distributed with Fairplay. n parties vote between two candidates, and on termination, all parties learn which candidate won. We verify that a coalition of fewer than $\lfloor \frac{n}{2} \rfloor - 1$ is unable to learn another party's vote. The property to verify the i*th* party's confidentiality is:

$$\mathsf{count}(\mathrm{in}_i, \mathbf{p}(\mathrm{in}_1^s, \ldots, \mathrm{in}_{\lfloor \frac{n}{2} \rfloor - 1}^s, \mathrm{in}_{\lfloor \frac{n}{2} \rfloor}, \ldots, \mathrm{in}_n, \ldots, \mathrm{out}_1^s, \ldots, \mathrm{out}_n)) \geqslant 2$$

Here we assume that parties $1 \ldots \lfloor \frac{n}{2} \rfloor - 1$ form a coalition that i is not a member of (so $i \geqslant \lfloor \frac{n}{2} \rfloor$). Members of the coalition are assumed to share their input and output values with each other, so their corresponding variables are shared.

Results

We evaluated the performance of countersat on these benchmarks, seeking answers to the following questions:

- *Can countersat be feasibly 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 our benchmarks, 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 somewhat helpful for performance in one case (**median**), while in others causing a modest slow-down in most others.

Setup. 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 4.11. 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.

Discussion. countersat was able to complete nearly all of the benchmarks, taking anywhere from a few seconds to several minutes to finish. When **editdist**, **diffpriv**, and **median** were translated into Boolean circuits by FairPlay, countersat timed out in the model-counting phase. As discussed previously, we addressed this by translating the code into $\mathcal{L}_{\mathsf{cnt}}(\mathsf{QF\_LIA})$ by hand, treating integers as single variables rather than as a set of binary digits. In this form, countersat was able to complete verification. Notice the speedup from **auction-circ** to **auction**; this confirms our suspicion that the Boolean encoding introduces a significant overhead in counting. In future work we will explore approximation methods, as well as formula decomposition strategies that utilize the inclusion-exclusion principle, to mitigate this bottleneck on larger benchmarks.

The results indicate that 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 for 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 feasible, 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 finding is most likely 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.

Summary and Conclusion

We introduced a problem called *satisfiability modulo counting*, which poses a satisfiability problem over a logic extended with parametric model-counting terms. The model-counting terms can range over formulas in arbitrary *base logics*, provided that it is possible to define two oracle functions for the base logic. We then developed an abstract decision procedure $\mathsf{sat}^{\#}$ for solving instance of this problem, again leaving the base logic unspecified but requiring implementations of three black-box routines that take the place of the oracles used to define the logic. Although we showed that satisfiability modulo counting is undecidable in the general case, we also showed that $\mathsf{sat}^{\#}$ is always sound, and is complete under certain well-specified conditions. We finished by discussing an implementation of this procedure for counting over linear-integer arithmetic, and showed that it can be used to perform bounded model checking over privacy-preserving programs written for a special-purpose cryptographic compiler.

This approach to privacy verification is still in its early stages of devel-

opment. It does not yet scale to many modest-sized programs, so future efforts in this area should remain focused on increasing scalability. The most promising avenues toward this goal lie in developing useful approximations for parametric model counting, as well as better program-analysis techniques to utilize this type of decision procedure.

5 CONCLUSION

Privacy remains a growing concern as applications generate, consume, and distribute new types of sensitive and valuable information. As the users of these applications become more aware of the issue, their natural response is to exercise greater caution when disclosing information to applications, as well as when selecting which applications to use. Responsible developers should respond in kind by producing applications that make users' information less vulnerable to exposure, granting control over which parties have access to it and what they are allowed to do with it.

However, as we have shown in this dissertation, the possibility of inference attacks often makes this task difficult for developers to accomplish—even when an application does not explicitly disclose a piece of sensitive information, a successful inference attack may allow an adversary to leverage other "benign" interactions to learn it nonetheless. Unlike traditional types of vulnerabilities, defensive strategies for inference attacks have not made their way into effective methods that developers can use. This dissertation proposed new techniques that begin to address this problem, giving developers better ways of understanding the balance of privacy and utility in their applications, as well as automatic methods to reason about the correctness of modern privacy protection mechanisms.

Much like traditional techniques for software-hardening that are based on bug-finding or searching for counterexamples, both of our contributions attempt to gain defensive insight through rigorous modeling of adversarial processes. Model Inversion served this purpose in our exploration of machine-learning applications, and parametric counting-safety properties did so when we looked at secure-multiparty computations. This approach has proven effective at mitigating the risks caused by a wide range of vulnerabilities in the past, and we expect it to do so for inference attacks in the future.

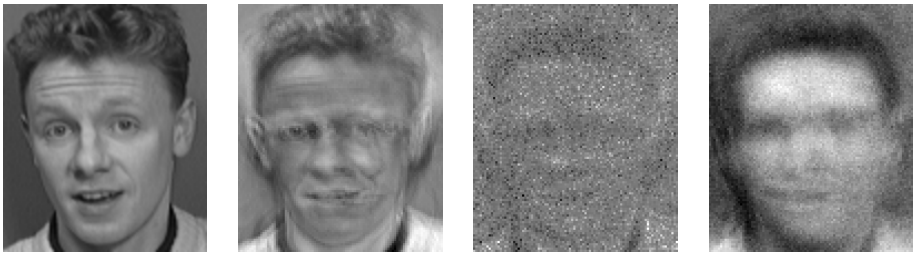
5.1 Ongoing and Future Work

The work presented in this dissertation suggests a number of extensions that need to be addressed in future work. We have initiated work on some of the most pressing issues, and identified several other directions that we hope will lead to substantial contributions.

Model Inversion

Numerous Large-Domain Unknowns. The model-inversion strategy that we presented in Chapter 3 is effectively limited to settings in which the number of unknown features is relatively small, and each unknown feature has a small domain. If these conditions do not hold, then the cost of marginalizing over unknowns, and optimizing the likelihood estimate over the target features, makes the algorithm intractable. This issue is not just academic, because ML models with these properties are becoming increasingly common in applications that perform object recognition in images. Such applications become potential privacy risks when they are used for facial recognition (Taigman et al., 2014; Huang et al., 2012; Lee et al., 2009), as a successful model-inversion attack may be able to identify individuals used to train the corresponding model.

We have begun to explore this issue in the context of facial recognition models. We trained a *softmax regression model*—a generalization of logistic regression that allows the class variable to take more than two values—on a corpus containing ten images each of 40 individuals (AT&T Laboratories Cambridge, 1998). Each image contains approximately 10,000 floating-point features representing pixel intensities, and we assume that the adversary only knows the nominal-valued response feature (corresponding to one of the 40 labels used in training). The set of target features that the adversary wishes to "invert" then corresponds to all of the pixel intensity features, and we assume a uniform prior distribution on these



Target **Softmax** **MLP** **DAE**

Figure 5.1: White-box model inversion results on a range of facial recognition models. Each image was generated by deriving a cost function from a recognition model, computing its explicit gradient, and running gradient descent until the corresponding image yielded cost below a fixed threshold.

features. Thus, the strategy from Chapter 3 amounts to brute-force search over many large-domain features, and a better approach is needed.

Most facial-recognition models return *confidence scores* for each possible response feature, so they are of the form $f : [0,1]^n \mapsto [0,1]^m$, where the function value represents the likelihood that the input (a vector of n pixel intensities) takes each particular response value (out of m total values). For a given response value $0 \leqslant i \leqslant m-1$, we can thus compute a *cost function* that represents the *distance* of given pixel vector x from response i: $c(x) = 1 - f(x)[i]$, where $f(x)[i]$ represents the i*th* component of f's output. Using approximate methods, we can then run black-box gradient descent on $c(x)$ to compute a pixel vector that, according to the facial-recognition model f, is likely to represent the i*th* individual from the training set. Somewhat surprisingly, this simple algorithm works on softmax facial-recognition models; an example result is shown in Figure 1.1 (Chapter 1).

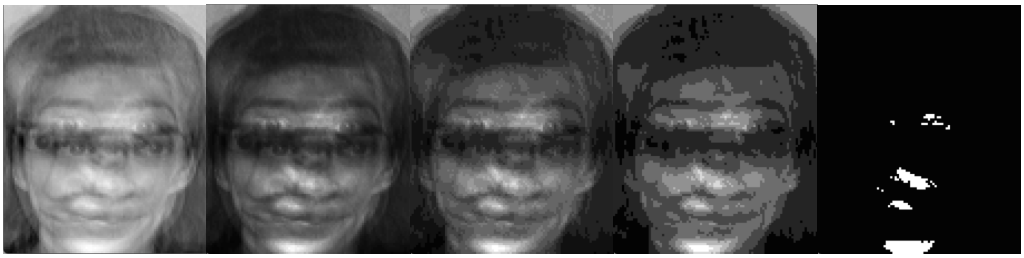
White-box attacks. While black-box gradient descent performed well on the softmax facial-recognition model, it does not produce comparable results on more sophisticated models. We applied the algorithm on multilayer-perceptron (MLP) and stacked denoising autoencoder models

(DAE), both trained on the same set of images, and found that a significantly larger number of gradient-descent iterations are required to reconstruct an approximation to one of the training images from these models. Because each numeric gradient approximation requires on the order of $2n$ calls to the cost function (where n is the dimension of the feature space), each of which takes approximately 70 milliseconds to complete on current hardware, a single MLP or DAE attack would take between 50 and 80 days to complete.

In a *white-box* setting, where the attacker has access to an explicit representation of the model rather than black-box information, it is possible to compute an *exact* gradient. This situation makes the attack possible on these more sophisticated models, the results of which are shown in Figure 5.1. Although the images reconstructed from the MLP and DAE models are not as crisp as those generated from the softmax models, experiments involving Mechanical Turk workers indicate that they are in many cases sufficient to re-identify the original training subject (Fredrikson et al., 2015).

These early results indicate that white-box adversaries may be able to infer substantially more sensitive information than their black-box counterparts. Characterizing the differences in capabilities between these two types of adversary in more precise, generally-applicable terms, remains important future work.

Countermeasures. While the case study in Chapter 3 showed that differential privacy is not a suitable countermeasure against model inversion with linear warfarin-dosing models, this result is not conclusive evidence that it is impossible to prevent MI without sacrificing utility in other settings. For example, we found that the black-box attack on the softmax facial-recognition model admits a simple countermeasure that is unlikely to affect utility in most applications: rounding the confidence scores re-



no rounding $\quad r = 0.001 \quad r = 0.005 \quad r = 0.01 \quad r = 0.05$

Figure 5.2: Black-box face MI attack on softmax models that round confidence scores to the nearest r rounding level r. The attack fails to produce a non-empy image at $r = 0.1$, thus showing that rounding yields a simple-but-effective countermeasure.

turned by the model. The results are presented in Figure 5.2 for rounding levels $r = \{0.001, 0.005, 0.01, 0.05\}$; the attack failed to produce an image for $r = 0.1$. "No rounding" corresponds to returning raw 64-bit floating-point confidence scores. Notice that even at $r = 0.05$, the attack fails to produce a recognizable image. While this approach may be specific to this particular type of model, building more general MI countermeasures is important future work.

Satisfiability Modulo Counting

Approximate parametric counting. One of the largest bottlenecks that we encountered when applying countersat to bounded model checking in Chapter 4 occurred during parametric model counting. When we attempted to verify the Fairplay circuit for two of our benchmarks (**editdist** and **median**), countersat timed out after 30 minutes. One way to address this problem is through more efficient approximate counting techniques. However, the approximations given by such a technique need to satisfy two key properties to remain useful to our abstract decision procedure $\mathsf{sat}^{\#}$:

1. *Sound upper- or lower-bounds*. For $\mathsf{sat}^{\#}$ to remain sound, it needs to know whether the approximate result is *definitely* an upper-bound or definitely a lower-bound. If the approximate result is not accurate enough to prove satisfiability or unsatisfiability, then it is acceptable for the procedure to spend more time counting, or to return "unknown". However, if the approximate result differs from the true result, but the sign of its divergence is unknown, then $\mathsf{sat}^{\#}$ can only be "probably" sound.

2. *Efficient refinement.* As $\mathsf{sat}^{\#}$ learns new lemmas, or finds that approximate counting results are not accurate enough for sound termination, it should be possible to efficiently refine the approximation to incorporate new information.

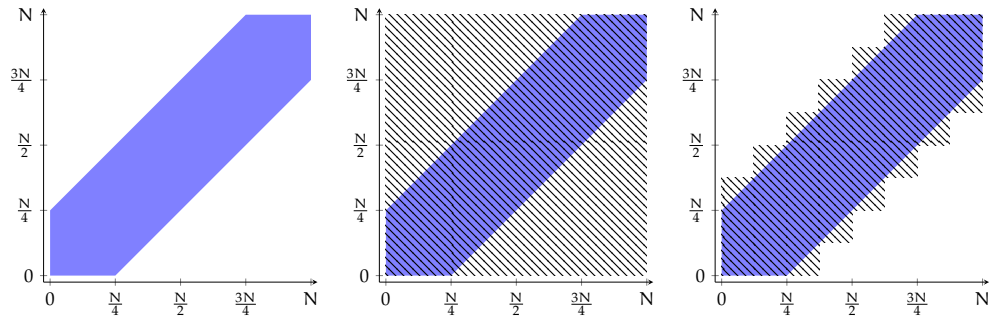
One possible approach is to approximate parametric polytopes using sets of bounding or inscribed parametric hypercubes, for upper- and lower-bounds, respectively. To see how this might work, consider the parametric polytope in Equation 5.1.

$$P(N) = \{[x, y] : y \leqslant x + 1 \wedge y \geqslant x - 1 \wedge 0 \leqslant x, y \leqslant N\} \tag{5.1}$$

This corresponds to the region in Figure 5.3(a) with six vertices: $\{(0,0)$, $(0,1)$, $(1,0)$, (N,N), $(N-1,N)$, $(N,N-1)\}$. As shown in Figure 5.3(b), we can over-approximate this polytope using a single parametric hypercube of the form:

$$H(N) = \{[x, y] : 0 \leqslant x, y, \leqslant N\} = [0, N] \times [0, N] \tag{5.2}$$

In general, we can find such an approximation by observing that the facets of the bounding hypercube each intersect with one facet of the polytope, so we can solve a linear system for each hypercube facet to arrive at the parametric hypercube. Thus, we would assume that the bounding



(a) (b) (c)

Figure 5.3: The parametric polytope from Equation 5.1 displayed in (a), and two successive over-approximations by parametric hypercubes (shown in the shaded regions of (b) and (c)). The over-approximation in (b) corresponds to Equation 5.2. The over-approximation shown in (c) is closer to the original polytope in terms of lattice-point count, and can be obtained efficiently from the over-approximation shown in (b) by subdividing the previous parametric hypercube.

hypercube will be of the form:

$$H(N) = [a_{x_l}N + b_{x_l}, a_{x_u}N + b_{x_u}] \times [a_{y_l}N + b_{y_l}, a_{y_u}N + b_{y_u}] \qquad (5.3)$$

for some values $a_{x_l}, a_{x_u}, b_{x_l}, b_{x_u}, a_{y_l}, a_u, b_{y_l}, b_{y_u}$ that we can find by fixing N at a sufficient number of distinct values $N_1, N_2, \ldots$, finding the corresponding non-parametric bounding hypercubes for $P(N_1), P(N_2), \ldots$, and solving the resulting system of linear equations.

This approach admits an efficient refinement scheme, wherein we subdivide each bounding hypercube in the current over-approximation into a disjoint set of covering hypercubes; an example is shown in Figure 5.3(c). This approach allows us to avoid re-computing facets corresponding to the boundaries of previous refinements. However, this approach is slightly complicated by the fact that parametric polytopes reside in a set of distinct *chamber spaces* (see Section 2.4) which change the polytope's facets, so the approximation must be computed for each chamber space. Developing

```
1 void pwcheck(int pw[2], int guess[2]) {
2   int match = 1;
3   assume(0 <= pw[0] <= 15 && 0 <= pw[1] <= 15);
4   for(int i = 0; i < 2; i++)
5     if(pw[i] != guess[i])
6       match = 0;
7 }
```

Figure 5.4: Example program: simple two-character password checker.

this approach, and extending it to under-approximations as well, is a promising direction for future work.

CEGAR model checking of $\mathcal{L}_{\#}(\mathsf{L})$-safety. Counterexample-guided abstraction refinement (Clarke et al., 2003) (CEGAR) is approach to symbolic model checking that incrementally builds the program abstraction. Starting with a coarse abstraction that over-approximates a program's state space, it searches for program paths that lead to possible property violations (called *counterexamples*), and on finding a *spurious* counterexample (i.e., a false positive), uses information from the path to refine the abstraction. This approach gains efficiency by only building parts of the abstraction that are needed to prove a given property, and using *path-local reasoning* to avoid expensive satisfiability-checking operations on the entire program.

Adapting CEGAR to the verification of $\mathcal{L}_{\#}(\mathsf{L})$-safety properties is important future work if the approach described in Chapter 4 is to scale to larger programs, as it will allow the model-counting component of $\mathsf{sat}^{\#}$ to focus on smaller subcomponents of the program. However, this adaptation is a challenging goal. Consider the program in Figure 5.4, which checks a simple two-character password given in the argument `pw` against a guess given in argument `guess`. Assume that the value of `pw` is unknown to the party that provides the value of `guess`, and suppose that we wanted to

verify that the following $\mathcal{L}_{\#}(\mathsf{L})$-safety property holds:

$$\texttt{match} = 1 \vee \mathsf{count}(\{\texttt{pw}_0, \texttt{pw}_1\}, \mathbf{p}) \geqslant 255$$

Recalling that $\mathbf{p}$ is a placeholder for the symbolic post-state formula, this property says that the number of assignments to pw must be at least $16 \times 16 - 1 = 255$, unless the password is guessed correctly. Note that we would treat the variables $\texttt{guess}_0$, $\texttt{guess}_1$, and match as counting parameters in this setting. The program in Figure 5.4 satisfies this property, as an adversary who observes these parameters will be able to exclude only one possible password if $\texttt{match} \neq 0$.

If we want to verify this property using the path-local reasoning from CEGAR, then we would look for individual paths that could lead to a valid counterexample. Suppose that we consider the path that is executed when $guess_0 \neq pw_0$ and $guess_1 = pw_1$. We would like to determine whether the path satisfies our property, so that we know whether it is a spurious counterexample or not. To do so, we would derive the following formula ϕ to embody the semantics of this path (essentially the symbolic post-state for the program given by this path):

$$\phi \stackrel{\text{def}}{=} 0 \leqslant pw_{0,1} \leqslant 15 \wedge pw_0 \neq guess_0 \wedge pw_1 = guess_1 \wedge match = 0$$

When we treat this formula as though it were the symbolic post-state in our case study from Chapter 4, we see that:

$$\mathsf{count}(\{pw_0, pw_1\}, \phi) = 15$$

However, this contradicts our property, from which we can only conclude that the path violates safety even though the program clearly does not! This demonstrates an interesting fact about $\mathcal{L}_{\#}(\mathsf{L})$-safety properties: whereas traditional safety properties are anti-monotone, i.e., if a set of paths P

satisfies a property then all $P' \subseteq P$ do as well, $\mathcal{L}_{\#}(L)$-safety properties are not. The fact that counting safety properties are not anti-monotone means that the path-local reasoning that underlies CEGAR techniques cannot be used for $\mathcal{L}_{\#}(L)$-safety verification. Finding a way to efficiently cope with this fact remains important future work.

REFERENCES

Abramson, M.A., C. Audet, G. Couture, J.E. Dennis, Jr., S. Le Digabel, and C. Tribes. The NOMAD project. Software available at: `http://www.gerad.ca/nomad`.

Acquisti, Alessandro, and Christina M. Fong. 2012. An Experiment in Hiring Discrimination Via Online Social Networks. *Social Science Research Network Working Paper Series*.

Anderson, Jeffrey L., Benjamin D. Horne, Scott M. Stevens, Amanda S. Grove, Stephanie Barton, Zachery P. Nicholas, Samera F.S. Kahn, Heidi T. May, Kent M. Samuelson, Joseph B. Muhlestein, John F. Carlquist, and for the Couma-Gen Investigators. 2007. Randomized trial of genotype-guided versus standard warfarin dosing in patients initiating oral anticoagulation. *Circulation* 116(22):2563–2570.

Armour, Stephanie. Data brokers come under fresh scruitny. *The Wall Street Journal*. February 12, 2014.

AT&T Laboratories Cambridge. 1998. The ORL database of faces. Available at: `http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`.

Backes, Michael, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic discovery and quantification of information leaks. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Barvinok, Alexander I. 1994. A polynomial-time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematical Operations Research* 19(4):769–779.

Bayardo, Roberto J., Jr., and Robert C. Schrag. 1997. Using CSP look-back techniques to solve real-world sat instances. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Beaver, D., S. Micali, and P. Rogaway. 1990. The round complexity of secure protocols. In *Proceedings of the ACM Symposium on the Theory of Computing*, 503–513.

Ben-David, Assaf, Noam Nisan, and Benny Pinkas. 2008. FairplayMP: a system for secure multi-party computation. In *Proceedings of the ACM Conference on Computer and Communications Security*.

Biere, Armin, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Proceedings of the Conference on Tools and Algorithms for Construction and Analysis of Systems*.

Bonate, Peter L. 2000. Clinical trial simulation in drug development. *Pharmaceutical Research* 17(3):252–256.

Brace, Larry D. 2001. Current status of the international normalized ratio. *Lab Medicine* 32(7):390–392.

Bryant, R.E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35(8):677–691.

Burch, J.R., E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. 1990. Symbolic model checking: 1020 states and beyond. In *Proceedings of the IEEE symposium on Logic in Computer Science*.

Carlquist, John F., Benjamin D. Horne, Joseph B. Muhlestein, Donald L. Lappe, Bryant M. Whiting, Matthew J. Kolek, Jessica L. Clarke, Brent C. James, and Jeffrey L. Anderson. 2006. Genotypes of the Cytochrome P450 Isoform, CYP2C9, and the Vitamin K Epoxide Reductase Complex Subunit 1 conjointly determine stable warfarin dose: a prospective study. *Journal of Thrombosis and Thrombolysis* 22(3).

Clarke, Edmund, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM* 50(5):752–794.

Clarke, Edmund M., Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. Cambridge, MA, USA: MIT Press.

Collins, George E. 1975. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In *Proceedings of the Conference on Automata Theory and Formal Languages*.

Cormode, Graham. 2011. Personal privacy vs population privacy: learning to attack anonymization. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining*.

Coudert, Olivier, Jean Christophe Madre, and Christian Berthet. 1991. Verifying temporal properties of sequential machines without building their state diagrams. In *Proceedings of the Workshop on Computer Aided Verification*.

Cramer, Ronald, Ivan Damgard, and Jesper Buus Nielson. 2012. Secure multiparty computation and secret sharing - an information theoretic appoach. Book draft, available at `http://www.daimi.au.dk/~ivan/mpc-book.pdf`.

Dalenius, T. 1977. Towards a methodology for statistical disclosure control. *Statistik Tidskrift* 15(429-444):2–1.

Davis, Martin, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commununications of the ACM* 5(7):394–397.

Davis, Martin, and Hilary Putnam. 1960. A computing procedure for quantification theory. *Journal of the ACM* 7(3):201–215.

De Moura, Leonardo, and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

Denning, Dorothy E., and Peter J. Denning. 1977. Certification of programs for secure information flow. *Communications of the ACM* 20(7): 504–513.

Dodis, Yevgeniy, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. 2008. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal of Computing* 38(1):97–139.

Dwork, Cynthia. 2006. Differential privacy. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*.

———. 2008a. An ad omnia approach to defining and achieving private data analysis. In *ACM Conference on Privacy, Security, and Trust in KDD*, 1–13.

———. 2008b. Differential privacy: a survey of results. In *Proceedings of the Conference on Theory and Applications of Models of Computation*.

Dwork, Cynthia, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Conference on Theory of Cryptography*.

———. 2011. Differential privacy: A primer for the perplexed. In *Proceedings of the Joint UNECE/Eurostat Work Session on Statistical Data Confidentiality*.

Fredrikson, Matt, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the ACM Conference on Computer and Communications Security (to appear)*.

Friedman, Arik, and Assaf Schuster. 2010. Data mining with differential privacy. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining*.

Fusaro, Vincent A., Prasad Patil, Chih-Lin Chi, Charles F. Contant, and Peter J. Tonellato. 2013. A systems approach to designing effective clinical trials using simulations. *Circulation* 127(4):517–526.

Goldreich, Oded. 2004. *Foundations of cryptography: Volume 2, basic applications*. New York, NY, USA: Cambridge University Press.

Haeberlen, Andreas, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential privacy under fire. In *Proceedings of the USENIX Security Symposium*.

Hamberg, A. K., Dahl, M. L., M. Barban, M. G. Srordo, M. Wadelius, V. Pengo, R. Padrini, and E.N. Jonsson. 2007. A PK-PD model for predicting the impact of age, CYP2C9, and VKORC1 genotype on individualization of warfarin therapy. *Clinical Pharmacology Theory* 81(4):529–538.

Hand, DavidJ., and RobertJ. Till. 2001. A simple generalisation of the area under the ROC curve for multiple class classification problems. *Machine Learning* 45(2):171–186.

Heusser, Jonathan, and Pasquale Malacaria. 2010. Quantifying information leaks in software. In *Proceedings of the Annual Computer Security Applications Conference*.

Hinman, P.G. 2005. *Fundamentals of mathematical logic*. Ak Peters Series, Wellesley, MA, USA: Taylor & Francis.

Holford, N., S. C. Ma, and B. A. Ploeger. Clinical trial simulation: A review. *Clinical Pharmacology Theory* 88(2):166–182.

Holford, N. H. G., H. C. Kimko, J. P. R. Monteleone, and C. C. Peck. 2000. Simulation of clinical trials. *Annual Review of Pharmacology and Toxicology* 40(1):209–234.

Huang, G.B., Honglak Lee, and E. Learned-Miller. 2012. Learning hierarchical representations for face verification with convolutional deep belief networks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*.

International Warfarin Pharmacogenetic Consortium. 2009. Estimation of the warfarin dose with clinical and pharmacogenetic data. *New England Journal of Medicine* 360(8):753–764.

Jaynes, E.T. 1982. On the rationale of maximum-entropy methods. *Proceedings of the IEEE* 70(9).

Jha, S., L. Kruger, and V. Shmatikov. 2008. Towards practical privacy for genomic computation. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Kamali, Farhad, and Hilary Wynne. 2010. Pharmacogenetics of warfarin. *Annual Review of Medicine* 61(1):63–75.

Kifer, Daniel, and Ashwin Machanavajjhala. 2011. No free lunch in data privacy. In *Proceedings of the ACM SIGMOD conference*.

Kim, M. J., S. M. Huang, U. A. Meyer, A. Rahman, and L. J. Lesko. 2009. A regulatory science perspective on warfarin therapy: a pharmacogenetic opportunity. *Journal of Clinical Pharmacology* 49:138–146.

Klebanov, Vladimir. 2012. Precise quantitative information flow analysis using symbolic model counting. In *Proceedings of the Workshop on Quantitative Aspects in Security Assurance*.

Klebanov, Vladimir, Norbert Manthey, and Christian Muise. 2013. SAT-based analysis and quantification of information flow in programs. In *Proceedings of Quantitative Evaluation of Systems*.

Köpf, Boris, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic quantification of cache side-channels. In *Proceedings of the Conference on Computer Aided Verification*.

Kosinski, Michal, David Stillwell, and Thore Graepel. 2013. Private traits and attributes are predictable from digital records of human behavior. *Proceedings of the National Academy of Sciences* 110(15):5802–5805.

Kovacs, Michael J., Marc Rodger, David R. Anderson, Beverly Morrow, Gertrude Kells, Judy Kovacs, Eleanor Boyle, and Philip S. Wells. 2003. Comparison of 10-mg and 5-mg warfarin initiation nomograms together with low-molecular-weight heparin for outpatient treatment of acute venous thromboembolism. *Annals of Internal Medicine* 138(9):714–719.

Kuruvilla, Mariamma, and Cheryle Gurk-Turner. 2001. A review of warfarin dosing and monitoring. *Proceedings of the Baylor University Medical Center* 14(3):305–306.

Lamport, L. 1977. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering* SE-3(2):125–143.

Lee, Honglak, Roger Grosse, Rajesh Ranganath, and Andrew Y. Ng. 2009. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the International Conference on Machine Learning*.

Lee, Jaewoo, and Chris Clifton. 2011. How much is enough? Choosing ϵ for differential privacy. In *Proceedings of the International Conference on Information Security*.

———. 2012. Differential identifiability. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining*.

LeFevre, Kristen, David J DeWitt, and Raghu Ramakrishnan. 2005. Incognito: Efficient full-domain k-anonymity. In *Proceedings of the ACM SIGMOD Conference*.

Lei, Jing. 2011. Differentially private M-estimators. In *Proceedings of the Conference on Neural Information Processing*.

Li, Ninghui, Wahbeh Qardaji, Dong Su, Yi Wu, and Weining Yang. 2013. Membership privacy: A unifying framework for privacy definitions. In *Proceedings of the ACM Conference on Computer and Communications Security*.

Lindell, Yehuda, and Benny Pinkas. 2000. Privacy preserving data mining. In *Proceedings of the Conference on the Theory and Applications of Cryptographic Techniques*.

———. 2009. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology* 22(2):161–188.

Luenberger, D.G., and Y. Ye. 2008. *Linear and nonlinear programming*. New York, NY, USA: Springer.

Malkhi, Dahlia, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay: A secure two-party computation system. In *Proceedings of the USENIX Security Symposium*.

McSherry, Frank, and Kunal Talwar. 2007. Mechanism design via differential privacy. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. Washington, DC, USA.

Mironov, Ilya. 2012. On the significance of the least significant bits for differential privacy. In *Proceedings of the ACM Conference on Computer and Communications Security*.

de Moura, Leonardo, and Dejan Jovanović. 2013. A model-constructing satisfiability calculus. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*.

National Heart, Lung, and Blood Institute. Clarification of optimal anticoagulation through genetics. Available from: `https://www.clinicaltrials.gov/ct2/show/NCT00839657`. NLM Identifier: NCT00839657.

Nieuwenhuis, Robert, Albert Oliveras, and Cesare Tinelli. 2005. Abstract DPLL and abstract DPLL modulo theories. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning*.

———. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Joural of the ACM* 53(6):937–977.

Pixley, Carl. 1991. Introduction to a computational theory and implementation of sequential hardware equivalence. In *Proceedings of the Workshop on Computer Aided Verification*.

Pugh, William. 1994. Counting solutions to Presburger formulas: how and why. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*.

Ramirez, Edith, Julie Brill, Maureen K. Ohlhausen, Joshua D. Wright, and Terrell McSweeny. 2014. Data Brokers: A Call for Transparency and Accountability. A Report of the Federal Trade Commission, May 2014.

Rubinstein, Benjamin IP, Peter L Bartlett, Ling Huang, and Nina Taft. 2012. Learning in a large function space: Privacy-preserving mechanisms

for SVM learning. *Journal of Privacy and Confidentiality* 4(1). Available at: `http://repository.cmu.edu/jpc/vol4/iss1/4`.

Sconce, Elizabeth A., Tayyaba I. Khan, Hilary A. Wynne, Peter Avery, Louise Monkhouse, Barry P. King, Peter Wood, Patrick Kesteven, Ann K. Daly, and Farhad Kamali. 2005. The impact of CYP2C9 and VKORC1 genetic polymorphism and patient characteristics upon warfarin dose requirements: proposal for a new dosing regimen. *Blood* 106(7):2329–2333.

Sorensen, Sonja V., Sarah Dewilde, Daniel E. Singer, Samuel Z. Goldhaber, Brigitta U. Monz, and Jonathan M. Plumb. 2009. Cost-effectiveness of warfarin: Trial versus "real-world" stroke prevention in atrial fibrillation. *American Heart Journal* 157(6):1064 – 1073.

Taigman, Yaniv, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*.

U.N. Office of the High Commiss. for Human Rights. 2012. United Nations Free & Equal Factsheet: Criminalization. Available at: `http://www.ohchr.org/Documents/Issues/Discrimination/LGBT/FactSheets/unfe-30-UN_Fact_Sheets_Criminalization_English.pdf`. Retrieved June 17, 2015.

United States Congress. 1996. Health Insurance Portability and Accountability Act of 1996 (HIPAA). Public Law 104-191, U.S. Statutes at Large 110: 1936-2103.

Verdoolaege, Sven, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica* 48(1):37–66.

Vinterbo, Staal. 2012. Differentially private projected histograms: Construction and use for prediction. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*.

Wang, Liewei, Howard L. McLeod, and Richard M. Weinshilboum. 2011. Genomics and drug response. *New England Journal of Medicine* 364(12): 1144–1153.

Whirl-Carrillo, M., E.M. McDonagh, J. M. Hebert, L. Gong, K. Sangkuhl, C.F. Thorn, R.B. Altman, and T.E. Klein. 2012. Pharmacogenomics knowledge for personalized medicine. *Clinical Pharmacology & Therapeutics* 92(4):414–417.

Yao, Andrew C. 1982. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*.

Zhang, Jun, Zhenjie Zhang, Xiaokui Xiao, Yin Yang, and Marianne Winslett. 2012. Functional mechanism: regression analysis under differential privacy. In *Proceedings of the International Conference on Very Large Data Bases*.