AUTOMATED PROGRAMMING FRAMEWORKS FOR ANALYZING DIFFERENTIAL PRIVACY

A Dissertation in
Computer Science and Engineering

by
Yuxin Wang

© 2022 Yuxin Wang

Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

May 2022
The dissertation of Yuxin Wang was reviewed and approved by the following:

Danfeng Zhang  
Associate Professor of Computer Science and Engineering  
Dissertation Co-Adviser  
Co-Chair of Committee

Daniel Kifer  
Professor of Computer Science and Engineering  
Dissertation Co-Adviser  
Co-Chair of Committee

Gang Tan  
Professor of Computer Science and Engineering

Dinghao Wu  
Professor of Information Sciences and Technology

Chita R. Das  
Professor of Computer Science and Engineering  
Department Head of Computer Science and Engineering
Abstract

The accelerating growth of data has led to fruitful researches and real-world applications. While large datasets have benefited performance in many fields, recent incidents of data leakages and abuses have raised public concerns for data privacy. It has become a vital yet challenging task to balance individuals’ privacy and data utilization for both researchers and data analysts.

Among many attempts to tackle this challenge, differential privacy has become a de facto standard that provides a promising way to release individuals’ sensitive data in a privacy-preserving manner. However, designing differentially-private algorithms is notoriously difficult and error-prone. Significant errors have happened even in peer-reviewed papers and systems. Such mistakes have led to researches on automated analysis of differential privacy algorithms to aid developers in the system design process. However, the limitations of existing tools either make the analysis time-consuming or fail to analyze sophisticated systems designed for differential privacy.

In this dissertation, we propose a set of novel programming frameworks that target at three major aspects of automated analysis of differential privacy: verification, counterexample detection and program synthesis. For verification, we develop ShadowDP that embeds a novel proving technique named Shadow Execution to enable verification of a complex algorithm Report Noisy Max with very few annotations. Unlike prior works, ShadowDP is built upon standard program logics, making it easy to offload the verification of differential privacy to off-the-shelf verifiers. Our evaluations show ShadowDP is more efficient by orders of magnitude, compared with existing verifiers for differential privacy.

For counterexample detection when a system fails to satisfy differential privacy, we propose CheckDP, the first integrated framework to prove and disprove differential privacy. A novel bidirectional Counterexample-Guided Inductive Synthesis (CEGIS) is developed and embedded in CheckDP, enabling it to simultaneously generate a proof for correct systems, as well as a counterexample for incorrect systems.

Lastly, we develop DPGen, an automated synthesizer with customizable utility metrics for differential
privacy. DPGen employs a novel approach to generate sketch programs and models the synthesis problem as an optimization problem involving privacy and utility, making it flexible and efficient in generating differentially-private programs with different requirements.
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................................................................. viii

LIST OF TABLES ................................................................................................................... xi

Acknowledgments .................................................................................................................... xii

Chapter 1 Introduction ............................................................................................................ 1

Chapter 2 Background ............................................................................................................. 5
  2.1 Differential Privacy .................................................................................................... 5
  2.2 Randomness Alignment .............................................................................................. 6

Chapter 3 Related Work .......................................................................................................... 8
  3.1 Verification of Differential Privacy .............................................................................. 8
  3.2 Counterexample Detection For Differential Privacy ...................................................... 9
  3.3 Synthesizing Differentially Private Algorithms ............................................................. 10

Chapter 4 Proving Differential Privacy with Shadow Execution .................................................. 11
  4.1 Introduction ............................................................................................................... 11
  4.2 Motivating Example ................................................................................................... 12
  4.3 Approach Overview .................................................................................................... 15
  4.4 ShadowDP: Syntax and Semantics ............................................................................... 17
    4.4.1 Syntax ............................................................................................................. 17
    4.4.2 Semantics ........................................................................................................ 20
  4.5 ShadowDP: Type System ............................................................................................ 21
    4.5.1 Notations ......................................................................................................... 21
    4.5.2 Expressions ..................................................................................................... 22
    4.5.3 Commands ...................................................................................................... 23
    4.5.4 Aligned Variables ............................................................................................. 24
    4.5.5 Shadow Variables ............................................................................................. 27
    4.5.6 Sampling Command ......................................................................................... 28
  4.6 Target Language ......................................................................................................... 28
  4.7 Soundness ................................................................................................................. 30
  4.8 Implementation and Evaluation ................................................................................... 35
    4.8.1 Implementation ................................................................................................. 35
    4.8.2 Case Studies .................................................................................................... 36
    4.8.3 Experiments ..................................................................................................... 43
    4.8.4 Proof Automation ............................................................................................. 44
  4.9 Summary ................................................................................................................... 44
Chapter 5  An Automated and Integrated Approach for Proving Differential Privacy and Finding
Precise Counterexamples

- 5.1 Introduction ................................................................. 46
- 5.2 Background ............................................................... 48
  - 5.2.1 Privacy Proof and Counterexample ....................... 48
  - 5.2.2 Challenges ......................................................... 48
- 5.3 Motivating Examples .................................................... 49
- 5.4 Approach Overview ..................................................... 51
- 5.5 Program Transformation ............................................... 54
  - 5.5.1 Syntax ............................................................... 54
  - 5.5.2 Semantics .......................................................... 56
  - 5.5.3 Program Transformation ....................................... 57
  - 5.5.4 Checking Expressions .......................................... 58
  - 5.5.5 Checking Commands ........................................... 60
  - 5.5.6 Checking Sampling Commands ............................. 61
  - 5.5.7 Function Signature Rewrite .................................... 64
  - 5.5.8 Shadow Execution ............................................... 65
  - 5.5.9 Soundness .......................................................... 68
- 5.6 Proof and Counterexample Generation ............................ 71
  - 5.6.1 Verify Sub-Loop ............................................... 72
  - 5.6.2 Invalidate Sub-Loop ............................................ 73
  - 5.6.3 Integrating Verify and Invalidate Sub-Loops .......... 74
- 5.7 Implementation and Evaluation ...................................... 75
  - 5.7.1 Implementation ................................................... 75
  - 5.7.2 Case Studies ....................................................... 76
  - 5.7.3 Partial Sum ......................................................... 86
  - 5.7.4 SmartSum and BadSmartSum ............................... 86
  - 5.7.5 Experiments ....................................................... 88
- 5.8 Summary .................................................................... 91

Chapter 6  Automated Program Synthesis for Differential Privacy

- 6.1 Introduction ............................................................. 92
- 6.2 Background ............................................................ 94
  - 6.2.1 Particle Swarm Optimization (PSO) ....................... 94
  - 6.2.2 Sparse Vector Technique (SVT) ........................... 95
- 6.3 Overview .................................................................. 96
  - 6.3.1 Challenges ........................................................ 96
  - 6.3.2 Approach Overview ......................................... 98
- 6.4 Sketch Generation .................................................... 100
  - 6.4.1 Syntax of Source and Target Program ................. 101
  - 6.4.2 Adding Noise Locations to Source Code ............. 102
- 6.5 Synthesis and Optimization ....................................... 105
  - 6.5.1 Mechanism Synthesis Problem ......................... 105
  - 6.5.2 Mechanism Optimization Problem ...................... 110
    - 6.5.2.1 Counterexample Generation ....................... 111
    - 6.5.2.2 Mechanism Generation ............................... 112
LIST OF FIGURES

Figure 4-1 Verifying Report Noisy Max with ShadowDP. Here, q is a list of query answers from a database, and max is the query index of the maximum query with Laplace noise generated at line 3. To verify the algorithm on the top, a programmer provides function specification as well as annotation for sampling command (annotations are shown in gray, where Ω represents the branch condition). ShadowDP checks the source code and generates the transformed code (at the bottom), which can be verified with off-the-shelf verifiers.

Figure 4-2 Selective alignment for Report Noisy Max.

Figure 4-3 ShadowDP: language syntax.

Figure 4-4 ShadowDP: language semantics.

Figure 4-5 Typing rules for expressions.

Figure 4-6 Typing rules for commands and auxiliary rules for commands.

Figure 4-7 Transformation of expressions and commands for aligned and shadow execution, where ⋆ ∈ {α, ⌣}.

Figure 4-8 Transformation rules to the target language.

Figure 4-9 Semantics for the target language.

Figure 4-10 Verifying Sparse Vector Technique with ShadowDP. Annotations are in gray where Ω represents the branch condition.

Figure 4-11 Verifying Numerical Sparse Vector Technique with ShadowDP. Annotations are in gray where Ω represents the branch condition.

Figure 4-12 Verifying Partial Sum using ShadowDP. Annotations are shown in gray.

Figure 4-13 Verifying SmartSum algorithm with ShadowDP. Annotations are shown in gray.


Figure 5-2 CheckDP: language syntax.

Figure 5-3 Program transformation rules for expressions.

Figure 5-4 Program transformation rules for commands and auxiliary rules. Distinguished variable vₑ and assertions are added to ensure differential privacy.

Figure 5-5 Expressions rules for extending CheckDP with shadow execution. Differences that shadow execution introduce are marked in gray boxes.
Figure 5-6 Command rules for extending CheckDP with shadow execution. Differences that shadow execution introduce are marked in gray boxes. ........................................... 67

Figure 5-7 Overview of the verify-invalidate loop of CheckDP. ................................................... 72

Figure 5-8 Tentative alignments and invalidating inputs. ............................................................ 73

Figure 5-9 Standard Sparse Vector Technique and its transformed code, where underlined parts are added by CheckDP. The transformed code contains two alignment templates for $\eta_1$ and $\eta_2$: $A_1 = \theta[0]$ and $A_2 = (q[i] + \eta_2[i] \geq T_s) ? (\theta[1] + \theta[2] \times \tilde{T}_s + \theta[3] \times \tilde{q}[i]) : (\theta[4] + \theta[5] \times \tilde{T}_s + \theta[6] \times \tilde{q}[i])$ .................................................. 77

Figure 5-10 Pseudo-code for Adaptive SVT. ............................................................................. 79

Figure 5-11 The transformed code for AdaptiveSVT, where underlined parts are added by CheckDP. It contains three alignment templates for $\eta_1$ and $\eta_2$: $A_1 = \theta[0], A_2 = \Omega_{Top} ? (\theta[1] + \theta[2] \times \tilde{T}_s + \theta[3] \times \tilde{q}[i]) : (\theta[4] + \theta[5] \times \tilde{T}_s + \theta[6] \times \tilde{q}[i])$ and $A_3 = \Omega_{Middle} ? (\theta[1] + \theta[2] \times \tilde{T}_s + \theta[3] \times \tilde{q}[i]) : (\theta[4] + \theta[5] \times \tilde{T}_s + \theta[6] \times \tilde{q}[i])$, where $\Omega_1$ denotes the corresponding branch condition at Line 8 and 13. ........................................... 80

Figure 5-12 Report Noisy Max and its transformed code, where $S = q[i] + \eta > b \implies i = 0$?

$\theta[0] : \theta[1] \text{ and } A = q[i] + \eta > b \implies i = 0$? $\theta[2] + \theta[3] \times \tilde{q}[i] + \theta[4] \times \tilde{b_a} : \theta[5] + \theta[6] \times \tilde{q}[i] + \theta[7] \times \tilde{b_o}$ ............................. 82

Figure 5-13 BadSVT1 and its transformed code, where underlined parts are added by CheckDP.
The transformed code contains a alignment template for $\eta_1$: $A_1 = \theta[0]$ ................................. 83

Figure 5-14 BadSVT2 and its transformed code, where underlined parts are added by CheckDP.
The transformed code contains two alignment templates for $\eta_1$ and $\eta_2$: $A_1 = \theta[0]$ and $A_2 = (q[i] + \eta_2 \geq T_s) ? (\theta[1] + \theta[2] \times \tilde{T}_s + \theta[3] \times \tilde{q}[i]) : (\theta[4] + \theta[5] \times \tilde{T}_s + \theta[6] \times \tilde{q}[i])$. 84

Figure 5-15 BadSVT3 and its transformed code, where underlined parts are added by CheckDP.
The transformed code contains two alignment templates for $\eta_1$ and $\eta_2$: $A_1 = \theta[0]$ and $A_2 = (q[i] + \eta_2 \geq T_s) ? (\theta[1] + \theta[2] \times \tilde{T}_s + \theta[3] \times \tilde{q}[i]) : (\theta[4] + \theta[5] \times \tilde{T}_s + \theta[6] \times \tilde{q}[i])$. 85

Figure 5-16 PartialSum and its transformed code, where underlined parts are added by CheckDP.
The transformed code contains an alignment template $A = \theta[0] + \theta[1] \times \tilde{sum} + \theta[2] \times \tilde{q}[i]$ . 87

Figure 5-17 Pseudo-code for SmartSum. .................................................................................. 88

Figure 5-18 The transformed code for SmartSum. Underlined parts are added by CheckDP: $A_1 = \theta[0] + \theta[1] \times \tilde{sum} + \theta[2] \times \tilde{q}[i] + \theta[3] \times \tilde{next}$ and $A_2 = \theta[4] + \theta[5] \times \tilde{sum} + \theta[6] \times \tilde{q}[i] + \theta[7] \times \tilde{next}$ .......................................................... 89

Figure 6-1 Sparse Vector Technique. ......................................................................................... 95

Figure 6-2 An alternative way of making SVTBase $\epsilon$-private. ............................................. 97

Figure 6-3 Overview of DPGen. ................................................................................................. 98

Figure 6-4 DPGen: source and target language syntax. ............................................................... 100

Figure 6-5 Sketch of SVT-Base with extra noise. ................................................................. 104

Figure 6-6 Program transformation rules for expressions. ....................................................... 106
Program transformation rules for commands. $S$ represents the scale template instrumented in Phase 1. Distinguished variable $v_e$ and assertions are added to ensure differential privacy.

Transformed mechanism of SVT-Sketch by DPGen. The instrumented parts are underlined. For better readability, the proof and scale templates are represented by $A_i$ and $S_i$, respectively.

Overview of the search loop.

AdaptiveSVT-Base, while-private feature is used to enable the synthesis of adaptive mechanisms.

Synthesized AdaptiveSVT based on AdaptiveSVT-Base.

The transformation rule of while-private loop; the parts identical to a standard while-loop are omitted for readability.

SVT-Inverse and its transformed code.

Numerical Sparse Vector Technique and its transformed code.

GapSVT and its transformed code.

Modified part of SVT to form SVT-WhilePriv.

Report Noisy Max and its transformed code. $L_i$ stands for shadow execution selectors.

PartialSum and its transformed code.

SmartSum and its transformed code.
LIST OF TABLES

Table 4-1  Time spent on type checking and verification. ........................................... 43

Table 5-1  Detected counterexamples for the incorrect algorithms and comparisons with other sampling-based counterexample detectors. #t stands for true and #f stands for false. ...... 90

Table 5-2  Alignments found for the correct algorithms. ............................................. 90

Table 6-1  Synthesized random variables with corresponding alignment proof. Ω, stands for the branch condition in each mechanism. Unnecessary random variables that are removed in the optimization are omitted.......................................................... 117
Acknowledgments

This dissertation would not have been possible without support from numerous friends, colleagues, and family members. It is a great pleasure to take the time and opportunity to thank them in this dissertation.

First and foremost, I would like to express my sincere gratitude to my advisers Dr. Danfeng Zhang and Dr. Daniel Kifer. The tremendous amount of time each of them has generously devoted to helping me with research and presentation skills have shaped me to become a better researcher. It is fairly safe to say that their enthusiasm in this field, along with their extensive knowledge and innovative ideas is what drove me forward over the past years, to which I am forever grateful. I can only hope to be as inspiring and insightful a mentor and researcher as they are. Moreover, I wish to thank Dr. Gang Tan and Dr. Dinghao Wu for serving on my defense committee. Their helpful feedback and comments have vastly improved the quality of my research and this dissertation.

Additionally, I would like to thank my mentors Adam Welc and Lazaro Clapp during my internship at Uber. Their knowledge, patience, and especially sense of humor, have made my summer internship enjoyable and meaningful.

During my adventures as a Ph.D candidate at Penn State, I have the fortune to meet with many outstanding friends, to whom I owe my thanks. The very first research paper of mine was published with Zeyu Ding, who is not only an excellent researcher, but also a great friend to me. Moreover, I have enjoyed the dinners and board games with many other amazing friends including but absolutely not limited to Gen Pei, Zhanwei He, Mingzhao Yu, Yongzhe Huang, Tian Xie, and Weiling Cong. The conversations and collaborations with my labmates Yingtai Xiao, Guanhong Wang, Quan Zhou, Protty Protivash, John Durrell, Feiyang Yu, Jeffrey Ching, Robert Brotzman, and Peixuan Li have also made the time at Penn State inspiring and fun. I also wish to thank Yifan Chen, Jialin Li, Yixuan Luo, Yuhao Ding, Kesong Li, Yixuan Li, Bin Fu, Wen Gu, and Yun Han, for the support, for their decade-long friendships and for the good time we spent together through the screen, despite the time differences.

Specifically, I would like to thank my best and forever friend Yue Pan for all her love, companionship,
and unwavering support over the past years. The pains during my Ph.D journey would have been more
sufferable, and the joys less felt, if not for her.

Last, but most certainly not least, I would like to thank Mom and Dad for all their encouragement,
unconditional love and for things far too many to count throughout almost three decades. My cousin
deserves my wholehearted thanks as well, for his constant support and tips to guide me through
undergraduate and Ph.D journeys. Furthermore, I wish to dedicate this dissertation to my grandparents,
who I wish could still be here to see my successes. There are simply no words in this world to fully
express my feelings for them.

Funding Acknowledgment  The work presented in this dissertation is supported by NSF CNS-1228669,
CNS-1702760, CNS-1816282, CNS-1566411, CNS-1931686 and a gift from Facebook. The findings
and conclusions presented in this document are those of the authors and do not necessarily reflect the
views of the organizations listed previously.
Dedication

To my beloved grandparents.
Chapter 1

Introduction

Differential Privacy [1] has become a de facto standard for ensuring data privacy, it allows organizations to collect and share data with provable bounds on the information that is leaked about any individual. Major data sharing initiatives by organizations such as Google [2, 3], Apple [4], Microsoft [5], Uber [6] and the U.S. Census Bureau [7, 8, 9, 10] have adopted differential privacy for ensuring proper uses of data. Differential privacy enjoys several properties that make it outstanding: (1) it does not require assumptions about the auxiliary information that the adversaries have access to; (2) it is resilient to post-processing of the released data; (3) different differentially-private systems can be simply composed together without breaking the privacy guarantees, and most importantly, (4) leakage of individuals’ privacy is quantified by a tunable parameter, often referred to as privacy budget.

Crucial to any differentially private system is the correctness of privacy mechanisms, the underlying privacy primitives in larger privacy-preserving algorithms. Manually developing the necessary rigorous proofs that a mechanism correctly protects privacy is a subtle and error-prone process. For example, detailed explanations of significant errors in peer-reviewed papers and systems can be found in [11, 12, 13]. Such mistakes have led to researches on automated program analysis of differential privacy to aid data analysts in the mechanism design process. Specifically, tools focusing on the following major aspects have been developed: verification, counterexample detection and program synthesis.

Most existing works focus on formal verification for proving that mechanisms satisfy differential privacy [14, 15, 16, 17, 18, 19, 20, 21]. The initial line of work on formal verification for differential privacy (e.g., [22, 19, 17, 23, 20]) was concerned with expressiveness. They enable the verification of many complex differential privacy mechanisms, but often require heavy manual annotations from the programmers. A parallel line of work (e.g., [24, 25, 26, 27]) focuses more on usability – on developing platforms that keep track of the privacy cost of an algorithm while limiting the types of algorithms that users can produce. More recently, works show a trend toward usability and expressiveness – generating
a correctness proof of sophisticated algorithm while minimizing the annotation burden on programmers. Sometimes, combining those two requires substantial changes to program logics: a recent work [16] is able to verify a challenging example Report Noisy Max [28] automatically, but it involves a complex verification system using customized program logics and verifiers. The complexity of the system is then translated to considerable running time for verification, making it unsuitable for prototyping mechanisms.

Existing verifiers are only able to generate a binary answer indicating whether a mechanism satisfies differential privacy or not. However, when a mechanism has a subtle bug, auxiliary information is often needed for programmers to find the root cause. Hence, a complimentary line of work that target at disproving the mechanisms [29, 30, 31] has been proposed. Normally, a counterexample that triggers the violation of differential privacy is returned by the tools. All of existing works rely on sampling, i.e., running a mechanism hundreds of thousands of times and then using statistical methods to search for and confirm the violation. A fundamental limitation with this approach is that it is often time-consuming to run the mechanisms, and infrequent violations could fail to be detected due to the statistical nature.

More recently, researchers have also aimed at a more challenging task: developing tools for automatically synthesizing differentially-private mechanisms [32, 33]. Creating (or synthesizing) a differentially private mechanism is a process that often starts with a noise-free (non-private) mechanism. The designer then decides where to add noise, and how much of it to add. This can be a non-trivial process – if not done carefully, the mechanism might either violate differential privacy or have low utility:

- From the privacy perspective, one must carefully choose exactly where the noise must be injected and how much noise to use. Such decisions are notoriously tricky. For example, the Sparse Vector Technique (SVT) [28] is designed to return the identities of $N$ queries whose answers are likely to be larger than a public threshold $T$. Lyu et al. [12] catalog several peer-reviewed yet incorrect variants of SVT. At the source code level, these incorrect variants are very similar to the correct ones, but the tiny differences broke their privacy properties.

- From the utility perspective, there are many ways of converting a non-private program into a differentially private one – each such option could have wildly different utility properties. For instance, in the aforementioned SVT, injecting noise in one specific place (the threshold) allows the mechanism to use much less noise everywhere else (while processing more queries). Aside
from different valid choices in noise locations, the mechanism must also allocate its privacy budget among different code fragments. This results in a trade-off where fragments with a larger share of the privacy budget use smaller amounts of noise. Picking the optimal (in terms of utility) way of adding randomness and allocating the privacy budget is also a non-trivial task (this is especially true for SVT [12]).

Given these issues, this dissertation proposes comprehensive programming frameworks targeting at automated proving, disproving and synthesizing differential privacy mechanisms:

- For verification of sophisticated mechanisms, we propose a new proof technique named shadow execution, and embed it into a language called ShadowDP. ShadowDP uses shadow execution to generate proofs of differential privacy with very few programmer annotations and without relying on customized logics and verifiers. In addition to verifying a challenging example Report Noisy Max, we show that it can verify a new variant of Sparse Vector that reports the gap between some noisy query answers and the noisy threshold. Moreover, ShadowDP reduces the complexity of verification: for all of the algorithms we have evaluated, type checking and verification in total takes at most 3 seconds, while prior work takes minutes on the same algorithms.

- To further automate the verification process and integrate counterexample detections in the program analysis framework, we propose CheckDP, an automated and integrated approach for proving or disproving claims that a mechanism is differentially private. CheckDP can find counterexamples for mechanisms with subtle bugs for which prior counterexample generators have failed. Furthermore, it was able to automatically generate proofs for correct mechanisms for which no formal verification was reported before. CheckDP is built on static program analysis, allowing it to be more efficient and precise in catching infrequent events than sampling based counterexample generators (which run mechanisms hundreds of thousands of times to estimate their output distribution). Moreover, its sound approach also allows automatic verification of correct mechanisms. When evaluated on standard benchmarks and newer privacy mechanisms, CheckDP generates proofs (for correct mechanisms) and counterexamples (for incorrect mechanisms) within 70 seconds without any false positives or false negatives.

- Lastly, we present DPGen, a program synthesizer that takes in non-private code (without any noise)
and automatically synthesizes its differentially private version (with carefully calibrated noise). Under the hood, DPGen uses novel algorithms to automatically generate a sketch program with candidate locations for noise, and then optimize privacy proof and noise scales simultaneously on the sketch program. Moreover, DPGen can synthesize sophisticated mechanisms that adaptively process queries until a specified privacy budget is exhausted. When evaluated on standard benchmarks, DPGen is able to generate differentially private mechanisms that optimize simple utility functions within 120 seconds. Compared with existing synthesizers, DPGen is significantly more efficient and have the ability to synthesize adaptive mechanisms, a task that all prior works fail.

Outline The rest of this dissertation is organized as follows. Chapter 2 presents relevant backgrounds on differential privacy. Chapter 4 introduces a novel proving technique and a verifier ShadowDP that is able to efficiently verify a set of sophisticated mechanisms. Chapter 5 presents CheckDP, an automated and integrated tool to automatically provide proofs for correct mechanisms, as well as providing counterexamples when the proof fails. Chapter 6 discusses an automated synthesier that is able to generate private and useful mechanisms from user-provided base sketch programs. Finally, we share thoughts on potential directions for future work in Chapter 7 and summarize this dissertation in Chapter 8.
Chapter 2

Background

2.1 Differential Privacy

In this dissertation, we focus on pure differential privacy [1]. Intuitively, a data analysis $A$ satisfies differential privacy if and only if for any dataset $D$, adding, removing, or changing a record in $D$ has little impact on the analysis result. Therefore, a differentially private analysis reveals little about any data record being analyzed. Each analysis is built out of atomic components called differentially private mechanisms (privacy mechanisms for short). These components themselves satisfy differential privacy.¹

More formally, we say that two datasets $D, D' \in \mathcal{D}$ are adjacent, written $D \sim D'$, when they only differ on one record. To offer privacy, a differentially private mechanism, say $M : \mathcal{D} \rightarrow \mathcal{O}$, injects carefully calibrated random noise during its computation. We call the execution of $M$ on $D$, written $M(D)$, the original execution and its execution on (neighboring) dataset $D'$, written $M(D')$, the related execution. Intuitively, $M$ is $\epsilon$-differentially private for some constant $\epsilon$ if for any possible output $o \in \mathcal{O}$, the ratio between the probabilities of producing $o$ on $D$ and $D'$ is bounded by $e^\epsilon$:

**Definition 2.1** (Pure Differential Privacy [34]). Let $\epsilon \geq 0$. A probabilistic computation $M : \mathcal{D} \rightarrow \mathcal{O}$ is $\epsilon$-differentially private if $\forall D \sim D'$ (where $D, D' \in \mathcal{D}$) and $\forall o \in \mathcal{O}$, we have

$$
\mathbb{P}[M(D) = o] \leq e^\epsilon \mathbb{P}[M(D') = o]
$$

A differentially private analysis $A$ interacts with a dataset through one or multiple privacy mechanisms that take a list of queries and their exact answers as input, and produce a differentially private (noisy) aggregation of them. An important factor to determine the amount of noise needed for privacy is the sensitivity of queries, which intuitively quantifies the maximum difference of the query results on adjacent dataset

¹In general, the privacy parameter of the analysis is upper bounded by the sum of the individual privacy parameters of the mechanisms [24].
databases. We use a vector \((q_1, q_2, \ldots)\) to denote the exact query answers from running a sequence of queries on a dataset and say that each query answer \(q_i\) has a sensitivity of \(\Delta_i\) if its corresponding query has a global sensitivity of \(\Delta_i\):

**Definition 2.2** (Global Sensitivity [28]). The global sensitivity \(\Delta_f\) of a query \(f\) is \(\sup_{D, D'}|f(D) - f(D')|\).

Similar to dataset adjacency, we say two vectors of query answers are adjacent, written \((q_1, q_2, \ldots) \sim (q'_1, q'_2, \ldots)\), when \(\forall i. \ |q_i - q'_i| \leq \Delta_i\). Moreover, a privacy mechanism \(M\) satisfies \(\varepsilon\)-differential privacy if for all pairs of adjacent query answers \((q_1, q_2, \ldots) \sim (q'_1, q'_2, \ldots)\) and all outputs \(o \in O\), we have \(\mathbb{P}[M(q_1, q_2, \ldots, \text{params}) = o] \leq e^\varepsilon \mathbb{P}[M(q'_1, q'_2, \ldots, \text{params}) = o]\), where \(\text{params}\) represent data-independent parameters (e.g., the value of \(\varepsilon\)) to \(M\). As the goal of this dissertation is to synthesize privacy mechanisms, we assume that the sensitivity of inputs are either manually specified or computed by sensitivity analysis tools (e.g., [35, 36]).

One popular privacy mechanism is the Laplace Mechanism [1], which adds Laplace noise to query answers.

**Theorem 2.1** (Laplace Mechanism [1]). Let \(\text{Lap}\) \((n)\) be a sample from the Laplace distribution with mean 0 and scale \(n\). The Laplace Mechanism takes as input a query answer \(q\) with sensitivity \(\Delta_q\), and a privacy parameter \(\varepsilon\). It outputs \(q + \text{Lap} (\Delta_q/\varepsilon)\) and it satisfies \(\varepsilon\)-differential privacy.

In this dissertation, we will focus on analyzing mechanisms that use Laplace noise for ensuring differential privacy. Discussions about extending our analysis frameworks to support other noise distributions will be presented in Chapter 7.

## 2.2 Randomness Alignment

Proving differential privacy has been a challenging problem even for experts. Hence, mechanizing the correctness reasoning emerges to aid programmers ensure differential privacy. In this dissertation, we adopt the Randomness Alignment technique, a simple yet powerful proof technique that enables various verification tools and counterexample detectors [14, 15, 37].

Consider a privacy mechanism \(M\) and an arbitrary pair of adjacent vectors of query answers \((q_1, q_2, \ldots) \sim (q'_1, q'_2, \ldots)\). A randomness alignment is a function \(\phi : \mathbb{R}^\infty \rightarrow \mathbb{R}^\infty\) that maps random
samples used by an execution of $M$ on $(q_1, q_2, \ldots)$ to random samples used by the adjacent execution of $M$ on $(q'_1, q'_2, \ldots)$ such that both executions produce the same output.

Take Laplace Mechanism introduced in Section 2.1 for example. For an arbitrary pair of 1-sensitive queries $q_1$ and $q_2$, i.e., $-1 \leq q_1 - q_2 = c \leq 1$, an injective function $\phi$ that maps $\eta$ to $\eta + c$ is created. Obviously, $\phi$ is an alignment since $q_1 + \eta = q_2 + \phi(\eta)$ for any $q_1, q_2$. Then for an arbitrary set of outputs $E \subseteq \mathbb{R}$, we have:

$$P(M(q_1) \in E) = \sum_{\eta | q_1 + \eta \in E} p_{1/\epsilon}(\eta) \leq \sum_{\eta | q_2 + f(\eta) \in E} p_{1/\epsilon}(\eta) \leq e^\epsilon \sum_{\eta | q_2 + f(\eta) \in E} p_{1/\epsilon}(f(\eta))$$

$$= e^\epsilon \sum_{\eta | q_2 + \eta \in E} p_{1/\epsilon}(\eta) = e^\epsilon P(M(q_2) \in E)$$

The first inequality is by the definition of $f$: $q_1 + \eta \in E \implies q_2 + f(\eta) \in E$. The $e^\epsilon$ factor results from the fact that $p_{1/\epsilon}(\eta + c) / p_{1/\epsilon}(\eta) \leq e^{c} \leq e^\epsilon$, when the Laplace distribution has scale $1/\epsilon$. The second to last equality is by change of variable from $f(\eta)$ to $\eta$ in the summation, using the injectivity of $f$.

In general, it is useful to treat the privacy cost as a function of the alignment needed for each sampling instruction. For each sampling instruction $\eta = \text{Lap} (r)$, we define the distance of $\eta$, written as $\#\eta$, as $\phi(\eta) - \eta$\footnote{Here we abuse notation slightly by applying $\phi$ point-wise, letting $\phi(\eta)$ be the random sample $M$ should use in place of $\eta$ in the adjacent execution.}. Then, the privacy cost of aligning the sample $\eta$ is bounded by $|\#\eta| / r$. To find the overall privacy cost (i.e., the $\epsilon$ in pure differential privacy), we then take the summation of privacy cost of each sample generated in program execution, due to the Composition Theorem of pure differential privacy [28]. We note that since we can align each sample individually, randomness alignment is also applicable to sophisticated mechanisms where the composition theorem falls short [14, 15, 37].
Chapter 3

Related Work

In this chapter, we discuss work related to automated analysis of differential privacy. To better facilitate the discussion, this chapter is divided into three sections: verification, counterexample detection and synthesis for differential privacy.

3.1 Verification of Differential Privacy

Differential privacy has been a fruitful target for formal verification due to its compositional property. Based on different proving techniques, several lines of work have been successful in this field. In this section, we will focus on two state-of-the-art proving techniques: randomness alignment and probabilistic couplings, with other language-based techniques discussed at the end.

**Randomness Alignment Based Proofs** The first work to employ randomness alignment for verification of differential privacy is LightDP [14]. Equipped with a novel dependent type system to track the exact differences between two executions under adjacent databases, LightDP is powerful in expressing sophisticated alignments. However, although only a few annotations are needed for verification, it still requires manual labors. Moreover, subtle mechanisms such as Report Noisy Max is still beyond reach of LightDP.

**Coupling Based Proofs** Besides alignment-based proofs, probabilistic couplings and liftings [21, 20, 16] have also been used in language-based verification of differential privacy. Coupling proofs are known to be more general than alignment-based proofs, while alignment-based proofs are more light-weight. Most notably, Albarghouthi and Hsu [16] proposed the first automated tool capable of generating coupling proofs for complex mechanisms. Notably, it is the first tool to be able to automatically verify
proofs for sophisticated algorithms such as Report-Noisy-Max. However, it builds itself upon customized program logics which generates first-order Horn clauses and probabilistic constraints generated, making verification significantly more time-consuming. Moreover, the intermediate transformed programs generated by it are unusable by other program analyses. In fact, existing works built on coupling proofs [17, 18, 19, 20, 21] all depend on customized relational logics and share the same limitations.

**Other Language-Based Proofs** With verified privacy mechanisms as building blocks, such as SVT and Report Noisy Max, we still need to verify that the larger program built on top of them is differentially private. An early line of work [17, 19, 23, 35, 36] uses (variations of) relational Hoare logic and linear indexed types to derive differential privacy guarantees. For example, Fuzz [35] and its successor DFuzz [36] combine linear indexed types and lightweight dependent types to allow rich sensitivity analysis and then use the composition theorem to prove overall system privacy.

Recent work such as Personalized Differential Privacy (PDP) [38] allows each individual to set its own different privacy level and PDP will satisfy difference privacy regarding the level she sets. PINQ [24] tracks privacy consumption dynamically on databases and terminate when the privacy budget is exhausted. However, along with other work such as computing bisimulations families for probabilistic automata [39, 40], they fail to provide a tight bound on the privacy cost of sophisticated algorithms.

### 3.2 Counterexample Detection For Differential Privacy

Ding et al. [29] and Bichsel et al. [30] proposed counterexample generators that rely on sampling – running an algorithm hundreds of thousands of times to estimate the output distribution of mechanisms (this information is then used to find counterexamples). Specifically, StatDP [29] uses statistical hypothesis testing to demonstrate high probability of privacy violations and DP-Finder [30] uses symbolic differentiation and gradient descent to search for counterexamples.

More recently, DP-Sniper [41] trains a classifier – a parametric family of posterior probability distributions to predict if an observed output is likely generated from one of two possible inputs, and use this classifier to select a set of outputs that can best distinguish these two inputs. All these methods rely on sampling – running an algorithm hundreds of thousands of times to estimate the output distribution.
of mechanisms and generate counterexample candidates/training data. The strength of these methods is that they do not rely on external solvers, and more importantly, they are not tied to (the limitation of) any particular proof technique (e.g., randomness alignment and coupling). However, sampling also make the counterexample detectors imprecise and more likely to fail in some cases.

Recent work [42, 43] targets both proving and disproving differential privacy. Barthe et al. [42] identify a non-trivial class of programs where checking (pure and approximate) differential privacy is decidable. However, these programs only allow a bounded number of samples from the Laplace distribution, and their inputs and outputs are from a finite domain. Farina [43] builds a relational symbolic execution framework, which when combined with probabilistic couplings, is able to prove differential privacy for SVT or generate failing traces for its two incorrect variants.

### 3.3 Synthesizing Differentially Private Algorithms

More recently, researchers have also developed tools targeting at automated synthesis for differential privacy. An early work [32] relies on user supplied examples and uses a sensitivity-directed program synthesis technique based on DFuzz [36]. However, it can only synthesize simple mechanisms where the privacy analysis follows directly from the composition theorem.

KOLAHAL, recently proposed by Roy et al. [33] is the first synthesizer for complex mechanisms. KOLAHAL takes, as inputs, a sketch mechanism with noise expressions in known locations as holes and a finite grammar for noise expressions, and leverages counterexamples generated by StatDP [29] and continuous optimization approximation to guide the optimization of noise functions. It supports multiple noise distributions (Laplace, Exponential) and is the first tool capable of synthesizing complex differential privacy mechanisms including Report-Noisy-Max, SVT and SmartSum. However, it (1) requires expert knowledge to supply known locations for noise, (2) is inefficient due to large search space for scales of noise, and (3) fails to synthesize sophisticated adaptive mechanisms such as AdaptiveSVT [44].
Chapter 4

Proving Differential Privacy with Shadow Execution

4.1 Introduction

A recent line of work for verification of differential privacy (most notably LightDP [14] and Synthesizing Coupling Proofs [16]) has sought to combine expressiveness and usability by providing verification tools that infer most (if not all) of the proof of privacy. The benchmark algorithms for this task were Sparse Vector [12, 28] and Report Noisy Max [28]. LightDP [14] was the first system that could verify Sparse Vector with very few annotations, but it could not verify tight privacy bounds on Report Noisy Max [28]. It is believed that proofs using randomness alignment, the proof technique that underpins LightDP, are often simpler, while approximate coupling, the proof technique that underpins [22, 19, 17, 23, 20], seems to be more expressive [16]. Subsequently, Albarghouthi and Hsu [16] produced the first fully automated system that verifies both Sparse Vector and Report Noisy Max. Although this new system takes inspiration from randomness alignment to simplify approximate coupling proofs, its verification system still involves challenging features such as first-order Horn clauses and probabilistic constraints; it takes minutes to verify simple algorithms. The complex verification system also prevents it from reusing off-the-shelf verification tools.

In this chapter, we present ShadowDP, a language for verifying differentially private algorithms. It is based on a new proof technique called “shadow execution”, which enables language-based proofs based on standard program logics. Built on randomness alignment, it transforms a probabilistic program into a program in which the privacy cost is explicit; so that the target program can be readily verified by off-the-shelf verification tools. However, unlike LightDP, it can verify more challenging algorithms such as Report Noisy Max and a novel variant of Sparse Vector called Difference Sparse Vector. We show that with minimum annotations, challenging algorithms that took minutes to verify by [16] (excluding proof synthesis time) now can be verified within 3 seconds with an off-the-shelf model checker.
One extra benefit of this approach based on randomness alignment is that the transformed program can also be analyzed by standard symbolic executors. This appears to be an important property in light of recent work on detecting counterexamples for buggy programs [29, 30, 31, 45]. Producing a transformed program that can be used for verification of correct programs and bug-finding for incorrect programs is one aspect that is of independent interest.

In this chapter, our key contributions are:

1. Shadow execution, a new proof technique for differential privacy (Section 4.3).

2. ShadowDP, a new imperative language (Section 4.4) with a flow-sensitive type system (Section 4.5) for verifying sophisticated privacy-preserving algorithms.

3. A formal proof that the verification of the transformed program by ShadowDP implies that the source code is $\epsilon$-differentially private (Section 4.7).

4. Case studies on sophisticated algorithms showing that verifying privacy-preserving algorithms using ShadowDP requires little programmer annotation burden and verification time (Section 4.8).

5. Verification of a variant of Sparse Vector Technique that releases the difference between noisy query answers and a noisy threshold at the same privacy level as the original algorithm [12, 28].

### 4.2 Motivating Example

To illustrate the challenges in proving differential privacy, we consider the Report Noisy Max algorithm [34], whose source code is shown on the top of Figure 4-1. It can be used as a building block in algorithms that iteratively generate differentially private synthetic data by finding (with high probability) the identity of the query for which the synthetic data currently has the largest error [46].

The algorithm takes a list $q$ of query answers, each of which differs by at most 1 if the underlying database is replaced with an adjacent one. The goal is to return the index of the largest query answer (as accurately as possible subject to privacy constraints).

To achieve differential privacy, the algorithm adds appropriate Laplace noise to each query. Here, $\text{Lap}(2/\epsilon)$ draws one sample from the Laplace distribution with mean zero and a scale factor $(2/\epsilon)$. For
function NoisyMax (\(\epsilon\), size: \(\text{num}_{\text{0,}0}\); q: \(\text{list num}_{\text{0,}+}\))
returns max: \(\text{num}_{\text{0,}+}\)

\[\forall i \geq 0. -1 \leq \tilde{q}^o[i] \leq 1 \land \tilde{q}^+ [i] = \tilde{q}^o [i]\]

1. \(i := 0\); \(\text{bq} := 0\); \(\text{max} := 0\);
2. \(\text{while} (i < \text{size})\)
3. \(\eta := \text{Lap} \left(\frac{2}{\epsilon}\right), \Omega ?^\top_0, \Omega ? \left(2 : 0\right)\);
4. \(\text{if} (q[i] + \eta > \text{bq} \lor i = 0)\)
5. \(\text{max} := i;\)
6. \(\text{bq} := q[i] + \eta;\)
7. \(i := i + 1;\)

The transformed program (slightly simplified for brevity) with underlined parts added by the type system:

1. \(v_\epsilon := 0;\)
2. \(i := 0; \text{bq} := 0; \text{max} := 0;\)
3. \(b_\eta^0 := 0; b_\eta^+ := 0;\)
4. \(\text{while} (i < \text{size})\)
5. \(\text{assert} (i < \text{size});\)
6. \(\text{havoc } \eta; v_\epsilon := \Omega ?(\theta + \epsilon):(v_\epsilon + 0);\)
7. \(\text{if} (q[i] + \eta > \text{bq} \lor i = 0)\)
8. \(\text{assert} (q[i] + \tilde{q}^+ [i] + \eta + 2 > \text{bq} + \tilde{b}_\eta^+ \lor i = 0);\)
9. \(\text{max} := i;\)
10. \(\tilde{\text{bq}}^+ := \text{bq} + \tilde{\text{bq}}^+ - (q[i] + \eta);\)
11. \(\text{bq} := q[i] + \eta;\)
12. \(\tilde{\text{bq}}^0 := \tilde{q}^o[i] + 2;\)
13. \(\text{else}\)
14. \(\text{assert} (-\left(q[i] + \tilde{q}^o [i] + \eta + 0 > \text{bq} + \tilde{\text{bq}}^o \lor i = 0)\right);\)
15. \(\text{// shadow execution}\)
16. \(\text{if} (q[i] + \tilde{q}^+ [i] + \eta > \text{bq} + \tilde{\text{bq}}^+ \lor i = 0)\)
17. \(\tilde{\text{bq}}^+ := q[i] + \tilde{q}^+ [i] + \eta - \text{bq};\)
18. \(i := i + 1;\)

Figure 4-1: Verifying Report Noisy Max with ShadowDP. Here, \(q\) is a list of query answers from a database, and max is the query index of the maximum query with Laplace noise generated at line 3. To verify the algorithm on the top, a programmer provides function specification as well as annotation for sampling command (annotations are shown in gray, where \(\Omega\) represents the branch condition). ShadowDP checks the source code and generates the transformed code (at the bottom), which can be verified with off-the-shelf verifiers.
privacy, the algorithm uses the noisy query answer \((q[i] + \eta)\) rather than the true query answer \((q[i])\) to compute and return the index of the maximum (noisy) query answer. Note that the return value is listed right below the function signature in the source code.

**Informal Proof Using Randomness Alignment** Proofs of correctness of Report Noisy Max can be found in [28]. We will start with an informal correctness argument, based on the *randomness alignment* technique (Section 2.2), to illustrate subtleties involved in the proof.

Consider the following two databases \(D_1, D_2\) that differ on one record, and their corresponding query answers:

<table>
<thead>
<tr>
<th></th>
<th>(q[0])</th>
<th>(q[1])</th>
<th>(q[2])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D_1)</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(D_2)</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Suppose in one execution on \(D_1\), the noise added to \(q[0], q[1], q[2]\) is \(\alpha_0^{(1)} = 1, \alpha_1^{(1)} = 2, \alpha_2^{(1)} = 1\), respectively. In this case, the noisy query answers are \(q[0] + \alpha_0^{(1)} = 2, q[1] + \alpha_1^{(1)} = 4, q[2] + \alpha_2^{(1)} = 3\) and so the algorithm returns 1, which is the index of the maximum noise query answer of 4.

**Aligning Randomness** In order to formally prove the algorithm using randomness alignment, we need to create an injective function of random bits in \(D_1\) to random bits in \(D_2\) to make the output the same. Recall that \(\alpha_0^{(1)}, \alpha_1^{(1)}, \alpha_2^{(1)}\) are the noise added to \(D_1\), now let \(\alpha_0^{(2)}, \alpha_1^{(2)}, \alpha_2^{(2)}\) be the noise added to the queries \(q[0], q[1], q[2]\) in \(D_2\), respectively. Consider the following injective function: for any query except for \(q[1]\), use the same noise as on \(D_1\); add 2 to the noise used for \(q[1]\) (i.e., \(\alpha_1^{(2)} = \alpha_1^{(1)} + 2\)).

In our running example, execution on \(D_2\) with this alignment function would result in noisy query answers \(q[0] + \alpha_0^{(2)} = 3, q[1] + \alpha_1^{(2)} = 5, q[2] + \alpha_2^{(2)} = 3\). Hence, the output once again is 1, the index of query answer 5.

In fact, we can prove that under this alignment, *every execution on \(D_1\) where 1 is returned* would result in an execution on \(D_2\) that produces the same answer due to two facts:

1. On \(D_1\), \(q[1] + \alpha_1^{(1)}\) has the maximum value;

2. On \(D_2\), \(q[1] + \alpha_1^{(2)}\) is greater than \(q[1] + \alpha_1^{(1)} + 1\) on \(D_1\) due to \(\alpha_1^{(2)} = \alpha_1^{(1)} + 2\) and the adjacency assumption (i.e., \(q[i]\) on \(D_2\) can only differ from \(q[i]\) on \(D_1\) by at most 1).
Hence, \( q[1] + \alpha_1^{(2)} \) on \( D_2 \) is greater than \( q[i] + \alpha_i^{(1)} + 1 \) on \( D_1 \) for any \( i \). By the adjacency assumption, that is the same as \( q[1] + \alpha_1^{(2)} \) is greater than any \( q[i] + \alpha_i^{(2)} \) on \( D_2 \). Based on randomness alignment, we can prove that the Report Noisy Max algorithm is \( \epsilon \)-private for this particular pair of databases.

**Challenges**  Unfortunately, the alignment function above only applies to executions on \( D_1 \) where index 1 is returned. If there is one more query \( q[3] = 4 \) and the execution gets noise \( \alpha_3^{(1)} = 1 \) for that query, the execution on \( D_1 \) will return index 3 instead of 1. To align randomness on \( D_2 \), we need to construct a different alignment function (following the construction above) that adds noise in the following way: for any query except for \( q[3] \), use the same noise as on \( D_1 \); add 2 to the noise used for \( q[3] \) (i.e., \( \alpha_3^{(2)} = \alpha_3^{(1)} + 2 \)). In other words, to carry out the proof, the alignment for each query depends on the queries and noise yet to happen in the future.

One approach of tackling this challenge, followed by existing language-based proofs of Report Noisy Max [20, 16], is to use the pointwise lifting argument: informally, if we can show that for any value \( i \), execution on \( D_1 \) returns value \( i \) implies execution on \( D_2 \) returns value \( i \) (with a privacy cost bounded by \( \epsilon \)), then a program is \( \epsilon \)-differential private. However, this argument does not apply to the randomness alignment technique. Moreover, doing so requires a customized program logic for proving differential privacy.

### 4.3 Approach Overview

In this section, we discuss the overview of a new proof technique “shadow execution”, which enables language-based proofs based on standard program logics. The key insight is to track a shadow execution on \( D_2 \) where the same noise is always used as on \( D_1 \). For our motivating example, we illustrate the shadow execution in Figure 4-2, with random noise \( \alpha_0^{(\dagger)} \), \( \alpha_1^{(\dagger)} \) and so on. Note that the shadow execution uses \( \alpha_i^{(\dagger)} = \alpha_i^{(1)} \) for all \( i \).

With shadow execution, we can construct a randomness alignment for each query \( i \) as follows:

**Case 1:** Whenever \( q[i] + \alpha_i^{(1)} \) is the maximum value so far on \( D_1 \) (i.e., \( max \) is updated), we use the alignments on shadow execution for all previous queries but add a noise \( \alpha_i^{(1)} + 2 \) for \( q[i] \) on \( D_2 \).
Case 2: Whenever $q[i] + a_i^{(1)}$ is smaller than or equal to any previous noise query answer (i.e., $\max$ is not updated), we keep the previous alignments for previous queries and use noise $a_i^{(1)}$ for $q[i]$ on $D_2$.

After seeing $q[1]$ on $D_1$ (Case 1), the construction uses noise in the shadow execution for previous query answers, and uses $a_1^{(1)} + 2 = 4$ as the noise for $q[1]$ on $D_2$. After seeing $q[2]$ on $D_1$ (Case 2), the construction reuses alignments constructed previously, and use $a_2^{(1)} = 1$ as the noise for $q[2]$. When $q[3]$ comes, the previous alignment is abandoned; the shadow execution is used for $q[0]$ to $q[2]$. It is easy to check that this construction is correct for any subset of query answers seen so far, since the resulting alignment is exactly the same as the informal proof above, when the index of maximum value is known.

**Randomness Alignment with Shadow Execution**  To incorporate the informal argument above to a programming language, we propose ShadowDP. We illustrate the key components of ShadowDP in this section, as shown in Figure 4-1, and details of all components in the rest of this chapter.

Similar to LightDP [14], ShadowDP embeds randomness alignments into types. In particular, each numerical variable has a type in the form of $\text{num}_{(\pi^0,\pi^1)}$, where $\pi^0$ and $\pi^1$ represent the “difference” of its value in the aligned and shadow execution respectively. In Figure 4-1, non-private variables, such as $\epsilon$, size, are annotated with distance 0. For private variables, the difference could be a constant or an expression. For example, the type of $q$ along with the precondition specifies the adjacency relation: each query answer’s difference is specified by $\ast$, which is desugared to a special variable $\bar{q}^\ast[i]$. The precondition in Figure 4-1 specifies that the difference of each query answer is bounded by 1 (i.e., query answers have sensitivity of 1).
ShadowDP reasons about the aligned and shadow executions in isolation, with the exception of sampling commands. A sampling command (e.g., line 3 in Figure 4-1) constructs the aligned execution by either using values from the aligned execution so far (symbol $\circ$), or switching to values from the shadow execution (symbol $\dagger$). The construction may depend on program state: in Figure 4-1, we switch to shadow values iff $q[i] + \eta$ is the max on $D_1$. A sampling command also specifies the alignment for the generated random noise.

With function specification and annotations for sampling commands, the type system of ShadowDP automatically checks the source code. If successful, it generates a non-probabilistic program (as shown at the bottom of Figure 4-1) with a distinguished variable $v_\epsilon$. The soundness of the type system ensures the following property: if $v_\epsilon$ is bounded by some constant $\epsilon$ in the transformed program, then the original program being verified is $\epsilon$-private.

**Benefits**  Compared with previous language-based proofs of Report Noisy Max [20, 16] (both are based on the pointwise lifting argument), ShadowDP enjoys a unique benefit: the transformed code can be verified based on *standard* program semantics. Hence, the transformed (non-probabilistic) program can be further analyzed by existing program verifiers and other tools. For example, the transformed program in Figure 4-1 is verified with an off-the-shelf tool CPAChecker[47] *without any extra annotation* within seconds.

### 4.4 ShadowDP: Syntax and Semantics

In this section, we present the syntax and semantics of ShadowDP, a simple imperative language for designing and verifying differentially private algorithms.

#### 4.4.1 Syntax

The language syntax is given in Figure 4-3. Most parts of ShadowDP is standard; we introduce a few interesting features.
Non-probabilistic Variables and Expressions  ShadowDP supports real numbers, booleans as well as standard operations on them. We use \( n \) and \( b \) to represent numeric and boolean expressions respectively. A ternary numeric expression \( b ? n_1 : n_2 \) evaluates to \( n_1 \) when the comparison evaluates to true, and \( n_2 \) otherwise. Moreover, to model multiple queries to a database and produce multiple outputs during that process, ShadowDP supports lists: \( e_1 : e_2 \) appends the element \( e_1 \) to a list \( e_2 \); \( e_1[e_2] \) gets the \( e_2 \)-th element in list \( e_1 \), assuming \( e_2 \) is bound by the length of \( e_1 \).

Random Variables and Expressions  To model probabilistic computation, which is essential in differentially private algorithms, ShadowDP uses random variable \( \eta \in RVars \) to store a sample drawn from a distribution. Random variables are similar to normal variables (\( x \in NVars \)) except that they are the only ones who can get random values from random expressions, via a sampling command \( \eta := g \).

We follow the modular design of LightDP [14], where randomness expressions can be added easily. In this chapter, we only consider the most interesting random expression, \( \text{Lap} \ r \). Semantically, \( \eta := \text{Lap} \ r \) draws one sample from the Laplace distribution, with mean zero and a scale factor \( r \), and assigns it to \( \eta \).

For verification purpose, a sampling command also requires a few annotations, which we explain shortly.
Types  Types in ShadowDP have the form of $B_{\langle d', d \rangle}$, where $B$ is the base type, and $d', d$ represent the alignments for the execution on adjacent database and shadow execution respectively. Base type is standard: it includes num (numeric type), bool (Boolean), or a list of elements with type $\tau$ (list $\tau$).

Distance $d$ is the key for randomness alignment proof. Intuitively, it relates two program executions so that the likelihood of seeing each is bounded by some constant. Since only numerical values have numeric distances, other data types (including bool, list $\tau$ and $\tau_1 \rightarrow \tau_2$) are always associated with $(0, 0)$, hence omitted in the syntax. Note that this does not rule out numeric distances in nested types.

For example, $(\text{list num}(1, 1))$ stores numbers that differ by exactly one in both aligned and shadow executions.

Distance $d$ can either be a numeric expression ($n$) in the language or $\ast$. A variable $x$ with type $\text{num}(\ast, \ast)$ is desugared as $x : \Sigma ((\hat{x} : \text{num}(0, 0), \hat{y} : \text{num}(0, 0)))$ $\text{num} (\hat{x}, \hat{y})$, where $\hat{x}$, $\hat{y}$ are distinguished variables invisible in the source code; hiding those variables in a $\Sigma$-type simplifies the type system (Section 4.5).

The star type is useful for two reasons. First, it specifies the sensitivity of query answers in a precise way. Consider the parameter $q$ in Figure 4-1 with type list $\text{num}(\ast, \ast)$, along with the precondition $\forall i \geq 0. -1 \leq q[i] \leq 1$. This notation makes the assumption of the Report Noisy Max algorithm explicit: each query answer differs by at most 1. Second, star type serves as the last resort when the distance of a variable cannot be tracked precisely by a static type system. For example, whenever ShadowDP merges two different distances (e.g., 3 and 4) of $x$ from two branches, the result distance is $\ast$; the type system instruments the source code to maintain the correct values of $\hat{x}$, $\hat{y}$ (Section 4.5).

Sampling with Selectors  Each sampling instruction is attached with a few annotations for proving differential privacy, in the form of $(\eta := \text{Lap} r, S, v_{\eta})$. Note that just like types, the annotations $S$, $v_{\eta}$ have no effects on the program semantics; they only show up in verification. Intuitively, a selector $S$ picks a version ($k \in \{\circ, \dagger\}$) for all program variables (including the previously sampled variables) at the sampling instruction, as well as constructs randomness alignment for $\eta$, specified by $v_{\eta}$ (note that the distance cannot be $\ast$ by syntactical restriction here). By definition, both $S$ and $v_{\eta}$ may depend on the program state when the sampling happens.

Return to the running example in Figure 4-1. As illustrated in Figure 4-2, the selective alignment is to
\[
\begin{align*}
\text{[skip]}_m &= \text{unit } m \\
[x := e]_m &= \text{unit } (m\{e/m\}) \\
[\eta := g, S, \forall \eta]_m &= \text{bind } [g] (\lambda v. \text{unit } m\{v/\eta\}) \\
[c_1; c_2]_m &= \text{bind } ([c_1]_m) [c_2] \\
\text{if } e \text{ then } c_1 \text{ else } c_2]_m &= \begin{cases} 
[c_1]_m & \text{if } [e]_m = \text{true} \\
[c_2]_m & \text{if } [e]_m = \text{false} 
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{while } e \text{ do } c]_m &= (\text{fix } (\lambda f. \lambda m. \text{if } [e]_m = \text{true} \text{ then } (\text{bind } [c]_m f) \text{ else } (\text{unit } m))) m \\
[c; \text{return } e]_m &= \text{bind } ([c]_m) (\lambda m'. \text{unit } [e]_{m'})
\end{align*}
\]

Figure 4-4: ShadowDP: language semantics.

- use shadow variables and align the new sample by 2 whenever a new max is encountered,
- use aligned variables and the same sample otherwise.

Hence, the sampling command in Figure 4-1 is annotated as \( (\eta := \text{Lap } (2/e), \Omega ? \dagger : \circ, \Omega ? 2 : 0) \), where \( \Omega \) is \( q[i]+\eta > bq \lor i=\emptyset \), the condition when a new max is found.

4.4.2 Semantics

As standard, the denotational semantics of the probabilistic language is defined as a mapping from initial memory to a distribution on (possible) final outputs. Formally, let \( \mathcal{M} \) be a set of memory states where each \( m \in \mathcal{M} \) maps all (normal and random) variables \((N\text{Vars} \cup R\text{Vars})\) to their values.

The semantics of an expression \( e \) of base type \( \mathcal{B} \) is interpreted as a function \([e] : \mathcal{M} \to [\mathcal{B}]\), where \([\mathcal{B}]\) represents the set of values belonging to the base type \( \mathcal{B} \). We omit expression semantics since it is standard. A random expression \( g \) is interpreted as a distribution on real values. Hence, \( [g] : \text{Dist}([\text{num}]) \). Moreover, a command \( c \) is interpreted as a function \([c] : \mathcal{M} \to \text{Dist}(\mathcal{M})\). For brevity, we write \([e]_m\) and \([c]_m\) instead of \([e](m)\) and \([c](m)\) hereafter. Finally, all programs have the form \((c; \text{return } e)\) where \( c \) contains no return statement. A program is interpreted as a function \( m \to \text{Dist}([\mathcal{B}]\) where \( \mathcal{B} \) is the return type (of \( e \)).
The semantics of commands is shown in Figure 4-4, which directly follows a standard semantics in [48]. Given a distribution \( \mu \in \text{Dist}(A) \), its support is defined as \( \text{support}(\mu) \triangleq \{a \mid \mu(a) > 0\} \). We use \( \mathbb{1}_a \) to represent the degenerate distribution \( \mu \) that \( \mu(a) = 1 \) and \( \mu(a') = 0 \) if \( a' \neq a \). Moreover, we define monadic functions \( \text{unit} \) and \( \text{bind} \) functions to formalize the semantics for commands:

\[
\text{unit} : A \to \text{Dist}(A) \triangleq \lambda a. \mathbb{1}_a
\]

\[
\text{bind} : \text{Dist}(A) \to (A \to \text{Dist}(B)) \to \text{Dist}(B) \triangleq \lambda \mu. \lambda f. \sum_{a \in A} (f a b) \times \mu(a)
\]

That is, \( \text{unit} \) takes an element in \( A \) and returns the Dirac distribution where all mass is assigned to \( a \); \( \text{bind} \) takes \( \mu \), a distribution on \( A \), and \( f \), a mapping from \( A \) to distributions on \( B \) (e.g., a conditional distribution of \( B \) given \( A \)), and returns the corresponding marginal distribution on \( B \). This monadic view avoids cluttered definitions and proofs when probabilistic programs are involved.

### 4.5 ShadowDP: Type System

ShadowDP is equipped with a flow-sensitive type system. If successful, it generates a transformed program with needed assertions to make the original program differentially private. The transformed program is simple enough to be verified by off-the-shelf program verifiers.

#### 4.5.1 Notations

We denote by \( \Gamma \) the typing environment which tracks the type of each variable in a flow-sensitive way (i.e., the type of each variable at each program point is traced separately). Typing rules for expressions and commands are formalized in Figure 4-5 and Figure 4-6, respectively. All typing rules share a common global invariant \( \Psi \), such as the sensitivity assumption annotated in the source code (e.g., the precondition in Figure 4-1). We also write \( \Gamma(x) = (d^\text{\_d}, d^\text{\_d}) \) for \( \exists \mathcal{B}. \Gamma(x) = \mathcal{B}(d^\text{\_d}, d^\text{\_d}) \) when the base type \( \mathcal{B} \) is irrelevant.
4.5.2 Expressions

Expression rules have the form of $\Gamma \vdash e : \tau$, which means that expression $e$ has type $\tau$ under the environment $\Gamma$. Most rules are straightforward: they compute the distance for aligned and shadow executions separately. Rule (T-OTimes) makes a conservative approach for nonlinear computations, following LightDP [14]. Rule (T-VAR) desugars star types when needed. The most interesting rule is (T-ODot), which generates the following constraint:

$$\Psi \Rightarrow (e_1 \odot e_2 \iff (e_1 + \pi_1) \odot (e_2 + \pi_3))$$

This constraint states that the boolean value of $e_1 \odot e_2$ is identical in both aligned and shadow executions. If the constraint is discharged by an external solver (our type system uses Z3 [49]), we are assured that $e_1 \odot e_2$ has distances $(0,0)$. 

Figure 4-5: Typing rules for expressions. $\Psi$ is an invariant that holds throughout program execution.
Typing Rules for Commands

\[
\frac{\Gamma \vdash e : \mathcal{B}(\sigma, \sigma') \quad \langle \Gamma', e \rangle = \begin{cases} \langle \Gamma[x \mapsto \mathcal{B}(\sigma, \eta^\dagger)], \text{skip}\rangle, & \text{if } pc = \bot \\
\langle \Gamma[x \mapsto \mathcal{B}(\sigma, \eta^\dagger)], x^\dagger := x + \eta^\dagger - e \rangle, & \text{else}\end{cases} \quad \Gamma' \quad (\text{T-Skip})}{\Gamma \vdash e : \mathcal{B}(\sigma, \sigma') \quad \langle \Gamma', e \rangle \quad \text{(T-ASGN)}}
\]

\[
\frac{\Gamma \vdash \{c_1 \rightarrow c'_1\} \Gamma_1 \quad pc + \Gamma \{c_2 \rightarrow c'_2\} \Gamma_2}{\Gamma \vdash \{c_1; c_2 \rightarrow c'_1; c'_2\} \Gamma'_2 \quad \Gamma' \quad \text{(T-SEQ)}}
\]

\[
\frac{pc + \Gamma \{\text{return } e \rightarrow \text{return } e\}}{pc + \Gamma \{\text{return } e \rightarrow \text{return } e\} \quad \text{(T-RETURN)}}
\]

\[
\frac{pc + \Gamma \{\text{if } e \text{ then } c_1 \text{ else } c_2 \rightarrow \{\text{if } e \text{ then } (\text{assert } (\langle e, \Gamma \rangle^\dagger); c'_1; c'_2) \text{ else } (\text{assert } (\langle e, \Gamma \rangle^\dagger); c'_1; c'_2)); c'\}\} \Gamma_1 \cup \Gamma_2}{pc + \Gamma \{\text{if } e \text{ then } c_1 \text{ else } c_2 \rightarrow \{\text{if } e \text{ then } (\text{assert } (\langle e, \Gamma \rangle^\dagger); c'_1; c'_2) \text{ else } (\text{assert } (\langle e, \Gamma \rangle^\dagger); c'_1; c'_2)); c'\}\} \Gamma_1 \cup \Gamma_2 \quad \text{(T-IF)}}
\]

\[
\frac{pc + \Gamma \{\text{while } e \text{ do } c \rightarrow c_3; \text{while } e \text{ do } (\text{assert } (\langle e, \Gamma \rangle^\dagger); c'_1; c'_2)); c'\}\} \Gamma_1 \cup \Gamma_2}{pc + \Gamma \{\text{while } e \text{ do } c \rightarrow c_3; \text{while } e \text{ do } (\text{assert } (\langle e, \Gamma \rangle^\dagger); c'_1; c'_2)); c'\}\} \Gamma_1 \cup \Gamma_2 \quad \text{(T-WHILE)}}
\]

Typing Rules for Random Assignments

\[
\frac{pc = \bot}{\Gamma' = \lambda x. \langle S((\eta^\dagger, \eta^\dagger)), \eta^\dagger \rangle \text{ where } \Gamma \vdash x : \mathcal{B}(\sigma, \eta^\dagger)} \quad \text{(T-LAPLACE)}}
\]

Instrumentation Rule

\[
\frac{pc = \bot}{\Gamma' = \lambda x. \langle S((\eta^\dagger, \eta^\dagger)), \eta^\dagger \rangle \text{ where } \Gamma \vdash x : \mathcal{B}(\sigma, \eta^\dagger)} \quad \text{(T-LAPLACE)}}
\]

Select Function

\[
\circ((e_1, e_2)) = e_1 \quad \uparrow((e_1, e_2)) = e_2 \quad (e ? S_1 : S_2)((e_1, e_2)) = e ? S_1((e_1, e_2)) : S_2((e_1, e_2))
\]

PC Update Function

\[
\text{updatePC}(pc, \Gamma, e) = \begin{cases} \bot, & \text{if } pc = \bot \land \Psi \Rightarrow (e \leftrightarrow (\langle e, \Gamma \rangle^\dagger)) \\
\top, & \text{else}\end{cases}
\]

Figure 4-6: Typing rules for commands and auxiliary rules for commands. \(\Psi\) is an invariant that holds throughout program execution. In most rules, shadow distances are handled in the same way as aligned distances, with exceptions highlighted in gray boxes.

4.5.3 Commands

The flow-sensitive type system tracks and checks the distances of aligned and shadow executions at each program point. Typing rules for commands have the form of
meaning that starting from the previous typing environment $\Gamma$, the new typing environment is $\Gamma'$ after $c$. We will discuss the other components $pc$ and $c'$ shortly.

### 4.5.4 Aligned Variables

The type system infers and checks the distances of both aligned and shadow variables. Since most rules treat them in the same way, we first explain the rules with respect to aligned variables only, then we discuss shadow variables in Section 4.5.5. To simplify notation, we write $\Gamma$ instead of $\Gamma^\circ$ for now since only aligned variables are discussed.

**Flow-Sensitivity** In each typing rule $pc + \Gamma \{ c \rightarrow c' \} \Gamma'$, an important invariant is that if $c$ runs on two memories that are aligned by $\Gamma$, then the final memories are aligned by $\Gamma'$.

Consider the assignment rule (T-Asgn). This rule computes the distance of $e$’s value, $n^\circ$, and updates the distance of $x$’s value after assignment to $n^\circ$.

More interesting are rules (T-If) and (T-While). In (T-If), we compute the typing environments after executing $c_1$ and $c_2$ as $\Gamma_1$ and $\Gamma_2$ respectively. Since each branch may update $x$’s distance in arbitrary way, $\Gamma_1(x)$ and $\Gamma_2(x)$ may differ. We note that numeric expressions and * type naturally form a two level lattice, where * is higher than any $n$. Hence, we use the following rule to merge two distances $d_1$ and $d_2$:

$$d_1 \sqcup d_2 \triangleq \begin{cases} d_1 & \text{if } d_1 = d_2 \\ * & \text{otherwise} \end{cases}$$

For example, $(3 \sqcup 4 = *)$, $(x + y \sqcup x + y = x + y)$, $(x \sqcup 3 = *)$. Hence, (T-If) ends with $\Gamma_1 \sqcup \Gamma_2$, defined as $\lambda x. \Gamma_1(x) \sqcup \Gamma_2(x)$.

As an optimization, we also use branch conditions to simplify distances. Consider our motivating example (Figure 4-1): at Line 4, $\eta$ has (aligned) distance $\Omega ? 2 : 0$, where $\Omega$ is the branch condition. Its distance is simplified to 2 in the true branch and 0 in the false branch.
Rule (T-While) is similar, except that it requires a fixed point $\Gamma_f$ such that $pc \vdash \Gamma \sqcup \Gamma_f \{c\} \Gamma_f$. In fact, this rule is deterministic since we can construct the fixed point as follows (the construction is similar to the one in [50]):

$$pc \vdash \Gamma' \{c \rightarrow c'_i\} \Gamma''_i \text{ for all } 0 \leq i \leq n$$

where $\Gamma'_0 = \Gamma, \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma, \Gamma''_{n+1} = \Gamma''_n$.

It is easy to check that $\Gamma''_n = \Gamma'_{n+1} = \Gamma''_n \sqcup \Gamma$ and $pc' \vdash \Gamma_n' \{c \rightarrow c'_i\} \Gamma''_n$ by construction. Hence, $\Gamma''_n$ is a fixed point: $pc \vdash \Gamma \sqcup \Gamma''_n \{c \rightarrow c'_i\} \Gamma''_n$. Moreover, the computation above always terminates since all typing rules are monotonic on typing environments\(^1\) and the lattice has a height of 2.

**Maintaining Dynamically Tracked Distances** Each typing rule $pc \vdash \Gamma \{c \rightarrow c'\} \Gamma'$ also sets the value of $\vec{x}^\circ$ to maintain distance dynamically whenever $\Gamma'(x) = \ast$. This is achieved by the instrumented commands in $c'$.

None of rules (T-Skip, T-Asgn, T-Seq, T-Ref) generate $\ast$ type, hence they do not need any instrumentation. The merge operation in rule (T-If) generates $\ast$ type when $\Gamma_1(x) \neq \Gamma_2(x)$. In this case, we use the auxiliary instrumentation rule in the form of $\Gamma_1, \Gamma_2, pc \Rightarrow c'$, assuming $\Gamma_1 \sqsubseteq \Gamma_2$. In particular, for each variable $x$ whose distance is “upgraded” to $\ast$, the rule sets $\vec{x}^\circ$ to the distance previously tracked by the type system ($\Gamma_1(x)$). Moreover, the instrumentation commands $c''_1, c''_2$ are inserted under their corresponding branches.

Consider the following example:

```plaintext
if (x > 1) x := y; else y := 1;
```

starting with $\Gamma_0 : \{x : 1, y : 0\}$. In the true branch, rule (T-Asgn) updates $x$ to the distance of $y$, resulting $\Gamma_1 : \{x : 0, y : 0\}$. Similarly, we get $\Gamma_2 : \{x : 1, y : 0\}$ in the false branch. Moreover, when we merge the typing environments $\Gamma_1$ and $\Gamma_2$ at the end of branch, the typing environment becomes $\Gamma_3 = \Gamma_1 \sqcup \Gamma_2 = \{x : \ast, y : 0\}$. Since $\Gamma_1(x) \neq \Gamma_2(x)$, instrumentation rule is also applied, which instruments $\vec{x}^\circ := \emptyset$ after $x := y$ and $\vec{x}^\circ := 1$ after $y := 1$.

Rule (T-While) may also generate $\ast$ types. Following the same process in rule (T-If), it also uses the instrumentation rule to update corresponding dynamically tracked distance variables. The instrumentation command $c_s$ is inserted before loop and $c''$ after the commands in the loop body.

\(^1\)That is, $\forall pc, c, \Gamma_1, \Gamma_2, \Gamma'_1, \Gamma'_2, c_1, c_2. pc \vdash \Gamma'_1 \{c \rightarrow c'_i\} \Gamma''_i \in \{1, 2\} \land \Gamma_1 \sqsubseteq \Gamma_2 \implies \Gamma''_1 \sqsubseteq \Gamma''_2$. 
\( \langle r, \Gamma \rangle^* = r \quad \langle \text{true}, \Gamma \rangle^* = \text{true} \quad \langle \text{false}, \Gamma \rangle^* = \text{false} \quad \langle x, \Gamma \rangle^* = x + \pi \uparrow \), if \( \Gamma \vdash x : \text{num}_{(\delta, \epsilon)} \),
else
\( \langle e_1 \text{ op } e_2, \Gamma \rangle^* = \langle e_1, \Gamma \rangle^* \text{ op } \langle e_2, \Gamma \rangle^* \) where \( \text{op} = \oplus \cup \otimes \cup \odot \)
\[ \langle e_1 [e_2], \Gamma \rangle^* = \begin{cases} e_1 [e_2] + \overset{\cdot}{e_1}[e_2] & \text{if } \Gamma \vdash e_1 : \text{list} \text{ num} \text{, else} \\ e_1 [e_2] & \end{cases} \]
\( \langle e_1 ? e_2 : e_3, \Gamma \rangle^* = \langle e_1 \rangle^* \ ? \langle e_2, \Gamma \rangle^* : \langle e_3, \Gamma \rangle^* \)
\( \langle \text{skip}, \Gamma \rangle^* = \text{skip} \quad \langle c_1; \Gamma \rangle^* = c'_1 \quad \langle c_2; \Gamma \rangle^* = c'_2 \quad \langle x := e, \Gamma \rangle^* = (\overset{\cdot}{x} := \langle e, \Gamma \rangle^* - x) \)
\( \langle \text{if } e \text{ then } c_1 \text{ else } c_2, \Gamma \rangle^* = \text{if } \langle e, \Gamma \rangle^* \text{ then } c'_1 \text{ else } c'_2 \)
\( \langle \text{while } e \text{ do } c, \Gamma \rangle^* = \text{while } \langle e, \Gamma \rangle^* \text{ do } c' \)

Figure 4-7: Transformation of expressions and commands for aligned and shadow execution, where \( \star \in \{0, \uparrow\} \).

**Well-Formedness**  Whenever an assignment \( x := e \) is executed, no variable’s distance should depend on \( x \). To see why, consider \( x := 2 \) with initial \( \Gamma^\circ(y) = x \) and \( m(x) = 1 \). Since this assignment does not modify the value of \( y \), the aligned value of \( y \) (i.e., \( y + \Gamma^\circ(y) \)) should not change. However, \( \Gamma^\circ(y) \) changes from 1 to 2 after the assignment.

To avoid this issue, we check the following condition for each assignment \( x := e : \forall y \in \text{Vars}. \ x \notin \text{Vars}(\Gamma(y)) \). In case that the check fails for some \( y \), we promote its distance to \( \bullet \), and use the auxiliary instrumentation \( \Rightarrow \) to set \( \overset{\bullet}{y} \) properly. Hence, well-formedness is guaranteed: no variable’s distance depends on \( x \) when \( x \) is updated.

**Aligned Branches**  For differential privacy, we require the aligned execution to follow the same branch as the original execution. Due to dynamically tracked distances, statically checking that in a type system could be imprecise. Hence, we use assertions in rules (T-Ir) and (T-While) to ensure the aligned execution does not diverge. In those rules, \( \langle e, \Gamma \rangle^\circ \) simply computes the value of \( e \) in the aligned execution; its full definition, along with the definition of \( \langle e, \Gamma \rangle^\uparrow \) for the shadow execution, is shown in Figure 4-7.
4.5.5 Shadow Variables

In most typing rules, shadow variables are handled in the same way as aligned ones, which is discussed above. The key difference is that the type system allows the shadow execution to take a different branch from the original execution.

The extra permissiveness is the key ingredient of verifying algorithms such as Report Noisy Max. To see why, consider the example in Figure 4-2, where the shadow execution runs on $D_2$ with same random noise as from the execution on $D_1$. Upon the second query, the shadow execution does not update max, since its noisy value 3 is the same as the previous max; however, execution on $D_1$ will update max, since the noisy query value of 4 is greater than the previous max of 2.

To capture the potential divergence of shadow execution, each typing rule is associated with a program counter $pc$ with two possible values $\bot$ and $\top$ (introducing program counters in a type system is common in information flow control to track implicit flows [51]). Here, $\top$ (resp. $\bot$) means that the shadow execution might take a different branch (resp. must take the same branch) as the original execution.

When $pc = \bot$, the shadow execution is checked in the same way as aligned execution. When $pc = \top$, the shadow distances are updated (as done in Rule (T-Asgn)) so that $x + x^{\top}$ remains the same. The new value from the shadow execution will be maintained by the type system when $pc$ transits from $\bot$ to $\top$ by code instrumentation for sub-commands in (T-If) and (T-While), as we show next.

Take a branch (if $e$ then $c_1$ else $c_2$) for example. The transition happens when $pc = \bot \land pc' = \top$. In this case, we construct a shadow execution of $e$ by an auxiliary function $\langle c, \Gamma \rangle^{\top}$ defined in Figure 4-7. The shadow execution essentially replaces each variable $x$ with their correspondence (i.e., $x + x^{\top}$), as is standard in self-composition [52, 53]. The only differences are:

1. $\langle c, \Gamma \rangle^{\top}$ is not applicable to sampling commands, since if the original execution takes a sample while the shadow execution does not, we are unable to align the sample variable due to different probabilities.

2. For convenience, we use $x + v^{\top}$ where $\Gamma \vdash x : \langle p, p' \rangle$ whenever the shadow value of $x$ is used; correspondingly, we update $x^{\top}$ to $v - x$ instead of updating the shadow value of $x$ to some value $v$.

Rule (T-While) is very similar in its way of handling shadow variables.
4.5.6 Sampling Command

Rule (T-LAPLACE) checks the only probabilistic command $\eta := \text{Lap } r, S, \pi_\eta$ in ShadowDP. Here, the selector $S$ and numeric distance $\pi_\eta$ are annotations provided by a programmer to aid type checking. For the sample $\eta$, the aligned distance is specified by $\pi_\eta$ and the shadow distance is always 0 (since by definition, shadow execution use the same sample as the original program). Hence, the type of $\eta$ becomes $\text{num}_{(\pi_\eta, 0)}$.

Moreover, the selector constructs the aligned execution from either the aligned ($\circ$) or shadow ($\dagger$) execution. Since the selector may depend on a condition $e$, we use the selector function $S((e_1, e_2))$ in Figure 4-6 to do so.

Rule (T-LAPLACE) also checks that each $\eta$ is generated in an injective way: the same aligned value of $\eta$ implies the same value of $\eta$ in the original execution.

Consider the sampling command in Figure 4-1. The typing environments before ($\Gamma_0$) and after ($\Gamma_1$) the command is shown below (we omit unrelated parts for brevity):

$$\Gamma_0 : \{ \text{bq} : (*, *), \cdots \}$$
$$\eta := \text{Lap } (2/e), \Omega \dagger : \circ, \Omega? 2 : 0;$$
$$\Gamma_1 : \{ \text{bq} : (\Omega ? \widehat{\text{bq}} \dagger : \widehat{\text{bq}}\circ, \widehat{\text{bq}} \dagger), \eta : (\Omega ? 2 : 0, 0), \cdots \}$$

In this example, $S$ is $\Omega ? \dagger : \circ$. So the aligned distance of variable bq will be $\Omega ? \widehat{\text{bq}}\dagger : \widehat{\text{bq}}\circ$, the shadow distance of variable bq is still $\widehat{\text{bq}}\dagger$. The distance of $\eta$ is $(\Omega ? 2 : 0, 0)$, where the aligned part is given in the annotation.

4.6 Target Language

One goal of ShadowDP is to enable verification of $\epsilon$-differential privacy using off-the-shelf verification tools. In the transformed code so far, we assumed assert commands to verify that certain condition holds. The only remaining challenging feature is the sampling commands, which requires probabilistic reasoning. Motivated by LightDP [14], we note that for $\epsilon$-differential privacy, we are only concerned with the maximum privacy cost, not its likelihood. Hence, in the final step, we simply replace the sampling command with a non-deterministic command havoc $\eta$, which semantically sets the variable $\eta$. 
\eta := \text{Lap} \ r; S, r \eta \Rightarrow \text{havoc} \ \eta; v_\epsilon := S((v_\epsilon, 0)) + |r_\eta|/r;

c \not\Rightarrow c, \text{ if } c \text{ is not a sampling command}

Figure 4-8: Transformation rules to the target language. Probabilistic commands are reduced to non-deterministic ones.

\begin{align*}
\text{[skip]}_m &= \{m\} \\
\text{x := e}_m &= \{m([e]_m/x)\} \\
\text{[havoc x]}_m &= \bigcup_{r \in \mathbb{R}} \{m[r/x]\} \\
\text{[c_1;c_2]}_m &= \bigcup_{m' \in [c_1]_m} [c_2]_{m'} \\
\text{[if e then c_1 else c_2]}_m &= \begin{cases} [c_1]_m & \text{if } [e]_m = \text{true} \\ [c_2]_m & \text{if } [e]_m = \text{false} \end{cases} \\
\text{[while e do c]}_m &= (\text{fix}(\lambda f. \lambda m. \text{if } [e]_m = \text{true} \ (\cup_{m' \in [c]_m} f m') \ \text{else } \{m\} ) \ m) \\
\text{[c:return e]}_m &= \bigcup_{m' \in [c]_m} \{[e]_{m'}\}
\end{align*}

Figure 4-9: Semantics for the target language.

to an arbitrary value upon execution, as shown in Figure 4-8. The denotational semantics interprets a command \( c \) in the target language as a function \([c] : M \rightarrow \mathcal{P}(M)\). The semantics of commands are formalized in Figure 4-9.

Note that a distinguished variable \( v_\epsilon \) is added by the type system to explicitly track the privacy cost of the original program. For Laplace distribution, aligning \( \eta \) by the distance of \( r_\eta \) is associated with a privacy cost of \(|r_\eta|/r\). The reason is that the ratio of any two points that are \(|r_\eta|\) apart in the Laplace distribution with scaling factor \( r \) is bounded by \( \exp(|r_\eta|/r) \). Since the shadow execution uses the same sample, it has no privacy cost. This very fact allows us to reset privacy cost when the shadow execution is used (i.e., \( S \) selects \( \dagger \)): the rule sets privacy cost to \( 0 + |r_\eta|/r \) in this case.

In Figure 4-1, \( v_\epsilon \) is set to \( \Omega ? 0 : v_\epsilon + \Omega ? \epsilon : 0 \) which is the same as \( \Omega ? \epsilon : v_\epsilon \). Intuitively, that implies that the privacy cost of the entire algorithm is either \( \epsilon \) (when a new max is found) or the same as the previous value of \( v_\epsilon \).
The type system guarantees the following important property: if the original program type checks and the privacy cost $v_\epsilon$ in the target language is bounded by some constant $\epsilon$ in all possible executions of the program, then the original program satisfies $\epsilon$-differential privacy. We will provide a soundness proof in the next section. Consider the running example in Figure 4-1. The transformed program in the target language is shown at the bottom. With a model checking tool CPAchecker [47], we verified that $v_\epsilon \leq \epsilon$ in the transformed program within 2 seconds (Section 4.8.3). Hence, the Report Noisy Max algorithm is verified to be $\epsilon$-differentially private.

4.7 Soundness

The type system performs a two-stage transformation:

$$pc \vdash \Gamma_1 \{ e \rightarrow c' \} \Gamma_2 \text{ and } c' \Rightarrow c''$$

Here, both $c$ and $c'$ are probabilistic programs; the difference is that $c$ executes on the original memory without any distance tracking variables; $c'$ executes on the extended memory where distance tracking variables are visible. In the second stage, $c'$ is transformed to a non-probabilistic program $c''$ where sampling instructions are replaced by havoc and the privacy cost $v_\epsilon$ is explicit. In this section, we use $c$, $c'$, $c''$ to represent the source, transformed, and target program respectively.

Overall, the type system ensures $\epsilon$-differential privacy (Theorem 4.5): if the value of $v_\epsilon$ in $c''$ is always bounded by a constant $\epsilon$, then $c$ is $\epsilon$-differentially private. In this section, we formalize the key properties of our type system and prove its soundness. The complete proofs are available in the Appendix.

Extended Memory Command $c'$ is different from $c$ since it maintains and uses distance tracking variables. To close the gap, we first extend memory $m$ to include those variables, denoted as $\overline{\text{Vars}} = \bigcup_{x \in \text{Vars}} \{ \overline{x}, \overline{x'} \}$ and introduce a distance environment $\gamma : \overline{\text{Vars}} \rightarrow \mathbb{R}$. 
Definition 4.1. Let \( \gamma : \overline{\text{Vars}} \rightarrow \mathbb{R} \). For any \( m \in M \), there is an extension of \( m \), written \( m \uplus (\gamma) \), such that

\[
m \uplus (\gamma)(x) = \begin{cases} 
m(x), & x \in \text{Vars} \\
\gamma(x), & x \in \overline{\text{Vars}}
\end{cases}
\]

We use \( M' \) to denote the set of extended memory states and \( m'_1, m'_2 \) to refer to concrete extended memory states. We note that although the programs \( c \) and \( c' \) are probabilistic, the extra commands in \( c' \) are deterministic. Hence, \( c' \) preserves the semantics of \( c \), as formalized by the following Lemma.

Lemma 4.1 (Consistency). Suppose \( pc \vdash \Gamma_1 \{ c \rightarrow c' \} \Gamma_2 \). Then for any initial and final memory \( m_1, m_2 \) such that \( \llbracket c \rrbracket_{m_1}(m_2) \neq 0 \), and any extension \( m'_1 \) of \( m_1 \), there is a unique extension \( m'_2 \) of \( m_2 \) such that

\[
\llbracket c' \rrbracket_{m'_1}(m'_2) = \llbracket c \rrbracket_{m_1}(m_2)
\]

Proof. By structural induction on \( c \). The only interesting case is the (probabilistic) sampling command, which does not modify distance tracking variables. \( \square \)

From now on, we will use \( m'_2 \) to denote the unique extension of \( m_2 \) satisfying the property above.

\( \Gamma \)-Relation To formalize and prove the soundness property, we notice that a typing environment \( \Gamma \) along with distance environment \( \gamma \) induces two binary relations on memories. We write \( m_1 \uplus (\gamma) \Gamma^\circ m_2 \) (resp. \( m_1 \uplus (\gamma) \Gamma^\uparrow m_2 \)) when \( m_1, m_2 \) are related by \( \Gamma^\circ \) (resp. \( \Gamma^\uparrow \)) and \( \gamma \). Intuitively, the initial \( \gamma \) and \( \Gamma \) (given by the function signature) specify the adjacency relation, and the relation is maintained by the type system throughout program execution. For example, the initial \( \gamma \) and \( \Gamma \) in Figure 4-1 specifies that two executions of the program is related if non-private variables \( \epsilon, \text{size} \) are identical, and each query answer in \( q[i] \) differs by at most one.

To facilitate the proof, we simply write \( m'_1 \Gamma m_2 \) where \( m'_1 \) is an extended memory in the form of \( m_1 \uplus (\gamma) \).

Definition 4.2 (\( \Gamma \)-Relations). Two memories \( m'_1 \) (in the form of \( m_1 \uplus (\gamma) \)) and \( m_2 \) are related by \( \Gamma^\circ \), written \( m'_1 \Gamma^\circ m_2 \), if \( \forall x \in \text{Vars} \cup \overline{\text{Vars}} \), we have

\[
m_2(x) = m'_1(x) + m'_1(d^0) \quad \text{if} \quad \Gamma \vdash x : \text{num}_{(d^0, d^0)}
\]
We define the relation on non-numerical types and the \( \Gamma^\dagger \) relation in a similar way.

By the definition above, \( \Gamma^\circ \) introduces a function from \( \mathcal{M} \) to \( \mathcal{M} \). Hence, we use \( \Gamma^\circ m'_1 \) as the unique \( m_2 \) such that \( m'_1 \Gamma^\circ m_2 \). The \( \Gamma^\dagger \) counterparts are defined similarly.

**Injectivity** For alignment-based proofs, given any \( \gamma \), both \( \Gamma^\circ \) and \( \Gamma^\dagger \) must be injective functions [14]. The injectivity of \( \Gamma \) over the entire memory follows from the injectivity of \( \Gamma \) over the random noises \( \eta \in RVars \), which is checked as the following requirement in Rule (T-LAPLACE):

\[
\Psi \Rightarrow ((\eta + \pi_\eta)\{\eta_1/\eta\} = (\eta + \pi_\eta)\{\eta_2/\eta\} \Rightarrow \eta_1 = \eta_2)
\]

where all variables are universally quantified. Intuitively, this is true since the non-determinism of the program is purely from that of \( \eta \in RVars \).

**Lemma 4.2** (Injectivity). Given \( c, c', pc, m', m'_1, m'_2, \Gamma_1, \Gamma_2 \) such that \( pc \vdash \Gamma_1 \{ c \rightarrow c' \} \Gamma_2 \), \( \llbracket c \rrbracket_{m'} m'_1 \neq 0 \land \llbracket c' \rrbracket_{m'} m'_2 \neq 0 \), \( \star \in \{ \circ, \dagger \} \), then we have

\[
\Gamma_2^\star m'_1 = \Gamma_2^\star m'_2 \implies m'_1 = m'_2
\]

**Soundness** The soundness theorem connects the “privacy cost” of the probabilistic program to the distinguished variable \( v \in \text{Vars} \) in the target program \( c'' \). To formalize the connection, we first extend memory one more time to include \( v \):

**Definition 4.3.** For any extended memory \( m' \) and constant \( \epsilon \), there is an extension of \( m' \), written \( m' \uplus (\epsilon) \), so that

\[
m' \uplus (\epsilon)(v) = \epsilon, \quad \text{and} \quad m' \uplus (\epsilon)(x) = m(x), \quad \forall x \in \text{dom}(m').
\]

For a transformed program and a pair of initial and final memories \( m'_1 \) and \( m'_2 \), we identify a set of possible \( v \) values, so that in the corresponding executions of \( c'' \), the initial and final memories are extensions of \( m'_1 \) and \( m'_2 \) respectively:

**Definition 4.4.** Given \( c' \Rightarrow c'', m'_1 \) and \( m'_2 \), the consistent costs of executing \( c'' \) w.r.t. \( m'_1 \) and \( m'_2 \), written \( c'' \upharpoonright_{m'_1} \), is defined as
Since \( (c'' \pmod{m'_1} \triangleleft (s : \epsilon \cup (\epsilon) \in \langle \epsilon' \rangle \cup (\epsilon') \cup (\epsilon')) \) by definition is a set of values of \( v \), we write \( \max (c'' \pmod{m'_1} \triangleright (s : \epsilon)) \) for the maximum cost.

The next lemma enables precise reasoning of privacy cost w.r.t. a pair of initial and final memories:

**Lemma 4.3 (Pointwise Soundness).** Let \( p_c, c, c', c'', \Gamma_1, \Gamma_2 \) be such that \( p_c \dashv \Gamma_1 \{ c \rightarrow c' \} \Gamma_2 \land c' \Rightarrow c'' \), then \( \forall m'_1, m'_2:

(i) the following holds:

\[
\lceil c \rceil_{m'_1} (m'_2) \leq \lceil c \rceil_{m'_1} (\Gamma_1 m'_2) \quad \text{when } p_c = \perp
\]

(ii) one of the following holds:

\[
\lceil c \rceil_{m'_1} (m'_2) \leq \exp (\max (c'' \pmod{m'_1} \triangleright (\Gamma_1 \triangleright \Gamma_2) m'_2)) \quad \text{(4.2a)}
\]

\[
\lceil c \rceil_{m'_1} (m'_2) \leq \exp (\max (c'' \pmod{m'_1} (\Gamma_1 \triangleright \Gamma_2) m'_2)) \quad \text{(4.2b)}
\]

The point-wise soundness lemma provides a precise privacy bound per initial and final memory. However, differential privacy by definition (Definition 2.1) bounds the worst-case cost. To close the gap, we define the worst-case cost of the transformed program.

**Definition 4.5.** For any program \( c'' \) in the target language, we say the execution cost of \( c'' \) is bounded by some constants \( \epsilon \), written \( c'' \leq \epsilon \), iff for any \( m'_1, m'_2,

\[
m'_2 \cup (\epsilon') \in \lceil c'' \rceil_{m'_1} \cup (\epsilon) \Rightarrow \epsilon' \leq \epsilon
\]

Note that off-the-shelf tools can be used to verify that \( c'' \leq \epsilon \) holds for some \( \epsilon \).

**Theorem 4.4 (Soundness).** Given \( c, c', c'', m'_1, \Gamma_1, \Gamma_2, \epsilon \) such that \( \perp \vdash \Gamma_1 \{ c \rightarrow c' \} \Gamma_2 \land c' \Rightarrow c'' \land c'' \leq \epsilon \), one of the following holds:

\[
\max_{S \subseteq M'} (\lceil c \rceil_{m'_1} (S) - \exp (\epsilon)) \leq 0,
\]

\[
\max_{S \subseteq M'} (\lceil c \rceil_{m'_1} (S) - \exp (\epsilon)) \leq 0.
\]
Proof. By definition of \( c'' \leq \varepsilon \), we have \( \max(e'' | m'_2) \leq \varepsilon \) for all \( m'_2 \in S \). Thus, by Lemma 4.3, we have one of the two:

\[
\begin{align*}
\mathbb{G}_i\mathbb{M}_i(m'_2) & \leq \exp(\varepsilon)\mathbb{G}_i\mathbb{M}_i(\Gamma^0_2m'_2), & \forall m'_2 \in S, \\
\mathbb{G}_i\mathbb{M}_i(m'_2) & \leq \exp(\varepsilon)\mathbb{G}_i\mathbb{M}_i(\Gamma^0_2m'_2), & \forall m'_2 \in S.
\end{align*}
\]

If the first inequality is true, then

\[
\max_{S \subseteq \mathcal{M}'} \left( \mathbb{G}_i\mathbb{M}_i(S) - \exp(\varepsilon)\mathbb{G}_i\mathbb{M}_i(\Gamma^0_2S) \right) = \max_{S \subseteq \mathcal{M}'} \sum_{m'_2 \in S} \left( \mathbb{G}_i\mathbb{M}_i(m'_2) - \exp(\varepsilon)\mathbb{G}_i\mathbb{M}_i(\Gamma^0_2m'_2) \right) \leq 0
\]

and therefore (4.3a) holds. Similarly, (4.3b) holds if the second inequality is true. Note that the equality above holds due to the injective assumption, which allows us to derive the set-based privacy from the point-wise privacy (Lemma 4.3).

We now prove the main theorem on differential privacy:

**Theorem 4.5 (Privacy).** Given \( \Gamma_1, \Gamma_2, c, c', c'', e, \varepsilon \) such that

\[
\Gamma^\circ_1 = \Gamma^\dagger_1 \land \bot \Gamma_1 \{ (c; \text{return } e) \rightarrow (c'; \text{return } e) \} \Gamma_2 \land c' \Rightarrow c''
\]

we have

\[
c'' \leq \varepsilon \Rightarrow c \text{ is } \varepsilon \text{-differentially private.}
\]

Proof. By the typing rule, we have \( \bot \Gamma_1 \{ c \rightarrow c' \} \Gamma_2 \). By the soundness theorem (Theorem 4.4) and the fact that \( \Gamma^\circ_1 = \Gamma^\dagger_2 \), we have \( \mathbb{G}_i\mathbb{M}_i(S) \leq \exp(\varepsilon)\mathbb{G}_i\mathbb{M}_i(\Gamma^0_2S) \). For clarity, we stress that all sets are over distinct elements (as we have assumed throughout this dissertation).

By rule (T-RETURN), \( \Gamma_2 \vdash e : \text{num}_{(0,e)} \) or \( \Gamma_2 \vdash e : \text{bool} \). For any set of values \( V \subseteq \mathcal{B} \), let

\[
S'_{\mathcal{V}} = \{ m' \in \mathcal{M}' | [e]_{m'} \in V \} \land S_{\mathcal{V}} = \{ m \in \mathcal{M} | [e]_m \in V \}
\]

then we have \( \Gamma^\circ_2S'_{\mathcal{V}} \subseteq S_{\mathcal{V}} \):

\[
m \in \Gamma^\circ_2S'_{\mathcal{V}} \Rightarrow m = \Gamma^\circ_2m' \text{ for some } m' \in S_{\mathcal{V}}
\]

\[
\Rightarrow [e]_m = [e]_{\Gamma^\circ_2m'} = [e]_{m'} \in V
\]
\[ \Rightarrow m \in S_V. \]

The equality in second implication is due to the zero distance when \( \Gamma_2 \vdash e : \text{num}_{(0,n)} \), and rule (T-ODot) when \( \Gamma_2 \vdash e : \text{bool} \). We note that \( \Gamma_2^\circ S'_V \neq S_V \) in general since \( \Gamma_2^\circ \) might not be a surjection. Let \( P' = (c'; \text{return } e) \), then for any \( \gamma \), we have

\[
\begin{align*}
\llbracket P' \rrbracket_{m_1 \triangledown (\gamma)}(V) &= \llbracket c' \rrbracket_{m_1 \triangledown (\gamma)}(S'_V) \\
&\leq \exp(\epsilon) \llbracket c' \rrbracket_{\Gamma_1^\circ/m_1 \triangledown (\gamma)}(\Gamma_2^\circ S'_V) \\
&\leq \exp(\epsilon) \llbracket c' \rrbracket_{\Gamma_1^\circ/m_1 \triangledown (\gamma)}(S_V) \\
&= \exp(\epsilon) \llbracket P \rrbracket_{\Gamma_1^\circ/m_1 \triangledown (\gamma)}(V).
\end{align*}
\]

Finally, due to Lemma 4.1, \( \llbracket P \rrbracket_{m_1}(V) = \llbracket P' \rrbracket_{m_1 \triangledown (\gamma)}(V) \). Therefore, by definition of privacy \( c \) is \( \epsilon \)-differentially private. \( \square \)

Note that the shallow distances are only useful for proofs; they are irrelevant to the differential privacy property being obeyed by a program. Hence, initially, we have \( \Gamma_1^\circ = \Gamma_1^\dagger \) (both describing the adjacency requirement) in Theorem 2, as well as in all of the examples formally verified by ShadowDP.

### 4.8 Implementation and Evaluation

#### 4.8.1 Implementation

We have implemented ShadowDP into a trans-compiler\(^2\) in Python. ShadowDP currently supports trans-compilation from annotated C code to target C code. Its workflow includes two phases: transformation and verification. The annotated source code will be checked and transformed by ShadowDP; the transformed code is further sent to a verifier.

**Transformation** ShadowDP tracks the typing environments in a flow-sensitive way, and instruments corresponding statements when appropriate. Moreover, ShadowDP adds an assertion \( \text{assert } (v_\epsilon \leq \epsilon) \)

\(^2\)Publicly available at https://github.com/cm1a-psu/shadowdp.
before the return command. This assertion specifies the final goal of proving differential privacy. The implementation follows the typing rules explained in Section 4.5.

**Verification** The goal of verification is to prove the assertion $\text{assert } (v_e \leq e)$ never fails for any possible inputs that satisfy the precondition (i.e., the adjacency requirement). To demonstrate the usefulness of the transformed programs, we use a model checker CPAChecker [47] v1.8. CPAChecker is capable of automatically verifying C program with a given configuration. In our implementation, predicate analysis is used. Also, CPAChecker has multiple solver backends such as MathSat [54], Z3 [49] and SMTInterpol [55]. For the best performance, we concurrently use different solvers and return the results as soon as any one of them verifies the program.

One limitation of CPAChecker and many other tools, is the limited support for non-linear arithmetics. For programs with non-linear arithmetics, we take two approaches. First, we verify the algorithm variants where $\epsilon$ is fixed (the approach taken in [16]). In this case, all transformed code in our evaluation is directly verified without any modification. Second, to verify the correctness of algorithms with arbitrary $\epsilon$, we slightly rewrite the non-linear part in a linear way or provide loop invariants (see Section 4.8.2). We report the results from both cases whenever we encounter this issue.

### 4.8.2 Case Studies

We investigate some interesting differentially private algorithms that are formally verified by ShadowDP.

**Sparse Vector Technique** Sparse Vector Technique (SVT) [28] is a powerful mechanism which has been proven to satisfy $\epsilon$-differential privacy (its proof is notoriously tricky to write manually [12]). In this section we show how ShadowDP verifies this algorithm and later show how a novel variant is verified.

Figure 4-10 shows the pseudo code of Sparse Vector Technique [28]. It examines the input queries and reports whether each query is above or below a threshold $T$. To achieve differential privacy, it first adds Laplace noise to the threshold $T$, compares the noisy query answer $q[i] + \eta_2$ with the noisy threshold $T_\ast$, and returns the result (true or false). The number of true’s the algorithm can output is
function SVT ($\varepsilon$, size, $T$, $N$: num(0,0); $q$: list num($\varepsilon$,+))
returns (out: list bool)
precondition $\forall i. -1 \leq \mathbb{Q}[i] \leq 1$

1. $\eta_1 := \text{Lap}(2/\varepsilon),\circ,1$;
2. $T_* := T + \eta_1; \text{count} := 0; \text{i} := 0$;
3. while (count $< N \land i < \text{size})$
   4. $\eta_2 := \text{Lap}(4N/\varepsilon),\circ,\Omega?2:0$;
   5. if ($q[i] + \eta_2 \geq T_*$) then
      6. out := true::out;
      7. count := count + 1;
   8. else
      9. out := false::out;
   10. $i := i + 1$;

The transformed program (slightly simplified for brevity) with underlined parts added by the type system:

1. $v_\varepsilon := 0$;
2. havoc $\eta_1; v_\varepsilon := v_\varepsilon + \varepsilon/2$;
3. $T_* := T + \eta_1; \text{count} := 0; \text{i} := 0$;
4. while (count $< N \land i < \text{size})$
   5. assert (count $< N \land i < \text{size})$;
   6. havoc $\eta_2; v_\varepsilon = \Omega?(v_\varepsilon + 2\times\varepsilon/4N):(v_\varepsilon + 0)$;
   7. if ($q[i] + \eta_2 \geq T_*$) then
      8. assert ($q[i] + \tilde{q}\circ[i] + \eta_2 + 2 \geq T_* + 1$);
      9. out := true::out;
     10. count := count + 1;
   11. else
      12. assert ($\neg(q[i] + \tilde{q}\circ[i] + \eta_2 \geq T_* + 1)$);
     13. out := false::out;
    14. $i := i + 1$;

Figure 4-10: Verifying Sparse Vector Technique with ShadowDP. Annotations are in gray where $\Omega$ represents the branch condition.

bounded by argument $N$. One key observation is that once the noise has been added to the threshold, outputting false pays no privacy cost [28]. As shown in Figure 4-10, programmers only have to provide two simple annotations: “$\circ,1$” for $\eta_1$ and “$\circ,\Omega?2:0$” for $\eta_2$. Since the selectors in this example only select aligned version of variables, the shadow execution is optimized away (controlled by $pc$ in rule (T-If)). ShadowDP successfully type checks and transforms this algorithm. However, due to a
nonlinear loop invariant that CPAChecker fails to infer, it fails to verify the program. With the loop invariant provided manually, the verification succeeds, proving this algorithm satisfies $\epsilon$-differential privacy (we also verified a variant where $\epsilon$ is fixed to $N$ to remove the non-linearity).

**Gap Sparse Vector Technique** We now consider a novel variant of Sparse Vector Technique. In this variant, whenever $q[i] + \eta_2 \geq T_*$, it outputs the value of the gap $q[i] + \eta_2 - T_*$ (how much larger the noisy answer is compared to the noisy threshold). Note that the noisy query value $q[i] + \eta_2$ is reused for both this check and the output (whereas other proposals either (1) draw fresh noise and result in a larger $\epsilon$ [28], or (2) re-use the noise but do not satisfy differential privacy, as noted in [12]). For noisy query values below the noisy threshold, it only outputs `false`. We call this algorithm Gap Sparse Vector Technique (GapSVT). More specifically, Line 6 in Figure 4-10 is changed as follows:

\[
\text{out} := \text{true}; \quad \Rightarrow \ 	ext{out} := (q[i] + \eta_2 - T_*)::\text{out};
\]

This variant can be easily verified with little changes to the original annotation. One observation is that, to align the `out` variable, the gap appended to the list must have 0 aligned distance. Thus we change the distance of the alignment annotation for $\eta_2$ from “$\Omega \ ? 2 : 0$” to “$\Omega \ ? (1 - \bar{q}[i]) : 0$”, the other part of the annotation remains the same.

ShadowDP successfully type checks and transforms the program. Due to the non-linear arithmetics issue, we rewrite the assignment command as follows:

\[
v_\epsilon := v_\epsilon + (1 - \bar{q}[i]) \times \epsilon/4N; \quad \Rightarrow \ 	ext{assert} (|1 - \bar{q}[i]| \leq 2); v_\epsilon := v_\epsilon + 2 \times \epsilon/4N;
\]

and provide nonlinear loop invariants; then it is verified (we also verified a variant where $\epsilon$ is fixed to 1).

**Numerical Sparse Vector Technique** Numerical Sparse Vector Technique (NumSVT) [28] is an interesting variant of SVT which outputs estimated numerical query answers when the query answer is large. To achieve differential privacy, like SVT, it adds noise to the threshold $T$ and each query answer $q[i]$ and tests if the each noisy query answer is above the noisy threshold or not. The difference is that NumSVT draws a fresh noise $\eta_3$ when the noisy query answer is above the noisy threshold, and then releases $q[i] + \eta_3$ instead of simply releasing `true`. The pseudo code for this algorithm is shown in
Figure 4-11: Verifying Numerical Sparse Vector Technique with ShadowDP. Annotations are in gray where \( \Omega \) represents the branch condition.

Figure 4-11.

In this algorithm, ShadowDP needs an extra annotation for the new sampling command of \( \eta_3 \). The same trick when inferring the annotations in GapSVT can also be applied here. Recall the observation
function PartialSum (\(\epsilon, \text{size} : \text{num}(0,0) ; q : \text{list num}_{\ast,\ast}\))
returns (out : \text{num}(0,\ast))
precondition \(\forall i. -1 \leq (q[i]) \leq 1 \land (\forall i. (q[i]) \neq 0 \Rightarrow (\forall j. q[j] = 0))\)

1. \(\text{sum} := 0; i := 0;\)
2. \(\text{while} (i < \text{size})\)
3. \(\text{sum} := \text{sum} + q[i];\)
4. \(i := i + 1;\)
5. \(\eta = \text{Lap} (1/\epsilon), \circ, -\text{sum}^\circ;\)
6. \(\text{out} := \text{sum} + \eta;\)

The transformed program, where underlined commands are added by the type system:

1. \(\text{sum} := 0; i := 0;\)
2. \(\text{sum}^\circ := 0;\)
3. \(\text{while} (i < \text{size})\)
4. \(\text{assert} (i < \text{size});\)
5. \(\text{sum} := \text{sum} + q[i];\)
6. \(\text{sum}^\circ := \text{sum}^\circ + \tilde{q}^\circ[i];\)
7. \(i := i + 1;\)
8. \(\text{havoc} \eta; v_\epsilon := v_\epsilon + |\text{sum}^\circ| \times \epsilon;\)
9. \(\text{out} := \text{sum} + \eta;\)

Figure 4-12: Verifying Partial Sum using ShadowDP. Annotations are shown in gray.

that we want the final output variable \(\text{out}\) to have distance \(0, -\), which translates to the numerical query answer \(q[i] + \eta_3\) having a distance \(0, -\). Hence, \(\eta_3\) must have distance \(-\tilde{q}^\circ [i]\) and the alignment annotation for it is written as “\(0, -\tilde{q}^\circ [i]\)”. The rest of the annotations remain the same as standard SVT.

Due to the non-linear issue of the verifier, we rewrite the privacy cost assignments (Line 10) as follows:

\[ v_\epsilon = v_\epsilon + |\tilde{q}^\circ[i]| \times \epsilon/3N; \quad \Rightarrow \quad \text{assert} (|\tilde{q}^\circ[i]| \leq 1); v_\epsilon = v_\epsilon + \epsilon/3N; \]

Along with manual loop invariants provided, CPAChecker successfully verified the rewritten program.

**Partial Sum** We now study an \(\epsilon\)-differentially private algorithm PartialSum (Figure 4-12) which simply sums over a list of queries. To achieve differential privacy, it simply adds noise using Laplace mechanism to the final sum and output the noisy sum. One difference from the shown algorithms is the
adjacency assumption: at most one query answer may differ by 1, which is reflected in the precondition.

In this example, since the only noise is added to the sum immediately before return, there is no difference between aligned and shadow execution. Again, in order to make the distance of the final output 0, it is easy to deduce that the distance of $\eta$ should be $-\sum^\circ$. Adding the annotation $\circ -\sum^\circ$ to Line 5 in Figure 4-12, ShadowDP successfully transforms and verifies the program. Note that since the privacy cost assignment $v_\epsilon := v_\epsilon + |\sum| \times \epsilon$; contains non-linear arithmetic, we carefully rewrite this command to assert $(|\sum| \leq 1)$; $v_\epsilon := v_\epsilon + \epsilon$; ShadowDP is able to type check and verify this algorithm within seconds.

**Smart Sum and Prefix Sum** Another interesting algorithm Smart Sum (Figure 4-13) [56], which is designed to continually release aggregate statistics in a privacy-preserving manner, has been previously verified [17, 23] with heavy annotations. Notably, ShadowDP is able to verify Smart Sum with very little annotation burden for the programmers (Line 4 and 9) within seconds.

Inferring the alignment annotations for this algorithm is non-trivial, but the goal is the same: eliminate the aligned distance of the final output variable out. We noticed that variable next is calculated with noise and appended to out in each iteration. Hence, we assign distance $-\sum^\circ - \tilde{q}^\circ [i]$ to $\eta_1$ and $-\tilde{q}^\circ [i]$ to $\eta_2$ to eliminate the the distances introduced to next, making sure it always stays a 0-distance variable throughout the entire execution.

With the above annotations, Smart Sum is successfully type checked and transformed by ShadowDP. However, due to the non-linear issue, we rewrite the commands of Line 6 and Line 12 as follows:

\[
\begin{align*}
v_\epsilon &:= v_\epsilon + | -\sum^\circ - \tilde{q}^\circ [i]| \times \epsilon; & \Rightarrow & \text{if } (| -\sum^\circ - \tilde{q}^\circ [i]| > 0) \\
& \quad \textbf{assert } (| -\sum^\circ - \tilde{q}^\circ [i]| \leq 1); & v_\epsilon &:= v_\epsilon + \epsilon; \\

v_\epsilon &:= v_\epsilon + | -\tilde{q}^\circ [i]| \times \epsilon; & \Rightarrow & \text{if } (| -\tilde{q}^\circ [i]| > 0) \\
& \quad \textbf{assert } (| -\tilde{q}^\circ [i]| \leq 1); & v_\epsilon &:= v_\epsilon + \epsilon;
\end{align*}
\]

Moreover, one difference with Smart Sum from the other examples is that it in fact satisfies $2\epsilon$-differential privacy instead of $\epsilon$-differential privacy [56], thus the last assertion added to the program is
The transformed program with underlined parts added by the type system:

```
next:=0; i := 0; sum := 0;
while (i < size ∧ i ≤ T)
  if (i + 1) mod M = 0 then
    \( \eta_1 := \operatorname{Lap}(1/\epsilon), \sigma, -\sum^0 - \tilde{q}^0[i] \); 
    next:= sum + q[i] + \eta_1;
    sum := 0;
    out := next::out;
  else
    \( \eta_2 := \operatorname{Lap}(1/\epsilon), \sigma, -\tilde{q}^0[i] \);
    next:= next + q[i] + \eta_2;
    sum := sum + q[i];
    out := next::out;
    i := i + 1;
```

Figure 4-13: Verifying SmartSum algorithm with ShadowDP. Annotations are shown in gray.

changed to \( \text{assert } (v_\epsilon \leq 2 \times \epsilon) \);. Then, CPAChecker is able to verify this algorithm.

We also verified a simplified variant of Smart Sum named Prefix Sum algorithm [16], where the
Table 4-1: Time spent on type checking and verification.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type Check (s)</th>
<th>Verification by ShadowDP (s)</th>
<th>Verification by [16] (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report Noisy Max</td>
<td>0.465</td>
<td>1.932</td>
<td>22</td>
</tr>
<tr>
<td>Sparse Vector Technique (N = 1)</td>
<td>0.398</td>
<td>1.856</td>
<td>27</td>
</tr>
<tr>
<td>Sparse Vector Technique</td>
<td>0.399</td>
<td>2.629</td>
<td>580</td>
</tr>
<tr>
<td>Numerical Sparse Vector Technique (N = 1)</td>
<td>0.418</td>
<td>1.783</td>
<td>4</td>
</tr>
<tr>
<td>Numerical Sparse Vector Technique</td>
<td>0.421</td>
<td>2.584</td>
<td>5</td>
</tr>
<tr>
<td>Gap Sparse Vector Technique</td>
<td>0.424</td>
<td>2.494</td>
<td>N/A</td>
</tr>
<tr>
<td>Partial Sum</td>
<td>0.445</td>
<td>1.922</td>
<td>14</td>
</tr>
<tr>
<td>Prefix Sum</td>
<td>0.449</td>
<td>1.903</td>
<td>14</td>
</tr>
<tr>
<td>Smart Sum</td>
<td>0.603</td>
<td>2.603</td>
<td>255</td>
</tr>
</tbody>
</table>

else branch is always taken. More specifically, Prefix Sum can be retrieved by removing Lines 3 - 8 from Figure 4-13. The annotation remains the same for \( \eta_2 \) and type checking and transformation follows Smart Sum. Note that Prefix Sum satisfies \( \epsilon \)-differential privacy, so the last assertion remains unchanged. CPAChecker then verifies the transformed Prefix Sum within 2 seconds.

### 4.8.3 Experiments

ShadowDP is evaluated on Report Noisy Max algorithm (Figure 4-1) along with all the algorithms discussed in Section 4.8.2. For comparison, all the algorithms verified in [16] are included in the experiments (where Sparse Vector Technique is called Above Threshold in [16]). One exception is ExpMech algorithm, since ShadowDP currently lacks a sampling command for Exponential noise. However, as shown in [14], it should be fairly easy to add a noise distribution without affecting the rest of a type system.

Experiments are performed on a Dual Intel® Xeon® E5-2620 v4@2.10 GHz CPU machine with 64 GB memory. All algorithms are successfully checked and transformed by ShadowDP and verified by CPAChecker. For programs with non-linear arithmetics, we performed experiments on both solutions discussed in Section 4.8.2. Transformation and verification all finish within 3 seconds, as shown in Table 4-1, indicating the simplicity of analyzing the transformed program, as well as the practicality of verifying \( \epsilon \)-differentially private algorithms with ShadowDP.
4.8.4 Proof Automation

ShadowDP requires two kinds of annotations: (1) function specification and (2) annotation for sampling commands. As most verification tools, (1) is required since it specifies the property being verified. In all of our verified examples, (2) is fairly simple and easy to write as shown in our case studies. To further improve the usability of ShadowDP, we discuss some heuristics to automatically generate the annotations for sampling commands.

Sampling commands requires two parts of annotation:

1. **Selectors.** The selector has two options: aligned (⊙) or shadow (†), with potential dependence. The heuristic is to enumerate branch conditions. For Report Noisy Max, there is only one branch condition Ω, giving us four possibilities: ⊙ / † / Ω ? ⊙ † / Ω ? † ⊙.

2. **Alignments for the sample.** It is often simple arithmetic on a small integer such as 0, 1, 2 or the exact difference of query answers and other program variables. For dependent types, we can also use the heuristic of using branch conditions. For Report Noisy Max, this will discover the correct alignment Ω ? 2 : 0.

This enables the discovery of all the correct annotations for the algorithm studied in this chapter. We defer a systematic study of proof automation to Chapter 5.

4.9 Summary

In this chapter we presented ShadowDP, a new language for the verification of differential privacy algorithms. ShadowDP uses shadow execution to generate more flexible randomness alignments that allows it to verify more algorithms, such as Report Noisy Max. We also used it to verify a novel variant of Sparse Vector that reports the gap between noisy above-threshold queries and the noisy threshold.

Compared with prior work LightDP, which is also based on randomness alignment, ShadowDP is strictly more expressive than LightDP: LightDP is a restricted form of ShadowDP where the shadow execution is never used (i.e., when the selector always picks the aligned execution). Moreover, ShadowDP is equipped with a flow-sensitive type system, making it both more expressive and more usable, since
only sampling command need annotations. Lastly, ShadowDP allows extra permissiveness of allowing two related executions to take different branches, which is also crucial in verifying Report Noisy Max.

ShadowDP is also more efficient than another line of prior work based on probabilistic couplings due to the simplicity of randomness alignment. In fact, ShadowDP verifies all algorithms within 3 seconds while the coupling verifier takes 255 seconds in verifying Smart Sum and 580 seconds in verifying Sparse Vector (excluding proof synthesis time).
Chapter 5

An Automated and Integrated Approach for Proving Differential Privacy and Finding Precise Counterexamples

5.1 Introduction

Finding a counterexample is typically a two-phase process that (1) first searches an infinitely large space for candidate counterexamples and then (2) uses an exact symbolic probabilistic solver like PSI [57] to verify that the counterexample is indeed valid. The search phase currently presents the most problems (i.e., large runtimes or failure to find counterexamples are most often attributed to the search phase). Earlier search techniques were based on sampling (running a mechanism hundreds of thousands of times), which made them slow and inherently imprecise: even with enormous amounts of samples, they can still fail if a privacy-violating section of code is not executed frequently enough or if the actual privacy cost is slightly higher than the privacy claim. Recently, static program analyses were proposed to accomplish both goals [42, 43]. However, they either only analyze a non-trivial but restricted class of programs [42], or rely on heuristic strategies whose effectiveness on many subtle mechanisms is unclear [43].

In this chapter, we present CheckDP, an automated and integrated tool for proving or disproving the correctness of a mechanism that claims to be differentially private. Significantly, CheckDP automatically finds counterexamples via static analysis, making it unnecessary to run the mechanism. Like prior work [30], CheckDP still uses PSI [57] at the end. However, replacing sampling-based search with static analysis enables CheckDP to find violations in a few seconds, while previous sampling-based methods [29, 30] may fail even after running for hours. Furthermore, sampling-based methods may still require manual setting of some program inputs (e.g., DP-Finder [30] requires additional arguments to be set manually for Sparse Vector Technique in our evaluation) while CheckDP is fully automated. Furthermore, the integrated approach of CheckDP allows it to efficiently analyze a larger class of differentially privacy mechanisms, compared with concurrent work using static analyses [42, 43].

Meanwhile, CheckDP still offers state-of-the-art verification capability compared with existing
language-based verifiers and is further able to automatically generate proofs for 3 mechanisms for which no formal verification was reported before. CheckDP takes the source code of a mechanism along with its claimed level of privacy and either generates a proof of correctness or a verifiable counterexample (a pair of related inputs and a feasible output). CheckDP is built upon a proof technique called *randomness alignment* [14, 15, 44], which recasts the task of proving differential privacy into one of finding *alignments* between random variables used by two related runs of the mechanism. CheckDP uses a novel verify-invalidate loop that alternatively improves tentative proofs (in the form of alignments), which are then used to improve tentative counterexamples (and vice versa) until either the tentative proof has no counterexample, or the tentative counterexample has no alignment. We evaluated CheckDP on correct/incorrect versions of existing benchmarks and newly proposed mechanisms. It generated a proof for each correct mechanism within 70 seconds and a counterexample for each incorrect mechanism within 15 seconds.

In summary, the following contributions are listed in this chapter:

1. CheckDP, one of the first automated tools (with concurrent work [42, 43]) that generates both proofs for correct mechanisms and counterexamples for incorrect mechanisms (Section 5.4).

2. A syntax-directed translation from the probabilistic mechanism being checked to non-probabilistic target code with explicit proof obligations (Section 5.5).

3. An alignment template generation algorithm (Section 5.5.6).

4. A novel verify-invalidate loop that incrementally improves tentative proofs and counterexamples (Section 5.6).

5. Case studies and experimental comparisons between CheckDP and existing tools using correct/incorrect versions of existing benchmarks and newly proposed mechanisms. For incorrect mechanisms, CheckDP automatically found counterexamples in all cases, even in cases where competing methods [29, 30] failed. For correct mechanisms, CheckDP automatically generated proofs of privacy, including proofs for 3 mechanisms for which no formal verification was reported before (Section 5.7).
5.2 Background

5.2.1 Privacy Proof and Counterexample

Not all randomness alignments serve as proofs of differential privacy. To form a proof, one must show that (1) the alignment forces the two related executions to produce the same output, (2) the privacy cost of an alignment must be bounded by the promised level of privacy, and (3) the alignment is injective. Hence, in this chapter, an (alignment-based) privacy proof refers to a randomness alignment that satisfies these requirements.

On the other hand, to show that an algorithm violates differential privacy, it suffices to demonstrate the existence of a counterexample. Formally, if an algorithm $M$ claims to satisfy $\epsilon$-differential privacy, a counterexample to this claim is a triple $(inp, inp', o)$ such that $inp \sim inp'$ and $P[M(inp) = o] > \epsilon e^\epsilon P[M(inp') = o]$.

5.2.2 Challenges

LightDP [14] and ShadowDP (Chapter 4) can check if a manually generated alignment is an alignment-based privacy proof. On the other hand, an exact symbolic probabilistic solver, such as PSI [57], can check if a counterexample, either generated manually or via a sampling-based generator, witnesses violation of differential privacy. To the best of our knowledge, CheckDP is the first tool that automatically generates alignment-based proofs/counterexamples via static program analysis.¹ To do so, a key challenge is to tackle the infinite search space of proofs (i.e., alignments) and counterexamples. CheckDP uses a novel proof template generation algorithm to reduce the search space of candidate alignments and uses a novel verify-invalidate loop to find tentative proofs, counterexamples showing their privacy cost is too high, improved proofs, improved counterexamples, etc.

5.3 Motivating Examples

To illustrate our approach, we now discuss two variants of the Sparse Vector Technique [28], one correct and one incorrect. Using the two variants, we sketch how CheckDP automatically proves/disproves (as appropriate) their claimed privacy properties.

Sparse Vector Technique (SVT) [28]  A powerful mechanism proven to satisfy differential privacy. It can be used as a building block for many advanced differentially private algorithms. This mechanism is designed to solve the following problem: given a series of queries and a preset public threshold, we want to identify the first \( N \) queries whose answers are above the threshold, but in a privacy-preserving manner. To achieve this, it adds independent Laplace noise both to the threshold and each query answer, then it returns the identities of the first \( N \) queries whose noisy answers are above the noisy threshold. The standard implementation of SVT outputs \textit{true} for the above-threshold queries and \textit{false} for the others (and terminates when there are a total of \( N \) outputs equal to \textit{true}). We use two variants of SVT for an overview of CheckDP.

\textbf{GapSVT}  This is an improved (and correct) variant of SVT which provides numerical information about some queries. When a noisy query exceeds the noisy threshold, it outputs the difference between these noisy values; otherwise it returns \textit{false}. This provides an estimate for how much higher a query is compared to the threshold. The algorithm was first proposed and verified in [15]; its pseudo code is shown in Figure 5-1. Here, \( \text{Lap} \left( 2/\epsilon \right) \) draws one sample from Laplace distribution with mean 0 and scale factor of \( 2/\epsilon \). This random value is then added to the public threshold \( T \) (stored as noisy threshold \( T_\ast \)). For each query answer, another independent Laplace noise \( \eta_2 = \text{Lap} \left( 4N/\epsilon \right) \) is added. If the noisy query answer \( q[i] + \eta_2 \) is above the noisy threshold \( T_\ast \), the gap between them \( (q[i] + \eta_2 - T_\ast) \) is added to the output list \( \text{out} \), otherwise 0 is added.

One key observation from the manual proofs of SVT and its variants [28, 12, 13, 44] is that the privacy cost is only paid for the queries whose noisy answers are above the noisy threshold. In other words, outputting \textit{false} does not incur any new privacy cost. Correspondingly, the correct alignment for GapSVT [15, 44] (that is, the distance that \( \eta_1 \) and \( \eta_2 \) need to be moved to ensure the output is the same when the input changes from \( q[i] \) to \( q'[i] \equiv q[i] + \tilde{q}[i] \), for all \( i \)) is: \( \eta_1 : 1 \) and...
function $\text{GapSVT}(T,N,\text{size},q,\text{list num})$
returns (out: list num, bound(\epsilon))
precondition $\forall i. \ -1 \leq (\bar{q}[i]) \leq 1$

1. $\eta_1 := \text{Lap}(2/\epsilon)$
2. $T_* := T + \eta_1$
3. count := 0; i := 0;
4. while (count < N \land i < size)
5.   $\eta_2 := \text{Lap}(4N/\epsilon)$
6.   if (q[i] + $\eta_2 \geq T_*$) then
7.       out := (q[i] + $\eta_2 - T_*$):out;
8.       count := count + 1;
9.   else
10.       out := false::out;
11.       i := i + 1;

function $\text{Transformed GapSVT}(T,N,\text{size},q,\bar{q},\text{sample},\theta)$
returns (out)

12. $v_\epsilon := 0; \ idx = 0;$
13. $\eta_1 := \text{sample}[\idx]; \ idx := idx + 1;$
14. $v_\epsilon := v_\epsilon + |A_1| \times \epsilon/2; \ \eta_1 := A_1;$
15. $T_* := T + \eta_1$
16. $\hat{T}_* := \hat{\eta}_1$
17. count := 0; i := 0;
18. while (count < N \land i < size)
19.   $\eta_2 := \text{sample}[\idx]; \ idx := idx + 1;$
20.   $v_\epsilon := v_\epsilon + |A_2| \times \epsilon/4N; \ \eta_2 := A_2;$
21.   if (q[i] + $\eta_2 \geq T_*$) then
22.     assert (q[i] + $\eta_2 + \bar{q}[i] + \hat{\eta}_2 \geq T_* + \hat{T}_*$);
23.     assert (\bar{q}[i] + $\eta_2 - \hat{T}_* == 0$);
24.     out := (q[i] + $\eta_2 - T_*$):out;
25.     count := count + 1;
26.   else
27.     assert ($- (q[i] + \eta_2 + \bar{q}[i] + \hat{\eta}_2 \geq T_* + \hat{T}_*)$);
28.     out := false::out;
29.     i := i + 1;
30. assert ($v_\epsilon \leq \epsilon$);

Figure 5-1: GapSVT and its transformed code, where underlined parts are added by CheckDP. The transformed code contains two alignment templates for $\eta_1$ and $\eta_2$: $A_1 = \theta[0]$ and $A_2 = (q[i] + \eta_2 \geq T_*) \Rightarrow (\theta[1] + \theta[2] \times \hat{T}_* + \theta[3] \times \bar{q}[i]) : (\theta[4] + \theta[5] \times T_* + \theta[6] \times \bar{q}[i])$. The random variables and $\theta$ are inserted as part of the function input.
\( \eta_2 : q[i] + \eta_2 \geq T_\star \) ? (1 - \( q[i] \)) : 0.

Note that \( \eta_2 \) is aligned with non-zero distance only under the true branch; hence, no privacy cost is paid in the other branch. It is easy to verify that if every query has sensitivity 1, the cost of this alignment is bounded by \( \epsilon \).

**BadSVT4** We also consider a variant of SVT (and GapSVT) that incorrectly tries to release numerical information. When a noisy query answer is larger than the noisy threshold, the algorithm releases that noisy query answer (that is, it does not subtract from it the noisy threshold); otherwise it outputs false. This is an incorrect variant of SVT [58] that was reported in [12] and was called iSVT4 in [29]. More precisely, BadSVT4 replaces line 7 of GapSVT with the following: out := (\( q[i] + \eta_2 \))::out;.

This small change makes it not \( \epsilon \)-differentially private [12]. The reason why is subtle, but the intuition is the following. Suppose BadSVT4 returns a noisy query answer \( q[i] + \eta_2 = 3 \), the attacker is able to deduce that \( T_\star \leq 3 \). Once this information is leaked, outputting false in the else branch is no longer “free”; every output incurs a privacy cost.

### 5.4 Approach Overview

We use GapSVT and BadSVT4 to illustrate how CheckDP generates proofs and counterexamples.

**Code Transformation** CheckDP first takes the probabilistic algorithm being checked, written in the CheckDP language, and generates the non-probabilistic target code with assertions and alignment templates (i.e. templates for possible alignments). The bottom of Figure 5-1 shows the transformed code of GapSVT with alignment templates. The transformed code is distinguished from the source code in a few important ways: (1) The probabilistic sampling commands (at lines 1 and 5) are replaced by non-probabilistic counterparts that read samples from the instrumented function input sample. (2) An alignment template (e.g., \( \mathcal{A}_1, \mathcal{A}_2 \)) is generated for each sampling command; each template contains a few holes, i.e., \( \theta \), which is also instrumented as function input. (3) A distinguished variable \( v_\epsilon \) is added to track the overall privacy cost and lines 14 and 20 update the cost variable in a sound way. (4) Assertions are inserted in the transformed code (lines 22,23,27,30) to ensure the following soundness...
property: if $M(inp)$ is transformed to $M'(inp, \widehat{inp}, sample, \theta)$, then

$$\exists \theta. \forall inp, \widehat{inp}, sample. \text{ all assertions in } M' \text{ pass} \implies M \text{ is differentially private}$$

We note that the transformed code forms the basis for both proof and counterexample generation in CheckDP.

**Proof / Counterexample Generation**  Inspired by the Counterexample Guided Inductive Synthesis (CEGIS) [59] technique, originally proposed for program synthesis, CheckDP uses a verify-invalidate loop to simultaneously generate proofs and counterexamples. Unlike CEGIS, however, the verify-invalidate loop is *bidirectional*, in the sense that it internally records all previous counterexamples (resp. proofs) to generate one proof (resp. counterexample) as the algorithm output. On the other hand, the CEGIS loop is *unidirectional*: it only collects and uses a set of inputs to guide synthesis internally (note that counterexample in our context means *one single* input where no alignment exists, rather than the set of inputs internally maintained in the CEGIS loop).

At a high level, the verify-invalidate loop of CheckDP includes two integrated sub-loops, one for proof generation and the other for counterexample generation.

**Verify Sub-loop**  Its goal is to generate a proof (i.e., an instantiation of $\theta$) such that

$$\forall inp, \widehat{inp}, sample. \text{ all assertions in } M' \text{ pass}$$

This is done by two iterative phases:

1. Generating invalidating inputs: Given a proof candidate (i.e., an instantiation of $\theta$), it is *incorrect* if

$$\exists inp, \widehat{inp}, sample. \text{ some assertion in } M' \text{ fail}$$

We use $I$ to denote a triple of $inp, \widehat{inp}, sample$. Hence, given any instantiation of $\theta$, we use an off-the-shelf symbolic execution tool such as KLEE [60] to find invalidating inputs when possible.

2. Generating proof candidates: with a set of invalidating inputs found so far $I_1, \ldots, I_i$, we can try
to generate a new proof candidate to satisfy $\exists \theta. M'(I_1, \theta) \land \cdots \land M'(I_i, \theta)$

Starting from a default instantiation (e.g., one that sets $\forall i. \theta[i] = 0$), CheckDP iteratively repeats phases 1 and 2. Since CheckDP uses all invalidating inputs found so far in Phase 2, the proof candidate after each iteration is improving.

When phase 1 gets stuck, CheckDP obtains a proof candidate $\theta$ which is a privacy proof if

$$\forall \text{inp}, \text{inp}, \text{sample}. M'(\text{inp}, \text{inp}, \text{sample}, \theta)$$

due to the soundness property above. Hence, a proof (alignment) can be validated by program verification tools such as CPAchecker [47]. For GapSVT, CheckDP generates and verifies (via CPAchecker) that $\theta = \{1, 1, 0, -1, 0, 0, 0\}$ results in a proof that GapSVT satisfies $\epsilon$-differential privacy.

**Invalidate Sub-loop** While the verify sub-loop is conceptually similar to a CEGIS loop [59], CheckDP also employs an invalidate sub-loop (integrated with the verify sub-loop); its goal is to generate one invalidating input $I$ such that

$$\forall \theta. \text{ some assertion in } M' \text{ fail}$$

This is done by two iterative phases:

1. Generating proof candidates: Given an invalidating input $I$, it is incorrect if $\exists \theta. M'(I, \theta)$. Hence, given any $I$, we can use KLEE [60] to find an alignment when possible.

2. Generating counterexamples: with a set of previously found alignments $\theta_1, \cdots, \theta_i$, we try to find a new invalidating input to satisfy $\exists I. \neg M'(I, \theta_1) \land \cdots \land \neg M'(I, \theta_i)$.

To integrate with the verify sub-loop, Phase 1 of the invalidate sub-loop starts when Phase 2 of the verify sub-loop gets stuck with a set of invalidating inputs $I_1, \cdots, I_i$; it uses $I_i$ to proceed since it is the most promising one. When Phase 1 of invalidate sub-loop gets stuck, CheckDP obtains a counterexample candidate, which can be validated by PSI [57] (this is necessary since a mechanism might be differentially private even if no alignment-based proof exists).

---

Note that a set of invalidating inputs $I_1, \cdots, I_i$, generated from Phase 2 of the verify sub-loop is not a counterexample candidate, since by definition, a differential privacy counterexample consists of only one invalidating input.
For example, the counterexample found for BadSVT4 sets the threshold $T = 0, N = 1$ (max number of outputs equal to true before termination), neighboring inputs $q = [0, 0, 0, 0, 0]$ and $q' = [1, 1, 1, 1, -1]$, and the following output to examine $[0, 0, 0, 0, 1]$. PSI confirms that the probability of this output when $q$ is an input is $\geq e^\epsilon$ times the probability of this output when $q'$ is the input.

When Phase 1 of the invalidate sub-loop generates a new alignment $\theta$, which happens in our empirical study (Section 5.7), Phase 2 follows to generate an “improved” invalidating input, which is then used to start Phase 2 of the validate sub-loop.

5.5 Program Transformation

CheckDP takes a probabilistic program along with an adjacency specification (i.e., how much two adjacent inputs can differ) and the claimed level of differential privacy as inputs. CheckDP uses a lightweight type system that translates the original probabilistic program into a non-probabilistic program with assertions to ensure differential privacy.

The transformed code will be used by the verify-invalidate loop (Section 5.2.1) to either find a proof or a counterexample. The soundness proof for the transformation can be found in the Appendix.

5.5.1 Syntax

The syntax of CheckDP source code is listed in Figure 5-2. Most of the syntax is standard with the following features:

- Real numbers, booleans and their standard operations;
- Ternary expressions $b \ ? \ n_1 : n_2$, it returns $n_1$ when $b$ evaluates to true or $n_2$ otherwise;
- List operations: $e_1 :: e_2$ appends element $e_1$ to list $e_2$, and $e_1[e_2]$ gets the $e_2^{th}$ element of list $e_1$;
- Loop with keyword while and branch with keyword if;
- A final return command return $e$. 
We now introduce other interesting parts that are needed for developing differentially private algorithms.

**Random Expressions**  Differential privacy relies heavily on probabilistic computations: many mechanisms achieve differential privacy by adding appropriate random noise to variables. To model this behavior, we embed a sampling command $\eta ::= \text{Lap } r$ in CheckDP, which draws a sample from the Laplace distribution with mean 0 and scale of $r$. In this chapter, we only focus on the most interesting sampling command Lap $r$ (which is used in Laplace Mechanism and GapSVT in Section 5.2). However, we note that it is fairly easy to add new sampling distributions to CheckDP.

For clarity, we distinguish variables holding random values, denoted by $\eta \in H$, from other ones, denoted by $x \in V$.

**Types with Distances**  To enable alignment-based proof, one important aspect of the type system in CheckDP is the ability to compute and track the distances for each program variable. Motivated by verification tools using alignments (e.g., LightDP [14] and ShadowDP [15]), types in the source language of CheckDP have the form of $B_0$ or $B_\tau$, where $B$ is the base type such as numerics ($\text{num}$), booleans ($\text{bool}$) and lists ($\text{list } \tau$). The subscript of each type is the key to alignment-based proofs: it explicitly

---

**Figure 5-2**: CheckDP: language syntax.

<table>
<thead>
<tr>
<th>Reals</th>
<th>$r \in \mathbb{R}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booleans</td>
<td>$b \in {\text{true, false}}$</td>
</tr>
<tr>
<td>Vars</td>
<td>$x \in V$</td>
</tr>
<tr>
<td>Rand Vars</td>
<td>$\eta \in H$</td>
</tr>
<tr>
<td>Linear Ops</td>
<td>$\oplus ::= + \</td>
</tr>
<tr>
<td>Other Ops</td>
<td>$\otimes ::= \times \</td>
</tr>
<tr>
<td>Comparators</td>
<td>$\od ::= \langle \rangle</td>
</tr>
<tr>
<td>Bool Exprs</td>
<td>$b ::= \text{true} \</td>
</tr>
<tr>
<td>Num Exprs</td>
<td>$n ::= r \</td>
</tr>
<tr>
<td>Expressions</td>
<td>$e ::= n \</td>
</tr>
<tr>
<td>Commands</td>
<td>$c ::= \text{skip} \</td>
</tr>
<tr>
<td>Rand Exps</td>
<td>$g ::= \text{Lap } r$</td>
</tr>
<tr>
<td>Types</td>
<td>$\tau ::= \text{num}_d \</td>
</tr>
<tr>
<td>Distances</td>
<td>$d ::= 0 \</td>
</tr>
</tbody>
</table>
tracks the difference between the value of a variable in two related runs.

In the source language of CheckDP, the distances can either be 0 or \(*\): the former indicates the variables stay the same in the related runs; the latter means that the variable might hold different values in two related runs and the value difference is stored in a distinguished variable \(\tilde{x}\) added by the program transformation (i.e., a syntactic sugar for dependent sum type \(\sum(x: \text{num}_0) \ B_{\tilde{x}}\)). For example, inputs \(T, N, \text{size}\) are annotated with distance 0 in Figure 5-1, meaning that they are public parameters to the algorithm; query answers \(q\) are annotated with distance \(*\), meaning that each \(q[i]\) differ by exactly \(q[i]\) in two related runs. The type system distinguishes zero-distance variables as an optimization: as we show shortly, it helps to reduce the code size for later stages (Section 5.5.5) as well as aids proof template generation (Section 5.5.6).

Note that boolean types \((\text{bool})\) and list types \((\text{list } \tau)\) cannot be associated with numeric distances, hence omitted in the syntax. However, nested cases such as \(\text{list num}\), still accurately track the distances of the elements inside the list.

Finally, CheckDP also supports shadow execution, a technique that underpins ShadowDP [15] and is crucial to the verification of challenging mechanisms such as Report Noisy Max [34]. However, in order to focus on the most interesting parts of CheckDP, we first present the transformation without shadow execution, and then discuss how to support shadow execution.

### 5.5.2 Semantics

Following the standard definition in [14] and Chapter 4, we define the denotational semantics of the probabilistic language as a mapping from initial memory to a distribution on a (possible) final outputs. More formally, let \(M\) be a set of memory states where each \(m \in M\) maps all variables to their values, including normal variables \((x \in V)\) and random variables \((\eta \in H)\). A program is interpreted as a function \(M \rightarrow \text{Dist}(B)\) where \(B\) is the return type (i.e., the type of \(e\)). Since the semantics is the same as the one in Section 4.4.2, we omit it here.
5.5.3 Program Transformation

CheckDP is equipped with a flow-sensitive type system whose typing rules for expressions and commands are shown in Figure 5-3 and Figure 5-4, respectively. At command level, each rule has the following format:

\[ \vdash \Gamma \{ c \rightarrow c' \} \Gamma' \]

where a typing environment \( \Gamma \) tracks for each program variable its type with distance, \( c \) and \( c' \) are the source and target programs respectively, and the flow-sensitive type system also updates typing environment to \( \Gamma' \) after command \( c \). At a high-level, the type system transforms the probabilistic source code \( c \) into the non-probabilistic target code \( c' \) in a way that if all assertions in \( c' \) holds, then \( c \) is differentially private.

CheckDP’s program transformation is motivated by those of LightDP and ShadowDP [14, 15], all built on randomness alignment proof. However, there are a few important differences:

- CheckDP generates an alignment template for each sampling instruction, rather than requiring them to be manually provided by a programmer.

- CheckDP defers all privacy-related checks to assertions. This is crucial since information needed for proof and counterexample generation is unavailable in a lightweight static type system;

- CheckDP only tracks if a variable has the same value in two related runs (with distance 0) or not (with distance *). This design aids alignment template generation and reduces the size of transformed code.
### Transformation Rules for Expressions with Form $\Gamma \vdash e : \mathcal{B}_{(\tau_1,\tau_2)}$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type (T-NAME)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash r : \text{num}_0</td>
<td>\text{true}$</td>
<td>(T-INTEGER)</td>
</tr>
<tr>
<td>$\Gamma \vdash b : \text{bool}</td>
<td>\text{true}$</td>
<td>(T-BOOLEAN)</td>
</tr>
<tr>
<td>$\Gamma, x : \mathcal{B}_0 \vdash x : \mathcal{B}_0</td>
<td>\text{true}$</td>
<td>(T-REFERENCE)</td>
</tr>
<tr>
<td>$\Gamma \vdash e : \text{bool}</td>
<td>C$</td>
<td>(T-LOGICAL)</td>
</tr>
<tr>
<td>$\Gamma \vdash \neg e : \text{bool}</td>
<td>C$</td>
<td>(T-LOGICAL)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : \text{num}_{\tau_1}</td>
<td>C_1 \quad \Gamma \vdash e_2 : \text{num}_{\tau_2}</td>
<td>C_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 \oplus e_2 : \text{num}_{\tau_1}</td>
<td>C_1 \quad \Gamma \vdash e_2 : \text{num}_{\tau_2}</td>
<td>C_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 \odot e_2 : \text{bool}</td>
<td>C_1 \quad \Gamma \vdash e_1 \odot e_2 : \text{num}_{\tau_1}</td>
<td>C_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : \mathcal{B}_{\tau_1}</td>
<td>C_1 \quad \Gamma \vdash e_1 : \text{list} \mathcal{B}_{\tau_2}</td>
<td>C_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : \text{list} \tau</td>
<td>C_1 \quad \Gamma \vdash e_2 : \text{num}_n</td>
<td>C_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 \llbracket e_2 \rrbracket : \tau</td>
<td>C_1 \quad \Gamma \vdash e_3 : \mathcal{B}_{\tau_1}</td>
<td>C_2 \quad \Gamma \vdash e_3 : \mathcal{B}_{\tau_2}</td>
</tr>
</tbody>
</table>

### 5.5.4 Checking Expressions

Each typing rule for expression $e$ computes the correct distance for its resulting value:

$$\Gamma \vdash e : \mathcal{B}_n | C$$

which reads as: expression $e$ has type $\mathcal{B}$ and distance $n$ under the typing environment $\Gamma$ if the constraints $C$ are satisfied. The reason to collect constraints $C$ instead of statically checking them, is to defer all privacy-related checks to later stages.

Most of the expression rules are straightforward: they check the base types (just like a traditional type system) and compute the distance of $e$’s value in two related runs. For example, all constants
Transformation Rules for Commands with Form \( \vdash \Gamma \{ c \rightarrow c' \} \Gamma' \)

\[
\begin{align*}
\Gamma &\vdash e : B_\mathcal{O} \mid C \quad \langle d, c \rangle = \begin{cases} 
(0, \text{skip}), & \text{if } n = 0, \\
(*, \vec{x} := \vec{y}), & \text{otherwise}
\end{cases} \\
\vdash \Gamma \{ x := e ; \text{assert} (C); x := e ; c \} \Gamma'[x \mapsto B_\mathcal{O}] &\quad \text{(T-ASGN)} \\
\vdash \Gamma \{ c_1 \rightarrow c'_1 \} \Gamma_1 &\quad \vdash \Gamma_1 \{ c_2 \rightarrow c'_2 \} \Gamma_2 &\quad \vdash \Gamma \{ c_1 ; c_2 \rightarrow c'_1 ; c'_2 \} \Gamma_2 &\quad \text{(T-SEQ)} \\
\vdash \Gamma \{ \text{skip} \rightarrow \text{skip} \} \Gamma &\quad \text{(T-SKIPL)} \\
\vdash \Gamma \{ \text{return} e \rightarrow \text{assert} (C \land n = 0); \text{return} e \} \Gamma &\quad \text{(T-RETURN)} \\
\vdash \Gamma \{ \text{while } e \text{ do } c \rightarrow c_s ; (\text{while } e \text{ do } (\text{assert} (C); c'; c'')) \} \Gamma \sqcup \Gamma_f &\quad \text{(T-WHILE)} \\
\vdash \Gamma \{ \text{if } e \text{ then } c_1 \text{ else } c_2 \rightarrow \text{if } e \text{ then } (\text{assert} (C); c'_1; c'_1'') \text{ else } (\text{assert} (C); c'_2; c'_2'') \} \Gamma_1 \sqcup \Gamma_2 &\quad \text{(T-If)} \\
\end{align*}
\]

Transformation Rules for Merging Environments

\[
\begin{align*}
\Gamma_1 \sqsubseteq \Gamma_2 \quad c = \{ \vec{x} := 0 \mid \Gamma_1(x) = \text{num}_0 \land \Gamma_2(x) = \text{num}_2 \} \\
\Gamma_1, \Gamma_2 \Rightarrow c
\end{align*}
\]

Figure 5-4: Program transformation rules for commands and auxiliary rules. Distinguished variable \( v_e \) and assertions are added to ensure differential privacy.

must be identical (Rules (T-Num,T-Boolean)) and the distance of a variable is retrieved from the environment (T-VarZero,T-VarStar) (note that rule (T-VarStar) just desugers the \( * \) notation). For linear operation (\( \oplus \)), the distance of the result is computed in a precise way (Rule (T-OPlus)), while the other operations are treated in a more conservative way: constraints are generated to ensure that the result is identical in Rules (T-OTimes, T-ODot). For example, (T-ODot) ensures boolean value of \( e_1 \odot e_2 \) will be the same in two related runs by adding a constraint:

\[(e_1 \odot e_2) \Rightarrow (e_1 + n_1) \odot (e_2 + n_2)\]

(T-Cons) restricts constructed list elements to have 0-distance (note that the restriction does not
apply to input lists), while (T-INDEX) requires the index to have zero-distance. Rule (T-SELECT) restricts $e_1$ and $e_2$ to have the same distance.

The constraints gathered in the expression rules will later be explicitly instrumented as assertions in the translated programs, which we will explain shortly.

### 5.5.5 Checking Commands

For each program statement, the type system updates the typing environment and if necessary, instruments code to update $\hat{x}$ variables to the correct distances. Moreover, it ensures that the two related runs take the same branch in if-statement and while-statement.

**Flow-Sensitivity**  Each typing rule updates the typing environment to track if a variable has zero-distance. When a variable has non-zero distance, it instruments the source code to properly maintain the corresponding $\hat{x}$ variables. The most interesting rules are: rule (T-Asgn) properly promotes the type of $x$ to be $B_\ast$ (tracked by distance variables) in $\Gamma'$ if the distance of $e$ is not 0. Rule (T-AsgnZero) is similar, except that it optimizes away updates to $\hat{x}$ and properly downgrades type to $B_0$ if $e$ has zero-distance.

For example, line 16 in GapSVT (Figure 5-1) is instrumented to update distance of $T_\ast$, according to the distance of $T + \eta_1$. Moreover, variable count in GapSVT always has the type $\text{num}_0$; therefore its distance variable never appears in the translated program due to the optimization in (T-AsgnZero).

Rule (T-If) and (T-While) are more complicated since they both need to merge environments. In rule (T-If), as $c_1$ and $c_2$ might update $\Gamma$ to $\Gamma_1$ and $\Gamma_2$ respectively, we need to merge them in a natural way: the distance of a type form a two-level lattice with $0 \sqsubseteq \ast$. Thus we define a union operator $\sqcup$ for distances $d$ as:

$$
\begin{align*}
d_1 \sqcup d_2 & \triangleq \\
& \begin{cases} 
  d_1 & \text{if } d_1 = d_2 \\
  \ast & \text{otherwise}
\end{cases}
\end{align*}
$$

therefore the union operator for two environments are defined as follows: $\Gamma_1 \sqcup \Gamma_2 = \lambda x. \Gamma_1[x] \sqcup \Gamma_2[x]$.

Moreover, we use an auxiliary function $\Gamma_1, \Gamma_2 \Rightarrow c$ to “promote” a variable to star type: an assignment $\hat{x} := 0$ is instrumented when $\Gamma_1(x) = 0$ and $\Gamma_2(x) = \ast$. 
For example, with $\Gamma(x) = \ast$, $\Gamma(y) = \ast$ and $\Gamma(b) = 0$, rule (T-If) translates the source code

$\textbf{if}\ b\ \textbf{then}\ x := y\ \textbf{else}\ x := 1$

to the following: $\textbf{if}\ b\ \textbf{then}\ (x := y; \bar{x} := \bar{y};)\ \textbf{else}\ (x := 1; \bar{x} := 0)$

where $\bar{x} := \bar{y}$ is instrumented by (T-Asgn) and $\bar{x} := 0$ is instrumented due to the promotion.

Similarly, the typing environments are merged in rule (T-While), except that it requires a fixed point $\Gamma_f$ such that $\vdash \Gamma \sqcup \Gamma_f \{e\} \Gamma_f$. We follow the construction in [15], noting that the computation always terminates since all of the translation rules are monotonic and the lattice only has two levels.

**Assertion Generation**

To ensure differential privacy, the type system inserts assertion in various rules:

- To ensure that two related runs take the same control flow, (T-If) and (T-While) asserts constraints gathered from making sure that the boolean has zero-distance (e.g., from (T-ODot)). As an optimization, we use the branch condition to simplify constraints when possible. For example, constraints $e_1 \odot e_2 \iff (e_1 + n_1) \odot (e_2 + n_2)$ are simplified as $(e_1 + n_1) \odot (e_2 + n_2)$ (resp. $\neg((e_1 + n_1) \odot (e_2 + n_2))$) in the true (resp. false) branch.

- To ensure that the final output value is differentially private, rule (T-Return) asserts that its distance is zero (i.e., identical in two related runs).

- To ensure all constraints collected in the expression rules are satisfied, assignment rules (T-Asgn) and (T-AsgnStar) also insert corresponding assertions.

### 5.5.6 Checking Sampling Commands

Rule (T-Laplace) performs a few important tasks:

**Generating Non-probabilistic Target Program**

The rule (T-Laplace) removes the sampling instruction and assign to $\eta$ the next (unknown) sample value $\text{sample}[\text{id}_x]$, where $\text{sample}$ is a parameter of type list num added to the transformed code. The typing rule also increments $\text{id}_x$ so that the next sampling command will read out the next value.

**Checking Injectivity**

T-Laplace adds an assertion $c_a$ to check the injectivity of the generated alignment (a fundamental requirement of alignment-based proofs): the same aligned value of $\eta$ implies the same
value of $\eta$ in the original execution.

**Tracking Privacy Cost** A distinguished privacy cost variable $v_\epsilon$ is also instrumented to track the cost for aligning the random variables in the program. Due to the properties of Laplace distribution, for a sampling command $\eta := \text{Lap} r$ with alignment template $A$, we have $\frac{\mathbb{P}(\eta)}{\mathbb{P}(\eta + A)} \leq e^{|A|/r}$.

Hence, the privacy cost for aligning $\eta$ by $A$ is $|A|/r$. Note that the symbols in gray, including $A$, are placeholders when the rule is applied, since function `GenerateTemplate` takes all assertions in the transformed code as inputs. Once translation is complete, the placeholders are filled in by the algorithm that we discuss next.

**Alignment Template Generation** For each sampling command $\eta := \text{Lap} r$, an alignment of $\eta$ is needed in a randomness alignment proof. In its most flexible form, the alignment can be written as any numerical expression $v$, which is prohibitive for our goal of automatic proof generation. On the other hand, using simple heuristics such as only considering constant alignment does not work: for example, the correct alignment for $\eta_2$ in GapSVT is written as “$(q[1] + \eta_2 \geq T_\star) \wedge (1 - \text{q}[1]) : 0$”, where the alignment actually depends on which branch is taken during the execution.

To tackle the challenges, CheckDP generates an alignment template for each sampling instruction; a template is a numerical expression with “holes” whose values are to be searched for in later stages. For example, the template generated for $\eta_2$ in GapSVT is

$$(q[1] + \eta_2 \geq T_\star) \wedge (\theta[0] + \theta[1] \times \text{q}[1], \theta[2] \times \text{q}[1]) : (\theta[3] + \theta[4] \times \text{q}[1])$$

where $\theta[0] - \theta[5]$ are symbolic coefficients to be found later.

In general, for each sampling command $\eta = \text{Lap} r$, CheckDP first uses static program analysis to find a set of relevant program expressions, denoted by $\mathbb{E}$, and a set of relevant program variables, denoted by $\mathbb{V}$ (as described shortly). Second, it generates an alignment template as follows:

$$A_{\mathbb{E}} := \begin{cases} e_0 ? A_{\mathbb{E}\backslash\{e_0\}} : A_{\mathbb{E}\backslash\{e_0\}} & \text{if } \mathbb{E} = \{e_0, \cdots\} \\ \theta_0 + \sum_{v_i \in \mathbb{V}} \theta_i \times v_i \text{ with fresh } \theta_0, \cdots, \theta_{|\mathbb{V}|} & \text{otherwise} \end{cases}$$
where $\theta$ denotes coefficients ("holes") to be filled out by later stages and each of them is generated fresh.

To find proper $E$ and $V$, our insight is that the alignments serve to "cancel out" the differences between two related runs (i.e., to make all assertions pass). Algorithm 1 follows the insight to compute $E$ and $V$ for each sampling instruction: it takes $\Gamma_s$, the typing environment right before the sampling instruction and $A$, all assertions in the transformed code, as inputs. It also assumes an oracle $\text{Depends}(e, x)$ which returns true whenever the expression $e$ depends on the variable $x$. We note that the oracle can be implemented as standard program dependency analysis [61, 62] or information flow analysis [63]; hence, we omit the details in this dissertation.

**Algorithm 1:** Proof template generation for sampling command $\eta := \text{Lap } r$.

```
input : $\Gamma_s$: typing environment at sampling command
       $A$: set of the generated assertions in the program

1 function GenerateTemplate($\Gamma_s, A$):
  2 $E \leftarrow \emptyset, V \leftarrow \emptyset$
  3 foreach $\text{assert} (e) \in A$ do
    4 if $\text{Depends}(e, \eta)$ then
      5 if $\text{assert}(e)$ is generated by (T-I/f.pc) then
        6 $e' \leftarrow$ the branch condition of $\text{if}$
        7 $E \leftarrow E \cup \{e'\}$
      8 foreach $v \in \text{Vars} \cup \{e_1[e_2]|e_1[e_2] \in e\}$ do
        9 if $\Gamma_s \not\ni v : B_0 \land \text{Depends}(e, v)$ then
          10 $V \leftarrow V \cup \{v\}$
    11 foreach $e \in E \cup V$ do
      12 remove $e$ from $E$ and $V$ if not in scope
  13 return $E, V$;
```

This template generation algorithm first checks (at line 4) if aligning $\eta$ has a chance to make an assertion pass. If so, it will increment $E$ and $V$ as follows. For $E$, we notice that only for the assertions generated by rule (T-Ir), depending on the branch condition allows the alignment to have different values under different branches. Hence, we add the branch condition to $E$ in this case.

For $V$, our goal is to use the alignment to “cancel” the differences caused by other variables and array elements such as $q[i]$ used in $e$. Hence, we only need to consider $\widehat{v}$ if (1) $v$ is different between two related runs (i.e., $\Gamma_s \not\ni v : B_0$) and (2) $v$ contributes the assertion (i.e., $e$ depends on $v$).
Finally, the algorithm performs a “scope check”: if any element in $E$ or $V$ contains out-of-scope variables, then the element is excluded; for example, $\eta_1$ should not depend on $q[i]$ in GapSVT since $q[i]$, essentially an iterator of $q$, is not in scope at that point.

Take the alignment template generation for $\eta_1$ and $\eta_2$ in GapSVT for example. The assertions in the translated programs are

(we only list the assertion in the true branch since the constraint in false branch is symmetric):

1. **assert** $(q[i] + \eta_2 + \hat{q}[i] + \hat{h}_2 \geq T_\star + \hat{T}_\star)$

2. **assert** $(\hat{q}[i] + \hat{h}_2 - \hat{T}_\star = 0)$

For $\eta_1$, we have $\Gamma_s = \{q : *\}$ (we omit the base types and the variables that have 0 distance for brevity) and both assertions depend on $\eta_1$. Since both assertions depend on $\eta_1$ and $q[i]$, Algorithm 1 adds $\hat{q}[i]$ into $V$. Moreover, assertion 1 is generated by rule (T-Ir).

Thus, the algorithm adds $q[i] + \eta_2 \geq T_\star$ into $E$. Finally, since $q[i]$ is out of scope at the sampling instruction, expression using $q[i]$ and variable $q[i]$ are excluded, resulting $V = \{\}$ and $E = \{\}$.

For $\eta_2$, we have $\Gamma_s = \{q: *, T_\star : *\}$. Since both assertions depend on $\eta_2$ and $q[i]$ and $T$, Algorithm 1 adds $\hat{q}[i]$ and $\hat{T}_\star$ into $V$. Similar to $\eta_1$, the algorithm also adds $q[i] + \eta_2 \geq T_\star$ into $E$. Finally, all expressions and variable are in scope, resulting $V = \{\hat{q}[i], \hat{T}_\star\}$ and $E = \{q[i] + \eta_2 \geq T_\star\}$.

5.5.7 Function Signature Rewrite

The final stage of the program transformation is to rewrite function signature to reflect the extra inputs used in the transformed code. In general, $M(inp)$ is transformed to a new function signature $M'(inp, \hat{inp}, sample, \theta)$ where $\hat{inp}$ are the distance variables associated with inputs whose distance is not zero (e.g., $\hat{q}$ is associated with $q$ in GapSVT), $sample$ is a list of random values used in $M$, and $\theta$ are the missing holes in alignment templates.
5.5.8 Shadow Execution

To tackle challenging mechanisms such as Report Noisy Max [34], shadow execution is introduced (Chapter 4). Intuitively, the shadow execution tracks another program execution where the injected noises are always the same as those in the original execution. Therefore, values computed in the shadow execution incur no privacy cost. The aligned execution can then switch to shadow execution when certain conditions are met, so that privacy cost can be reset to zero.

Supporting shadow execution in CheckDP only requires a few modifications to the program transformation component:

1. Expressions will have a pair of distances $(h_d - d_y)$, where $d^o$ tracks the distance in the aligned execution and $d^\uparrow$ in the shadow execution;

2. Since the branches and loop conditions in shadow execution are not aligned, they might diverge from the original execution. Hence, a separate shadow branch/loop is generated to correctly update the shadow distances for the variables.

At a high level, the extension encodes the selectors (which requires manual annotations in ShadowDP) and integrates them with the generated templates. With the extra “holes” in the templates, the verify-invalidates loop will automatically find alignments (including selectors) / counterexamples. The complete set of transformation rules with shadow execution is shown in Figure 5-5 and Figure 5-6 for expressions and commands, respectively. The differences that shadow execution introduces are highlighted in gray.

**Expressions** Since shadow execution is tracked, types for each variable would be extended to include a pair of distances $(d^o, d^\uparrow)$. More specifically, the extended types should be defined as: $	au ::= \text{num}(d^o, d^\uparrow) \mid \text{bool} \mid \text{list } \tau$.

With the modified types, corresponding modifications to the transformation rules for expressions are straightforward and minimal: the handling of shadow distances are essentially the same as that of aligned distances.

**Normal Commands** Following the type system of ShadowDP, a program counter $pc \in \{\top, \bot\}$ is introduced to each transformation rule for commands to capture potential divergence of shadow execution.
\[
\begin{align*}
\Gamma &\vdash r : \text{num}(0,0) \mid \text{true} \quad \text{(T-NUM)} & \Gamma &\vdash b : \text{bool} \mid \text{true} \quad \text{(T-BOOLEAN)} \\
\Gamma &\vdash e : \text{bool} \mid C \quad \text{(T-NEG)} & \Gamma &\vdash \neg e : \text{bool} \mid C \\
\Gamma &\vdash e_1 : \text{num}(p_1, p_2) \mid C_1 \quad \Gamma &\vdash e_2 : \text{num}(p_3, p_4) \mid C_2 \quad \Gamma &\vdash e_1 \oplus e_2 : \text{num}(p_1, p_2 + p_3, p_4) \mid C_1 \land C_2 \quad \text{(T-OPlus)} \\
\Gamma &\vdash e_1 : \text{num}(p_1, p_2) \mid C_1 \quad \Gamma &\vdash e_2 : \text{num}(p_3, p_4) \mid C_2 \quad \Gamma &\vdash e_1 \otimes e_2 : \text{num}(0, 0) \mid C_1 \land C_2 \land (p_1 = p_2 = p_3 = p_4 = 0) \quad \text{(T-OTimes)} \\
\Gamma &\vdash e_1 \oplus e_2 : \text{bool} \mid C_1 \land C_2 \land (e_1 \oplus e_2) \Leftrightarrow (e_1 + n_1) \oplus (e_2 + n_2) \land (e_1 \otimes e_2) \Leftrightarrow (e_1 + n_2) \otimes (e_2 + n_4) \quad \text{(T-ODot)} \\
\Gamma &\vdash e_1 : B_{(p_1, p_2)} \mid C_1 \quad \Gamma &\vdash e_2 : \text{list} B_{(p_3, p_4)} \mid C_2 \quad \Gamma &\vdash e_1 \circ e_2 : \text{list} B_{(p_3, p_4)} \mid C_1 \land C_2 \land (p_1 = p_2 = p_3 = p_4 = 0) \quad \text{(T-Cons)} \\
\Gamma &\vdash e_1 : \text{list} \tau \mid C_1 \quad \Gamma &\vdash e_2 : \text{num}(p_1, p_2) \mid C_2 \quad \Gamma &\vdash e_1[e_2] : \tau \mid C_1 \land C_2 \land (p_1 = p_2 = 0) \quad \text{(T-Index)} \\
\Gamma &\vdash e_1 : \text{bool} \mid C_1 \quad \Gamma &\vdash e_2 : B_{(p_1, p_2)} \mid C_2 \quad \Gamma &\vdash e_3 : B_{(p_3, p_4)} \mid C_3 \quad \Gamma &\vdash e_1 ? e_2 : e_3 : B_{(p_1, p_2)} \mid C_1 \land C_2 \land C_3 \land (p_1 = p_2 = p_3 = p_4) \quad \text{(T-Select)}
\end{align*}
\]

Figure 5-5: Expressions rules for extending CheckDP with shadow execution. Differences that shadow execution introduce are marked in gray boxes.

Specifically, \( pc \vdash c \rightarrow c' \). \( pc = \top \) (resp. \( \bot \)) means that the branch / loop command might diverge in the shadow execution (resp. must stay the same). The value of \( pc \) is used to guide how each rule should handle the shadow distances (e.g., \( (T-\text{Asgn}) \)), which we will explain shortly. Therefore, another auxiliary function \( \text{updatePC} \) is added to track the value of \( pc \).

Compared with the type system of ShadowDP, the first major difference is in \( (T-\text{Asgn}) \). If \( pc = \bot \), shadow distances are handled as the aligned distances. However, when \( pc = \top \) (shadow execution diverges), it updates the shadow distance of the variable to make sure the value in shadow execution (i.e., \( x + \hat{x}^\dagger \)) remains the same after the assignment. For example, Line 30 in Figure 5-12 is instrumented to maintain the value of bq in the shadow execution (bq + \( \hat{\text{bq}}^\dagger \)), so that the branch at Line 36 is not affected.
Transformation Rules for Commands with Form $pc \vdash \Gamma \{ c \rightarrow c' \} \Gamma'$

$$
\begin{align*}
\text{T-ASGN} & \quad \Gamma \vdash e : B_{[n_0, n_1]} \mid C \quad \Gamma \{ (d', c') \} = \begin{cases} 
(0, \text{skip}), & \text{if } n' = 0; \\
(x, x' = n'), & \text{otherwise}
\end{cases} \\
\text{T-SEQ} & \quad \Gamma \cdot \Gamma \{ c_1 \rightarrow c'_1 \} \Gamma_1 \quad \Gamma \cdot \Gamma \{ c_2 \rightarrow c'_2 \} \Gamma_2 \\
\text{T-SKIP} & \quad \Gamma \vdash \Gamma \{ \text{skip} \rightarrow \text{skip} \} \Gamma \\
\text{T-RETURN} & \quad \Gamma \vdash e : B_{[n_0, n_1]} \mid C \quad \Gamma \{ \text{return} \rightarrow \text{assert} (C \land x^0 = 0); \text{return} \} e \Gamma \\
\text{T-WHILE} & \quad \Gamma \vdash \Gamma \{ \text{while } e \text{ do } c \rightarrow c_s; (\text{while } e \text{ do } \text{assert } ((e, \Gamma) \circ \circ; c'; c'')); \Gamma \} \Gamma \cup \Gamma_f \\
\text{T-IF} & \quad \Gamma \vdash \Gamma \{ c_1 \rightarrow c'_1 \} \Gamma_i \quad \Gamma \vdash \Gamma \{ c_2 \rightarrow c'_2 \} \Gamma_f \quad \Gamma \vdash \Gamma \{ \text{updatePC}(pc, \Gamma, e) \} c_s \\
\text{T-LAPLACE} & \quad \Gamma \vdash \Gamma \{ q = \text{Lap} \rightarrow c_s; \quad q = \text{sample}[dx]; idx = idx + 1; v_x := \{ S \text{?} v_x := 0 \} + | A | r; \eta := A; \text{mpl} \} \Gamma' \quad \text{if } \text{num}(s) \}
\end{align*}
$$

Transformation Rules for Merging Environments

$$
\begin{align*}
\Gamma_1 \sqsubseteq \Gamma_2 \\
\Gamma_1 \vdash c^0 : x^0 := 0 \mid \Gamma_2(x) = \text{num}(0, d_1) \land \Gamma_2(x) = \text{num}(d_2, 0)) \\
c^0' := \begin{cases} x^0 := 0 \mid \Gamma_2(x) = \text{num}(d_1, 0) \lor \Gamma_2(x) = \text{num}(d_2, 0)) \\
\end{cases}
\end{align*}
$$

PC Update Function

$$
\text{updatePC}(pc, \Gamma, e) = \begin{cases} 
\bot, & \text{if } pc = \bot \land \Gamma \vdash e : \text{num}(\cdot, 0)) \\
\top, & \text{else}
\end{cases}
$$

Figure 5-6: Command rules for extending CheckDP with shadow execution. Differences that shadow execution introduce are marked in gray boxes.

by the new assignment of bq.

As previously explained, a separate shadow branch / loop has to be generated to correctly track the shadow distances of the variables. More specifically, Rules (T-IF) and (T-WHILE) is extended to include an extra shadow execution command $c^\uparrow$ when $pc$ transits from $\bot$ to $\top$. The shadow execution is constructed using the same auxiliary function $(c, \Gamma)^\uparrow$ defined from ShadowDP (Chapter 4, Figure 4-7).
It essentially replaces each variable with its correspondence (e.g., variable \(x\) to \(x^\dagger\)), as is standard in self-composition [52, 53]. Note that the value of an expression \(e\) in an aligned execution (i.e., \(\langle e, \Gamma \rangle^0\) used in Rules (T-Ifr) and (T-WHILE)) are defined in a similar way.

**Sampling Commands** The most interesting rule is (T-LAPLACE). In order to enable the automatic discovery of the selectors, our GenerateTemplate algorithm needs to be extended to return a selector template \(S\).

Intuitively, a selector expression \(S\) with the following syntax decides if the aligned or shadow execution is picked:

\[
\text{Var Versions} \quad k \in \{\circ, \dagger\} \\
\text{Selectors} \quad S ::= e ? S_1 : S_2 | k
\]

The definition of the selector template is then similar to the alignment template, where the value can depend on the branch conditions:

\[
S_E ::= \begin{cases} 
  e_0 ? S_{E\setminus\{e_0\}} : S_{E\setminus\{e_0\}}, \text{ when } E = \{e_0, \cdots\} \\
  \theta \text{ with fresh } \theta, \text{ otherwise}
\end{cases}
\]

Compared with other holes (\(\theta\)) in the alignment template (\(\mathcal{A}_E\)), the only difference is that \(\theta\) in \(S_E\) has Boolean values representing whether to stay on aligned execution (\(\circ\)), or switch to shadow execution (\(\dagger\)).

To embed shadow execution into CheckDP, the type system dynamically instruments an auxiliary command \(c_d\) according to the selector template \(S\). Once a switch is made (\(S = \dagger\)), the distances of all variables are replaced with their shadow versions by this command. Moreover, the privacy cost \(v_e\) will also be properly reset according to the selector.

**5.5.9 Soundness**

CheckDP enforces a fundamental property: suppose \(M(inp)\) is transformed to \(M'(inp, \widehat{inp}, sample, \theta)\), then \(M(inp)\) is differentially private if there is a list of values \(\theta\), such that all assertions in \(M'\) hold for all \(inp, \widehat{inp}, sample\). Recall that an alignment template \(\mathcal{A}\) is a function of \(\theta\). Hence, we have a concrete
alignment $\mathcal{A}(\theta)$ for each sampling instruction in the source code given $\theta$. We build the soundness of
CheckDP based on that of ShadowDP (Chapter 4). Their syntax and semantics of source code are almost
identical: the only difference is that ShadowDP requires every sampling command $\eta := \text{Lap } r$ to be
manually annotated with a selector and an alignment for $\eta$. Thus, we can easily rewrite a program $M$ in
CheckDP to a program $\tilde{M}$ in ShadowDP by adding the annotations:

$$\eta := \text{Lap } r \rightarrow \eta := \text{Lap } r; \circ; \mathcal{A}_\eta(\theta) \quad \text{(CheckDP to ShadowDP)}$$

where $\mathcal{A}_\eta$ is the alignment template for $\eta$.

Without loss of generality, we will proceed with the case with shadow execution (i.e., the type system
$\Gamma$ tracks a pair of distances for both aligned and shadow executions), since a proof without shadow
execution is subsumed by the one with shadow execution and a selector that always selects the aligned
distances.

**Theorem 5.1** (Soundness). Let $M$ be a mechanism written in CheckDP and $\mathcal{A}_\eta$ be the alignment
template generated for $\eta$. Let $\tilde{M}$ be the corresponding program in ShadowDP by the rule (CHECKDP TO
SHADOWDP). If (1) CheckDP type checks $M$, i.e., $\vdash \Gamma \{ M \rightarrow M' \} \Gamma'$ and (2) for all inputs to $M'$, the
assertions in $M'$ hold. Then $\tilde{M}$ type checks in ShadowDP, and the assertions in $\tilde{M}'$ in ShadowDP pass.

Let $M$ be a mechanism written in CheckDP. With a list of concrete values of $\theta$, let $\tilde{M}$ be the
corresponding mechanism in ShadowDP by rule (CHECKDP TO SHADOWDP). If (1) $M$ type checks, i.e.,
$\vdash \Gamma \{ M \rightarrow M' \} \Gamma'$ and (2) the assertions in $M'$ hold for all inputs. Then

1. $\tilde{M}$ type checks in ShadowDP, and
2. the assertions in $\tilde{M}'$ (transformed from $\tilde{M}$ by ShadowDP) pass.

**Proof.** The proof is mostly straightforward due to the similarity between the type systems of CheckDP
and ShadowDP. As stated in Section 5.5, the only difference that requires extra work in the proof is
that CheckDP only tracks if a variable has the same value in two related runs (with distance 0) or not
(with distance *), while ShadowDP also allows distance of an arbitrary expression. To gap the potential
difference, we define that $\Gamma'$ and $\tilde{\Gamma}'$ are consistent if

$$\forall x \in V \cup H. \langle x, \Gamma' \rangle^* = \langle x, \tilde{\Gamma}' \rangle^* \land \langle x, \Gamma' \rangle^+ = \langle x, \tilde{\Gamma}' \rangle^+$$
Note that since we only need to convert CheckDP types to the (more expressive) ShadowDP types, such restriction of CheckDP types does not cause any issue.

First we show that if an expression \( e \) of \( M \) type checks with \( \Gamma \) in CheckDP, and all of the generated constraints \( C \) hold, then \( e \) type checks with \( \hat{\Gamma} \) in ShadowDP with an equivalent type (including distances), as long as \( \Gamma \) is consistent with \( \hat{\Gamma} \). We list a few interesting cases here. The proofs for other types of expressions are omitted since their rules in CheckDP are identical other than collecting static checks in ShadowDP as constraints.

- \( e = x \): the interesting case is when \( \Gamma(x) = \mathcal{B}_{\langle *, * \rangle} \) and \( \hat{\Gamma}(x) = \mathcal{B}_{\langle \nu_1, \nu_2 \rangle} \). We have the derived types are equivalent under \( \Gamma \) and \( \hat{\Gamma} \) by the consistency assumption.

- \( e = e_1 ? e_2 : e_3 \): Let \( e_2, e_3 \) be such that \( \Gamma \vdash e_2 : \text{num}_{\langle \nu_1, \nu_2 \rangle}, \Gamma \vdash e_3 : \text{num}_{\langle \nu_3, \nu_4 \rangle} \). The T-Select rule restricts that \( \nu_1 = \nu_2 = \nu_3 = \nu_4 \), which entails the requirement that \( e_2 \) and \( e_3 \) have the same type in the corresponding rule of ShadowDP.

Next, we show that if \( \Gamma \) is consistent with \( \hat{\Gamma} \) and \( \Gamma \vdash \{ M \rightarrow M' \} \) \( \Gamma' \), then \( \Gamma \vdash \{ \hat{M} \rightarrow \hat{M}' \} \) \( \hat{\Gamma}' \) and \( \Gamma' \) and \( \hat{\Gamma}' \) are consistent. We proceed by rule induction on commands. For most rules, all assumptions in ShadowDP rules are guaranteed by the corresponding assertions in CheckDP, making them trivial cases. Next, we present the interesting cases and omit the rest ones.

- \( x := e \): let \( \Gamma \vdash e : \text{num}_{\langle \nu, \nu' \rangle} \). The interesting case is when \( pc = \bot \wedge \nu \neq 0 \). In CheckDP, since \( x := e \) type checks in CheckDP, we know that \( \Gamma'(x) = * \) and \( \hat{x} \) is updated to \( \nu \) after the transformed assignment. In ShadowDP, we have \( \Gamma'(x) = \nu \). Hence, \( \Gamma' \) and \( \hat{\Gamma}' \) are still consistent: \( \langle x, \Gamma' \rangle^\dagger = x + \nu = x + \hat{\Gamma}'(x) = \langle x, \hat{\Gamma}' \rangle^\dagger \).

- \( \eta := g \): the assertion \( c_d \) ensures that the corresponding static check succeeds in rule T-Laplace of ShadowDP. One notable difference between CheckDP and ShadowDP is that since selector \( S \) is unknown statically, a branch \( c_d \) is inserted to update the alignment of aligned execution. For consistency, checking \( \langle x, \Gamma' \rangle^\dagger = \langle x, \hat{\Gamma}' \rangle^\dagger \) is trivial since the shadow distances are updated in the same way as in ShadowDP. When \( S = \circ \), the interesting case is when the distance of \( x \) is promoted to \( * \) (i.e., \( \Gamma'(x) = \text{num}_{\langle \nu, \nu' \rangle} \wedge \Gamma(x) = \text{num}_{\langle 0, \nu' \rangle} \)). In this case, due to the inserted commands \( c' \), \( \langle x, \Gamma' \rangle^\circ = x + \hat{x} = x = \langle x, \hat{\Gamma}' \rangle^\circ \). When \( S = \dagger \), due to the inserted commands \( c'' \),
\( \langle x, \Gamma \rangle^+ = x + \hat{x}^+ = x + \pi^+ = \langle x, \hat{\Gamma} \rangle^0 \) where \( \Gamma \vdash x : \text{num}_{\bot, n} \). Finally, the typing environment changes to \( \Gamma'[\eta \mapsto \text{num}_{(n, 0)}] \) in ShadowDP, but since all nonzero distances are dynamically tracked in CheckDP, this becomes \( \Gamma'[\eta \mapsto \text{num}_{(\epsilon, 0)}] \), which is the one given by CheckDP rule.

\[ \Box \]

**Theorem 5.2 (Privacy).** With exactly the same notation and assumption as Theorem 5.1, \( M \) satisfies \( \epsilon \)-differential privacy.

**Proof.** This follows directly from ShadowDP (Chapter 4, Theorem 4.5) and the fact that \( M \) and \( \tilde{M} \) are semantically the same. \[ \Box \]

### 5.6 Proof and Counterexample Generation

Recall that the transformed source code has the form of the following: \( M'(\text{inp}, \tilde{\text{inp}}, \text{sample}, \theta) \). For brevity, Let \( I \) denote a triple of \( (\text{inp}, \tilde{\text{inp}}, \text{sample}) \), and \( C \) denote a counterexample in the form of \( C = (\text{inp}, \text{inp}', o) \) as defined in Section 5.2.1. Proofs / Counterexamples Generation is divided into two tasks:

- **Proof Generation:** find an instantiation of \( \theta \) such that assertions in \( M' \) never fail for any input \( I \), or
- **Counterexample Generation:** find an instantiation of \( I \), such that no \( \theta \) exists to make all assertions in \( M' \) pass, and then construct a counterexample \( C \) based on \( I \).

The key challenge here is the infinite search space of both \( \theta \) and \( I \). Our insight is to use a verify-invalidate loop, as depicted in Figure 5-7, to improve \( \theta \) and \( I \) after each iteration. At a high-level, the iterative process involves two sub-loops: the (green) verify sub-loop generates proofs, and the (blue) invalidate sub-loop generates counterexamples. Moreover, the two sub-loops are integrated: starting from a default \( \theta_0 \) where \( \theta_0[i] = 0 \). \( \forall i \), the procedure generates sequences of proofs and invalidating inputs in the form of \( \theta_0, I_0, \theta_1, I_1, \ldots \). The final \( \theta_k \) or \( I_k \) is used to construct proof or counterexamples correspondingly.
5.6.1 Verify Sub-Loop

The verify sub-loop that involves Invalidating Input Generation and Proof Generation components is responsible for generating a sequence of improving alignments $\theta_0, \theta_1, \cdots, \theta_i$ such that, if the mechanism is correct, $\theta_i$ is a privacy proof (i.e., $M'(I, \theta_i) \forall I$).

**Invalidating Input Generation** This component takes a proof candidate $\theta_i$ and then tries to find an input $I_i$ such that $\neg M'(I_i, \theta_i)$ (meaning at least one assertion in $M'$ fails).

Intuitively, $\theta_i$ is currently the “best” proof (initially, a default null proof $\theta_0 = \varnothing$ is used to bootstrap the process) and $I_i$, if any, shows that $\theta_i$ is not a valid proof (recall that a proof needs to ensure $M'(I, \theta_i) \forall I$). Hence, we call such $I_i$ an invaliding input of $\theta_i$ and feed it with all previously identified invalidating inputs to the Proof Generation component following the “Verify Sub-loop” edge.

**Proof Generation** This component takes in a series of invalidating inputs $I_0, \cdots, I_i$ seen so far, and tries to find a proof candidate $\theta_i$ such that: $M'(I_0, \theta_i) \land \cdots \land M'(I_i, \theta_i)$.

Intuitively, its goal is to find a proof candidate $\theta_i$ that makes all invalidating inputs seen so far valid. Most likely, an improved proof candidate $\theta_i$ is generated by the component. Then $\theta_i$ feeds back to the Invalidating Input Generation component, closing the loop.
Exit Edges  The verify loop has two exit edges. First, when no invalidating input is generated, \( \theta_i \) is likely a valid proof. Hence, \( \theta_i \) is passed to a verifier with the following condition: \( \forall I. M' (I, \theta_i) \). Due to the soundness result (Theorem 5.2), we have a proof of differential privacy when the verifier passes (the “Exit” edge from Verifier component). Otherwise, CheckDP uses the counterexample returned by the verifier to construct \( I_i \) (the “Verify Sub-loop” edge). We note that the verification step is required since KLEE, the symbolic executor that we use to find invalidating inputs, is unsound (i.e., it might miss an invalidating input) in theory; however, we did not experience any such unsound case of KLEE in our experience.

Second, the Proof Generation component might fail to find an alignment for \( I_0, \cdots, I_i \), a case that will eventually occur for incorrect mechanisms. This exit edge leads to the invalidate sub-loop that we discuss next.

5.6.2 Invalidate Sub-Loop

The invalidate sub-loop involves Counterexample Generation and Restart; it is responsible for generating one invalidating input \( I \) such that, if the mechanism is incorrect, \( I \) cannot be aligned (i.e., \( \not\exists \theta. M' (I, \theta) \)). At first glance, one could be attempting to directly use \( I_i \) from the Verify Sub-Loop. However, this is problematic both in theory and in practice: no alignment for \( I_0, \cdots, I_i \) does not imply no alignment of \( I_i \) alone. For this reason, such a naive approach fails for BadSmartSum and BadSVT4 in Section 5.7.

![Input Space](image1)
(a) A case where \( \theta_i \) cannot be improved.

![Input Space](image2)
(b) Iteratively improving the alignment \( \theta_i \).

Figure 5-8: Tentative alignments and invalidating inputs.
Counterexample Generation  This component takes an invalidating input $I_i$ and then tries to find an alignment $\theta_i$ such that $M'(I_i, \theta_i)$ (meaning that $I_i$ is not a counterexample since it can be aligned by $\theta_i$). For example, consider a corner case in Figure 5-8a, where Proof Generation fails to find a common proof of both $I_0$ and $I_1$, but each of $I_0$ and $I_1$ has a proof (illustrated by the two solid circles around them). Mostly likely, this occurs when the program being analyzed is incorrect (hence, no common proof) but neither $I_1$ nor $I_2$ is a good candidate for counterexample of differential privacy, since each of them can be aligned in isolation.

Restart  This component is symmetric to the Invalidating Input Generation component in the verify sub-loop: it takes all previously found proof candidates $\theta_1, \ldots, \theta_i$ and tries to find an invalidating input $I_{i+1}$ such that:

$$\neg M'(I_{i+1}, \theta_1) \land \cdots \land \neg M'(I_{i+1}, \theta_i).$$

If found, $I_{i+1}$ will intuitively be out of scope of all found proofs and serve as a “better” invalidating input. In theory, we can close the invalidate sub-loop by feeding $I_{i+1}$ back to Counterexample Generation. However, doing so will make proof and counterexample generation isolated tasks. Instead, we take an integrated approach, which we discuss shortly, where the verify and invalidate sub-loops communicate to generate proofs and counterexamples in a more efficient and simultaneous way.

Exit Edges  If no $\theta$ is found to prove $I_i = (inp, \widehat{inp}, sample)$, we can form the counterexample

$$C = (inp, inp + \widehat{inp}, M'(inp, \widehat{inp}, sample, \theta_0))$$

and send it to an external exact probabilistic solver PSI [57] for validation. In theory, the Restart component might fail to find a new invalidating input given $\theta_1, \ldots, \theta_i$. However, this “unknown” state never showed up in our experience.

5.6.3 Integrating Verify and Invalidate Sub-Loops

We integrate the verify and invalidate sub-loops as follows: following the “Invalidate Sub-loop” edge of the Proof Generation component, the latest invalidating input $I_i$ (i.e., the “best” invalidating input so far) is passed to the Counterexample Generation component to start the invalidate sub-loop. Moreover, the
newly generated invalidating input $I_i$ from the Restart component is fed back to the Proof Generation component to start the verify sub-loop.

We note that by the design of the verify-invalidate loop, it alternatively runs Invalidating Input Generation and Proof Generation components. By doing so, the proof keeps improving while the invalidating inputs are getting closer to a true counterexample (since the most recent one violates a “better” proof). More intuitively, consider an invalidating input $I_0$ as a point in the entire input space, illustrated in Figure 5-8b. A proof candidate $\theta_1$ is able to prove the algorithm for a subset of inputs including $I_0$ (indicated by the circle around $I_0$). The Invalidating Input Generation component then tries to find another invalidating $I_1$ that violates $\theta_1$ (falls outside of the $\theta_1$ circle). Next, the Proof Generation component finds better proof candidate $\theta_2$ which proves (“covers”) both $I_0$ and $I_1$.

We also note that it is crucial to consider all invalidating inputs so far rather than the last input $I_i$ in the Proof Generation component: the efficiency of our approach crucially relies on “improving” the proofs quantified by validating more invalidating inputs. Without the improving proofs, the iterative procedure might fail to terminate in case shown in Figure 5-8a: the procedure might repeat $I_0, \theta_1, I_1, \theta_2, I_0, \theta_1, \ldots$. This is confirmed in our empirical study.

**Unknown State** Due to the soundness result (Theorem 5.2), the program being analyzed is verified whenever CheckDP returns with a proof. Moreover, a validated counterexample by PSI disproves an incorrect mechanism. However, two reasons might lead to the “unknown” state in the Figure 5-7: the generated counterexample is invalid or the Restart component fails to find a new invalidating input. However, for all the correct and incorrect examples we explored, the unknown state never showed up.

### 5.7 Implementation and Evaluation

#### 5.7.1 Implementation

We implemented CheckDP in Python. Details of the *Program Transformation* and *Proof and Counterexample Generation* phases are as follows: The program transformation is implemented as a trans-compiler from CheckDP code (Figure 5-2) to C code. Following the transformation rules in Figure 5-3 and
Figure 5-4, the trans-compiler tracks the typing environment, gathers the needed constraints for the expressions, and more importantly, instruments corresponding statements when appropriate. Moreover, it adds a final assertion \( \text{assert} (v_e \leq \epsilon_b) \) before each \textbf{return} command, where \( \epsilon_b \) is the annotated privacy bound to be checked. Once all assertions are generated, the trans-compiler generates one alignment template for each sampling instruction as described in Algorithm 1.

For the verify-invalidate loop in Section 5.2.1, we used an efficient symbolic executor KLEE [60] for most tasks. Due to limited support of unbounded lists in KLEE, we fix the length of lists to be 5 in our evaluation. Also, to speed up the search, KLEE is configured to exit once an assertion is hit.

For the Verifier component, we deploy program verification tool CPAChecker [47], which is capable of automatically verifying C programs with given configuration (\textit{predicateAnalysis} is used). Note that CPAChecker is able to generate counterexamples for a failed verification. If the verification fails (which did not happen in our evaluation), CheckDP can feed the counterexample back to the Proof and Counterexample Generation component.

5.7.2 Case Studies

Aside from GapSVT, we also evaluate on the standard benchmark used in previous mechanism verifiers [16, 14, 15] and counterexample generators [29, 30],

\footnote{We note that like all tools designed for privacy mechanisms (e.g., [16, 14, 15, 29, 30]), the benchmark do not include iterative programs that are built on those privacy mechanisms, such as k-means clustering, k-medians, since they are out of scope.}

including correct ones such as NumSVT, PartialSum, and SmartSum, as well as the incorrect variants of SVT reported in [12] and BadPartialSum. To show the power of CheckDP and expressiveness of our template generation algorithm, we also evaluate on a couple of correct/incorrect mechanisms that, to the best of our knowledge, have not been proved/disproved by existing verifiers and counterexample generators. This set of mechanisms include: Sparse Vector with monotonic queries [12], AdaptiveSVT (called Adaptive Sparse Vector with Gap in [44]) as well as new incorrect variants of SVT, AdaptiveSVT and SmartSum. For all mechanisms we explore, CheckDP is able to: (1) provide a proof if it satisfies differential privacy, or (2) provide a counterexample if it violates the claimed level of privacy. Neither false positives nor false negatives were observed in the case studies.
function SVT(T,N,size.tap: num, q.tap: list num,)
returns (out: list bool), bound(\epsilon)
precondition \forall i. -1 \leq (\hat{q}[i]) \leq 1

1 \eta_1 := \text{Lap}(2/\epsilon)
2 T_* := T + \eta_1;
3 count := \emptyset; i := \emptyset;
4 while (count < N \land i < size)
5 \eta_2 := \text{Lap}(4N/\epsilon)
6 if (q[i] + \eta_2 \geq T_*) then
7     out := true::out;
8     count := count + 1;
9 else
10     out := false::out;
11     i := i + 1;

function TRANSFORMED SVT(T,N,size,q, \hat{q}, sample, \theta)
returns (out)

12 v_e := \emptyset; idx = \emptyset;
13 \eta_1 := sample[idx]; idx := idx + 1;
14 v_e := v_e + |A_1| \times \epsilon/2; \hat{\eta}_1 := \hat{A}_1;
15 T_* := T + \eta_1;
16 \hat{T}_* := \hat{\eta}_1;
17 count := \emptyset; i := \emptyset;
18 while (count < N \land i < size)
19 \eta_2 := sample[idx]; idx := idx + 1;
20 v_e := v_e + |A_2| \times \epsilon/4N; \hat{\eta}_2 := \hat{A}_2;
21 if (q[i] + \eta_2 \geq T_*) then
22     assert (q[i] + \eta_2 + \hat{q}[i] + \hat{\eta}_2 \geq T_* + \hat{T}_*);
23     out := true::out;
24     count := count + 1;
25 else
26     assert (\neg(q[i] + \eta_2 + \hat{q}[i] + \hat{\eta}_2 \geq T_* + \hat{T}_*));
27     out := false::out;
28     i := i + 1;
29 assert (v_e \leq \epsilon);

Figure 5-9: Standard Sparse Vector Technique and its transformed code, where underlined parts are added by CheckDP. The transformed code contains two alignment templates for \eta_1 and \eta_2: \hat{A}_1 = \theta[0] and \hat{A}_2 = (q[i] + \eta_2[i] \geq T_*) ? (\theta[1] + \theta[2] \times \hat{T}_* + \theta[3] \times \hat{q}[i]) : (\theta[4] + \theta[5] \times T_* + \theta[6] \times \hat{q}[i]).

We first show a correctly-implemented standard version of SVT [12]. This standard implementation outputs less information than the running example GapSVT, as it outputs true instead of the gap between
noisy query answer and noisy threshold. It can be obtained by changing Line 7 in Figure 5-1 from out := (q[i] + \eta_2)::out; to out := true::out;.

**Sparse Vector with Monotonic Queries** There exist use cases with SVT where the queries are monotonic. More formally, queries are monotonic if for related queries \( q \sim q', \forall i. q_i \leq q'_i \) or \( \forall i. q_i \geq q'_i \). In such cases, it suffices to add \( \eta_2 := \text{Lap}(2N/\epsilon) \) to each queries (Line 5 in Figure 5-1) and the algorithm still satisfies \( \epsilon \)-DP [12].

Thanks to the expressiveness of CheckDP, it only requires one modification in the function specification in order to verify this variant: modify the constraint on \( \bar{q}[i] \) in the precondition.

Specifically, the new precondition for SVT with monotonic queries becomes \( \forall i. 0 \leq \bar{q}[i] \leq 1 \) for the \( \forall i. q_i \leq q'_i \) and \( \forall i. -1 \leq \bar{q}[i] \leq 0 \) for the other case. The final found alignment by CheckDP is the same as the ones reported in the manual randomness alignment based proofs [44]:

\[
\eta_1 : 0, \quad \eta_2 : \begin{cases} 
q[i] + \eta_2 \geq T* ? 1 - \bar{q}[i] : 0, & \text{if } \forall i. q_i \leq q'_i \\
q[i] + \eta_2 \geq T* ? -\bar{q}[i] : 0, & \text{otherwise}
\end{cases}
\]

To the best of our knowledge, no prior verification works have automatically verified this variant.

**SVT with Wrong Privacy Claims (Imprecise SVT)** We also study another interesting yet quite challenging violation of differential privacy: suppose a mechanism satisfies 1.1-differential privacy but claims to be 1-differentially private. This slight violation requires precise reasoning about the privacy cost and poses challenges for prior sampling-based approaches. We thus evaluate a variant of SVT, referred to as Imprecise SVT, which is \( \epsilon = 1.1 \)-differentially private but with an incorrect claim of \( \epsilon = 1 \) (\text{check}(1) in the signature).

**NumSVT** Numerical Sparse Vector (NumSVT) [28] is another interesting correct variant of SVT which outputs a numerical answer when the input query is larger than the noisy threshold. It follows the same procedure as Sparse Vector Technique, the difference is that it draws a fresh noise \( \eta_3 \) in the \text{true} branch, and outputs \( q[i] + \eta_3 \) instead of \text{true}. Note that this is very similar to our running example GapSVT and BadGapSVT, the key difference is that the freshly-drawn random noise hides the information about
\begin{verbatim}
function ADAPTIVESVT (T,N,size: num, q: list num.)
returns (out: list num, bound(\epsilon))
precondition \forall i. -1 \leq (q[i]) \leq 1

1  cost := 0;
2  \eta_1 := \text{Lap} (2/\epsilon);
3  cost := cost + \epsilon/2;
4  T_* := T + \eta_1;
5  i := 0;
6  while (cost \leq \epsilon - 2 \times \epsilon/4N \land i < size)
7      \eta_2 := \text{Lap} (8N/\epsilon);
8      if (q[i] + \eta_2 - T_* \geq \sigma) then
9          out := (q[i] + \eta_2 - T_*)::out;
10         cost := cost + 2 \times \epsilon/(8N);
11      else
12          \eta_3 := \text{Lap} (4N/\epsilon);
13          if (q[i] + \eta_3 - T_* \geq 0) then
14              out := (q[i] + \eta_3 - T_*)::out;
15              cost := cost + 2 \times \epsilon/(4N);
16          else
17              out := 0::out;
18      i := i + 1;

Figure 5-10: Pseudo-code for Adaptive SVT.
\end{verbatim}

\(T_\ast\), unlike the BadGapSVT. This variant can be obtained by making the following changes in Figure 5-1:
(1) Line 1 is changed from Lap 2/\epsilon to Lap 3/\epsilon; (2) Line 5 is changed from Lap 4N/\epsilon to Lap 6N/\epsilon;
(3) Line 7 is change from out := (q[i] + \eta)\::out; to “\eta_3 := \text{Lap} (3N/\epsilon); out := (q[i] + \eta_3)\::out;”. CheckDP finds the same alignment as shown in [14] with which CPAChecker is able to verify the algorithm with this generated alignment.

**AdaptiveSVT and BadAdaptiveSVT** Ding et al. [44] recently proposed a new variant of SVT which adaptively allocates privacy budget, saving privacy cost when noisy query answers are much larger than the noisy threshold. The difference from standard (correct) SVT is that it first draws a \(\eta_2 := \text{Lap} 8N/\epsilon\) noise (instead of Lap 4N/\epsilon in SVT) and checks if the gap between noisy query and noisy threshold \(T_\ast\) is larger than a preset hyper-parameter \(\sigma\) (if \(q[i] + \eta_2 - T_\ast \geq \sigma\)). If the test succeeds, the gap is directly returned, hence costing only \(\epsilon/(8N)\) (instead of \(\epsilon/(4N)\)) privacy budget. Otherwise, it
function Transformed AdaptiveSVT (T,N,size,q,\(\widehat{q}\),sample,\(\theta\))
returns (out)

12 \(v_\varepsilon := \emptyset\); idx := \emptyset;
13 \(\eta_1 := \text{sample}[\text{idx}]\); idx := idx + 1;
14 \(v_\varepsilon := v_\varepsilon + |A_1| \times \varepsilon/2\); \(\widehat{\eta}_1 := A_1\);
15 \(T_* := T + \eta_1\);
16 \(\widehat{T}_* := \widehat{\eta}_1\);
17 count := 0; i := 0;
18 while (cost \leq \varepsilon - 2 \times \varepsilon/4N \land i < size)
19 \(\eta_2 := \text{sample}[\text{idx}]\); idx := idx + 1;
20 \(v_\varepsilon := v_\varepsilon + |A_2| \times \varepsilon/8N\); \(\widehat{\eta}_2 := A_2\);
21 if (q[i] + \(\eta_2 - T_* \geq \sigma\)) then
22 \hspace{0.5cm} assert (q[i] + \(\eta_2 + \widehat{q}[i] + \widehat{\eta}_2 - (T_* + \widehat{T}_*) \geq \sigma\));
23 \hspace{0.5cm} assert (q[i] + \(\eta_2 - \widehat{T}_* == \emptyset\));
24 \hspace{0.5cm} out := (q[i] + \(\eta_2 - T_*\))::out;
25 \hspace{0.5cm} cost := cost + 2 \times \varepsilon/(8N);
26 else
27 \hspace{0.5cm} assert (-(q[i] + \(\eta_2 + \widehat{q}[i] + \widehat{\eta}_2 - (T_* + \widehat{T}_*) \geq \sigma\)));
28 \hspace{0.5cm} \eta_3 := \text{sample}[\text{idx}]\); idx := idx + 1;
29 \hspace{0.5cm} v_\varepsilon := v_\varepsilon + |A_3| \times \varepsilon/4N\); \(\widehat{\eta}_3 := A_3\);
30 if (q[i] + \(\eta_3 - T_* \geq \emptyset\))
31 \hspace{0.5cm} assert (q[i] + \(\eta_3 + \widehat{q}[i] + \widehat{\eta}_3 - (T_* + \widehat{T}_*) \geq \emptyset\));
32 \hspace{0.5cm} assert (q[i] + \(\eta_3 - \widehat{T}_* == \emptyset\));
33 \hspace{0.5cm} out := (q[i] + \(\eta_3 - T_*\))::out;
34 \hspace{0.5cm} cost := cost + 2 \times \varepsilon/(4N);
35 else
36 \hspace{0.5cm} assert (-(q[i] + \(\eta_3 + \widehat{q}[i] + \widehat{\eta}_3 - (T_* + \widehat{T}_*) \geq \emptyset\));
37 \hspace{0.5cm} out := false::out;
38 \hspace{0.5cm} i := i + 1;
39 assert (v_\varepsilon \leq \varepsilon);

assert \(v_\varepsilon \leq \varepsilon\);

Figure 5.11: The transformed code for AdaptiveSVT, where underlined parts are added by CheckDP. It contains three alignment templates for \(\eta_1\) and \(\eta_2\): \(A_1 = \theta[0]\), \(A_2 = \Omega_{Top} \quad \theta[1] + \theta[2] \times \widehat{T}_* + \theta[3] \times \widehat{q}[i] + \theta[4] + \theta[5] \times T_* + \theta[6] \times \widehat{q}[i]\) and \(A_3 = \Omega_{Middle} \quad \theta[1] + \theta[2] \times \widehat{T}_* + \theta[3] \times \widehat{q}[i] + \theta[4] + \theta[5] \times T_* + \theta[6] \times \widehat{q}[i]\), where \(\Omega\) denotes the corresponding branch condition at Line 8 and 13.

draws \(\eta_3 := \text{Lap} 4N/\varepsilon\) and follows the same procedure as SVT. We also create an incorrect variant called BadAdaptiveSVT. It directly releases the noisy query answer instead of the gap after the first test.
Sampling-based methods can have difficulty detecting the privacy leakage because the privacy-violating branch of the BadAdaptiveSVT code is not executed frequently. We also create an incorrect variant of SmartSum by releasing a noise-less sum of queries in an infrequent branch.

**Report Noisy Max [34]** This is an important building block for developing differentially private algorithms. It generates differentially private synthetic data by finding the identity with the maximum (noisy) score in the database. Here we present this mechanism in a simplified manner: for a series of query answers $q$, where each of them can differ at most one in the adjacent underlying database, its goal is to return the index of the maximum query answer in a privacy-preserving way. To achieve differential privacy, the mechanism first adds $\eta = \text{Lap} \frac{2}{\epsilon}$ noise to each of the query answer, then returns the index of the maximum noisy query answers $q[i] + \eta$, instead of the true query answers $q[i]$. The pseudo code of this mechanism is shown in Figure 5-12.

To prove its correctness using randomness alignment technique, we need to align the only random variable $\eta$ in the mechanism (Line 3). Therefore, a corresponding privacy cost of aligning $\eta$ would be incurred for each iteration of the loop. However, manual proof [34] suggests that we only need to align the random variable added to the actual maximum query answer. In other words, we need an ability to “reset” the privacy cost upon seeing a new current maximum noisy query answer.

**Bad Noisy Max** We also created an incorrect variant of Report Noisy Max. This variant directly returns the maximum noisy query answer, instead of the index.

More specifically, it can be obtained by changing Line 5 in Figure 5-12 from $\text{max} := i$ to $\text{max} := q[i] + \eta$. CheckDP is then able to find a counterexample for this incorrect variant.

**BadSVT1 - 3** We now study other three incorrect variants of SVT collected from [12]. All three variants are based on the classic SVT algorithm we have seen (i.e., Line 7 in Figure 5-1 is $\text{out} := \text{true}::\text{out;}$).

BadSVT1 [64] adds no noise to the query answers and has no bounds on the number of true’s it can output. This variant is obtained by changing Line 4 from $\textbf{while} (\text{count}<N \land i<\text{size})$ to $\textbf{while} (i<\text{size})$ and Line 5 from $\text{Lap} (4N/\epsilon)$ to 0. Another variant BadSVT2 [65] has no bounds on the number of true’s it can output as well. It keeps outputting true even if the given privacy budget has
Figure 5-12: Report Noisy Max and its transformed code, where $S = q[i] + \eta > bq \lor i = 0 ? \theta[0] : \theta[1]$ and $A = q[i] + \eta > bq \lor i = 0 ? \theta[2] + \theta[3] \times \tilde{q}[i] + \theta[4] \times \tilde{bq}^\circ : \theta[5] + \theta[6] \times \tilde{q}[i] + \theta[7] \times \tilde{bq}$.
Figure 5-13: BadSVT1 and its transformed code, where underlined parts are added by CheckDP. The transformed code contains a alignment template for $\eta_1$: $A_1 = \theta[0]$. 

```plaintext
function BadSVT1 (T,N,size: num_0, q: list num_0)
returns (out: list bool, bound(\epsilon))
precondition \forall i. \ -1 \leq (q[i]) \leq 1

\eta_1 := \text{Lap}(2/\epsilon);
T_\star := T + \eta_1;
count := 0; i := 0;
while (i < size)
\eta_2 := \{0\};
if (q[i] + \eta_2 \geq T_\star) then
out := true::out;
count := count + 1;
else
out := false::out;
i := i + 1;

function TransformedBadSVT1 (T,N,size,q, q, \text{sample}, \theta)
returns (out)

v_\epsilon := 0; idx = 0;
\eta_1 := \text{sample}[idx]; idx := idx + 1;
v_\epsilon := v_\epsilon + |A_1| \times \epsilon / 2; \tilde{\eta}_1 := A_1;
T_\star := T + \eta_1;
\tilde{T}_\star := \tilde{\eta}_1;
count := 0; i := 0;
while (i < size)
\eta_2 := \{0\};
if (q[i] + \eta_2 \geq T_\star) then
assert (q[i] + \eta_2 + q[i] \geq T_\star + \tilde{T}_\star);
out := true::out;
count := count + 1;
else
assert (-(q[i] + \eta_2 + q[i] \geq T_\star + \tilde{T}_\star));
out := false::out;
i := i + 1;
assert (v_\epsilon \leq \epsilon);
```
function BADSVT2(T,N,size,q: list num0, bound(ε))
returns (out: list bool, bound(ε))

precondition ∀ i. -1 ≤ (q[i]) ≤ 1

1. η₁ := Lap (2/ε);
2. T* := T + η₁;
3. count := 0; i := 0;
4. while (i < size)
5.     η₂ := Lap (2/ε);
6.     if (q[i] + η₂ ≥ T*) then
7.         out := true::out;
8.         count := count + 1;
9.     else
10.        out := false::out;
11.        i := i + 1;

function TRANSFORMED BADSVT2(T,N,size,q, q̃, sample, θ)
returns (out)

12. ṽε := 0; idx = 0;
13. η₁ := sample[idx]; idx := idx + 1;
14. ṽε := ṽε + |A₁| × ε/2;  η̃₁ := A₁;
15. T̃* := T + η₁;
16. ̃T* := ̃η₁;
17. count := 0; i := 0;
18. while (i < size)
19.     η₂ := sample[idx]; idx := idx + 1;
20.     ṽε := ṽε + |A₂| × ε/2;  η̃₂ := A₂;
21.     if (q[i] + η₂ ≥ T*) then
22.         assert (q[i] + η₂ + ̃q[i] + η̃₂ ≥ T* + ̃T*);
23.         out := true::out;
24.         count := count + 1;
25.     else
26.         assert (¬(q[i] + η₂ + ̃q[i] + η̃₂ ≥ T* + ̃T*));
27.         out := false::out;
28.     i := i + 1;
29.     assert (ṽε ≤ ε);

Figure 5-15: BadSVT3 and its transformed code, where underlined parts are added by CheckDP. The transformed code contains two alignment templates for $\eta_1$ and $\eta_2$: $A_1 = \theta[0]$ and $A_2 = (q[i] + \eta_2 \geq T_\star) \land (\theta[1] + \theta[2] \times T_\star + \theta[3] \times q[i]) : (\theta[4] + \theta[5] \times T_\star + \theta[6] \times q[i])$. 

```plaintext
function BADSVT3 (T,N,size,num0,q:list num) returns (out:list bool, bound(\epsilon))
precondition \forall i. -1 \leq (q[i]) \leq 1

1  \eta_1 := Lap (4/\epsilon);
2  T_\star := T + \eta_1;
3  count := 0; i := 0;
4  while (count < N \land i < size)
5      \eta_2 := Lap (4/3\epsilon);
6      if (q[i] + \eta_2 \geq T_\star) then
7          out := true::out;
8          count := count + 1;
9      else
10         out := false::out;
11         i := i + 1;
12
function TRANSFORMED BADSVT3 (T,N,size,q,q, sample, \theta) returns (out)

12  v_\epsilon := 0; idx = 0;
13  \eta_1 := sample[idx]; idx := idx + 1;
14  v_\epsilon := v_\epsilon + |A_1| \times \epsilon/4; \eta_1 := A_1;
15  \tilde{T}_\star := T + \eta_1;
16  \tilde{T}_\star := \eta_1;
17  count := 0; i := 0;
18  while (count < N \land i < size)
19      \eta_2 := sample[idx]; idx := idx + 1;
20      v_\epsilon := v_\epsilon + |A_2| \times 3\epsilon/4; \eta_2 := A_2;
21      if (q[i] + \eta_2 \geq T_\star) then
22          assert (q[i] + \eta_2 + q[i] + \eta_2 \geq T_\star + \tilde{T}_\star);
23          out := true::out;
24          count := count + 1;
25      else
26          assert (\neg(q[i] + \eta_2 + q[i] + \eta_2 \geq T_\star + \tilde{T}_\star));
27          out := false::out;
28          i := i + 1;
29          assert (v_\epsilon \leq \epsilon);
```
spend its privacy budget in a different allocation strategy between the threshold $T$ and the query answers $q[i]$ ($1:3$ instead of $1:1$). However, the noise added to $\eta_2$ does not scale with parameter $N$: the $3/4$ privacy budget is allocated to each of the queries instead of being shared among them. More specifically, the noise generation commands (Line 1 and Line 5) in Figure 5-9 can be changed to $\eta_1 := \text{Lap}(4/\epsilon)$ and $\eta_2 := \text{Lap}(4/(3 \times \epsilon))$ to get this variant.

Note that the generated templates for the incorrect variants are identical to those of GapSVT since they all have similar typing environments, with one exception BadSVT1, since it does not sample $\eta_2$ at all.

Interestingly, since the errors share similar characteristics (no bounds on number of outputs / wrong scale of added noise), CheckDP finds a common counterexample $[0, 0, 0, 0, 0], [1, 1, 1, 1, -1]$ where $T = 0$ and $N = 1$ within 6 seconds, and this counterexample is further validated by PSI.

### 5.7.3 Partial Sum

Next, we study a simple algorithm PartialSum (Figure 4-12) which outputs the sum of queries in a privacy-preserving manner: it directly computes sum of all queries and adds a $\text{Lap}(1/\epsilon)$ to the final output sum. Note that similar to SmartSum, it has the same adjacency requirement (only one query can differ by at most one). The alignment is easily found for $\eta$ by CheckDP which is to “cancel out” the distance of sum variable (i.e., $-\text{sum}$). With the alignment CPAChecker verifies this algorithm.

An incorrect variant for PartialSum called BadPartialSum is created where Line 5 is changed from $1/\epsilon$ to $1/(2 \times \epsilon)$, therefore making it fail to satisfy $\epsilon$-differential privacy (though it actually satisfies $2\epsilon$-differential privacy). A counterexample $[0, 0, 0, 0, 0], [0, 0, 0, 0, 1]$ is found by CheckDP and further validated by PSI.

### 5.7.4 SmartSum and BadSmartSum

SmartSum [56] continually releases aggregated statistics with privacy protections. For a finite sequence of queries $q[0], q[1], \cdots, q[T]$, where $T$ is the length of $q$, the goal of SmartSum is to release the prefix
function PartialSum(size: num, q: list num,)
returns (out: num, bound(ε))
precondition ∀i. -1 ≤ (q[i]) ≤ 1 ∧ (∀i. (q[i]) ≠ 0 ⇒ (∀j. q[j] = 0))

1  sum := 0; i := 0;
2  while (i < size)
3      sum := sum + q[i];
4      i := i + 1;
5  η = Lap (1/ε);
6  out := sum + η;

function Transformed PartialSum(size,q, sample, θ)
returns (out)

7  v_ε := 0; sum := 0;
8  sum := 0; i := 0;
9  while (i < size)
10     sum := sum + q[i];
11     sum := sum + q[i];
12     i := i + 1;
13     v_ε := v_ε + |A| × ε; η := A;
14     assert (sum + η = 0);
15     out := sum + η;
16     assert (v_ε ≤ ε);

Figure 5-16: PartialSum and its transformed code, where underlined parts are added by CheckDP. The transformed code contains an alignment template A = θ[0] + θ[1] × sum + θ[2] × q[i].

sum: q[0], q[1], · · · , Σ_{i=0}^T q[i] in a private way. To achieve differential privacy, SmartSum first divides the sequence into non-overlapping blocks B_0, · · · , B_i with size M, then maintains the noisy version of each query and noisy version of the block sum, both by directly adding Lap 1/ε noise. Then to compute the k-th component of the prefix sum sequence Σ_{i=0}^k q[i], it only has to add up the noisy block sum that covers before k, plus the remaining (k + 1) mod M noisy queries. The pseudo code is shown in Figure 5-17 and its transformed code is shown in Figure 5-18. The if branch is responsible for dividing the queries and summing up the block sums (stored in sum variable), where else branch adds the remaining noisy queries.

Notably, SmartSum satisfies 2ε-differential privacy instead of ε-differential privacy. Moreover, the
function SmartSum(M, T, size: num, q: list num) returns (out: list num, bound(2ε))
precondition ∀i. -1 ≤ q[i]) ≤ 1 ∧ (∀i. (q[i]) ≠ 0 ⇒ (∀j. q[j] = 0))

1. next := 0; i := 0; sum := 0;
2. while (i < size ∧ i ≤ T)
3.    if ((i + 1) mod M = 0) then
4.        η₁ := Lap (1/ε);
5.        next := sum + q[i] + η₁;
6.        sum := 0;
7.        out := next::out;
8.    else
9.        η₂ := Lap (1/ε);
10.       next := next + q[i] + η₂;
11.       sum := sum + q[i];
12.       out := next::out;
13.       i := i + 1;

Figure 5-17: Pseudo-code for SmartSum.

The adjacency requirement of the inputs is that only one of the queries can differ by at most one. These two requirements are specified in the function signature (bound(2ε) and precondition).

An incorrect variant of SmartSum, called BadSmartSum, is obtained by changing Line 4 to η₁ := 0 in Figure 5-17. It directly releases sum + q[i] without adding any noise (since η₁ = 0), where sum stores the accurate, non-noisy sum of queries (at Line 11), hence breaking differential privacy. Interestingly, the violation only happens in a rare branch if ((i + 1) mod M = 0), where the accurate sum is added to the output list out. In other words, out contains mostly private data with only a few exceptions. This rare event makes it challenging for sampling-based tools to find the violation.

5.7.5 Experiments

We evaluate CheckDP on a Intel® Xeon® E5-2620 v4 @ 2.10 GHz CPU machine with 64 GB memory. To compare CheckDP with the state-of-the-art tools, we run publicly available tools (including StatDP [29]
function Transformed SmartSum (M, T, size, q, sample, \( \theta \))
returns (out)

14 \( v_e := \emptyset; \) idx := 0;
15 next := 0; i := 0; sum := 0;
16 sum := 0; next := 0;
17 while (i < size \wedge i \leq T)
18 if ((i + 1) mod M = 0) then
19 \( \eta_1 := \text{sample}[\text{idx}]; \) idx := idx + 1;
20 \( v_e := v_e + |A_1| \times \epsilon; \) \( \eta_1 := A_1; \)
21 next := sum + q[i] + \( \eta_1; \)
22 next := sum + \( q[i] + \eta_1; \)
23 sum := 0;
24 \( \bar{s} \) := 0;
25 assert (next = 0);
26 out := next::out;
27 else
28 \( \eta_2 := \text{sample}[\text{idx}]; \) idx := idx + 1;
29 \( v_e := v_e + |A_2| \times \epsilon; \) \( \eta_2 := A_2; \)
30 next := next + q[i] + \( \eta_2; \)
31 next := next + \( q[i] + \eta_2; \)
32 sum := sum + q[i];
33 \( \bar{s} \) := \( \bar{s} + q[i]; \)
34 assert (next = 0);
35 out := next::out;
36 i := i + 1;
37 assert (v_e \leq 2\epsilon);

Figure 5-18: The transformed code for SmartSum. Underlined parts are added by CheckDP. \( A_1 = \theta[0] + \theta[1] \times \text{sum} + \theta[2] \times \bar{q}[i] + \theta[3] \times \text{next} \) and \( A_2 = \theta[4] + \theta[5] \times \text{sum} + \theta[6] \times \bar{q}[i] + \theta[7] \times \text{next} \).

and DP-Finder [30]) on the benchmark.\(^4\) For the only existing automatic differential privacy verifier [16], we reuse the reported results in [16] when possible, since the tool is not available.

**Counterexample Generation** Table 5-1 lists the counterexamples (i.e., a pair of related inputs and a feasible output that witness the violation of claimed level of privacy) automatically generated by

\(^4\)Default settings are used in our evaluation: 100K/500K samples for event selection/hypothesis testing components of StatDP; 50 iterations for sampling and optimization components of DP-Finder where each iteration collects 409,600 samples on average.
Table 5-1: Detected counterexamples for the incorrect algorithms and comparisons with other sampling-based counterexample detectors. #t stands for true and #f stands for false.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>q1</th>
<th>q2</th>
<th>q3</th>
<th>Extra Args</th>
<th>Output</th>
<th>Iterations</th>
<th>Time(s)</th>
<th>StatDP [29]</th>
<th>DP-Finder [30]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BadNoisyMax</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[−1, 1, 1, 1]</td>
<td>N/A</td>
<td>0</td>
<td>3</td>
<td>5.7</td>
<td></td>
</tr>
<tr>
<td>BadSVT1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[1, 1, 1, 1, 1]</td>
<td>T: 0, N: 1</td>
<td>#f, #f, #f, #f, #f, #f</td>
<td>4</td>
<td>3.2</td>
<td>4.9 (Semi-Manual)</td>
</tr>
<tr>
<td>BadSVT2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[1, 1, 1, 1, 1]</td>
<td>T: 0, N: 1</td>
<td>#f, #f, #f, #f, #f, #f</td>
<td>4</td>
<td>2.0</td>
<td>15.6 (Semi-Manual)</td>
</tr>
<tr>
<td>BadSVT3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[1, 1, 1, 1, 1]</td>
<td>T: 0, N: 1</td>
<td>#f, #f, #f, #f, #f, #f</td>
<td>4</td>
<td>2.1</td>
<td>9.1 (Semi-Manual)</td>
</tr>
<tr>
<td>BadSVT4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[1, 1, 1, 1, 1]</td>
<td>T: 0, N: 1</td>
<td>[0, 0, 0, 0, 1]</td>
<td>4</td>
<td>5.7</td>
<td>10.6 (Semi-Manual)</td>
</tr>
<tr>
<td>BadAdaptiveSVT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[1, 1, 1, 1, 1]</td>
<td>T: 0, N: 1</td>
<td>[0, 0, 0, 0, 1, 17]</td>
<td>8</td>
<td>14.2</td>
<td>Search Failed</td>
</tr>
<tr>
<td>Imprecise SVT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[1, 1, 1, 1, 1]</td>
<td>T: 0, N: 1</td>
<td>#f, #f, #f, #f, #f, #f</td>
<td>4</td>
<td>8.6</td>
<td>Search Failed</td>
</tr>
<tr>
<td>BadSmartSum</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[0, 0, 0, 1, 0]</td>
<td>T: 3, M: 4</td>
<td>[0, 0, 0, 0, 0]</td>
<td>4</td>
<td>6.3</td>
<td>22.4 (Semi-Manual)</td>
</tr>
<tr>
<td>BadPartialSum</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[0, 0, 0, 0, 1]</td>
<td>N/A</td>
<td>0</td>
<td>3</td>
<td>3.7</td>
<td>3.8 (Semi-Manual)</td>
</tr>
</tbody>
</table>

Table 5-2: Alignments found for the correct algorithms.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Arrangement</th>
<th>Alignment</th>
<th>Iterations</th>
<th>Time(s)</th>
<th>ShadowDP [15]</th>
<th>Coupling [16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReportNoisyMax</td>
<td>Ω_{N,M} = 1 - q[i]</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>10</td>
<td>Manual</td>
</tr>
<tr>
<td>PartialSum</td>
<td>−siμm</td>
<td>N/A</td>
<td>N/A</td>
<td>2</td>
<td>5.6</td>
<td>Manual</td>
</tr>
<tr>
<td>SmartSum</td>
<td>−qmμm</td>
<td>N/A</td>
<td>N/A</td>
<td>6</td>
<td>6.8</td>
<td>Manual</td>
</tr>
<tr>
<td>SVT</td>
<td>1</td>
<td>Ω_{SVT} = 1 - q[i]</td>
<td>0</td>
<td>N/A</td>
<td>4</td>
<td>Manual</td>
</tr>
<tr>
<td>Monotone SVT (Increase)</td>
<td>0</td>
<td>Ω_{SVT} = 1 - q[i]</td>
<td>0</td>
<td>N/A</td>
<td>8</td>
<td>Manual</td>
</tr>
<tr>
<td>Monotone SVT (Decrease)</td>
<td>0</td>
<td>Ω_{SVT} = 1 - q[i]</td>
<td>0</td>
<td>N/A</td>
<td>8</td>
<td>Manual</td>
</tr>
<tr>
<td>GapsVT</td>
<td>1</td>
<td>Ω_{SVT} = 2 - q[i]</td>
<td>0</td>
<td>N/A</td>
<td>6</td>
<td>Manual</td>
</tr>
<tr>
<td>NumbVT</td>
<td>1</td>
<td>Ω_{SVT} = 2 - q[i]</td>
<td>0</td>
<td>N/A</td>
<td>4</td>
<td>Manual</td>
</tr>
<tr>
<td>AdaptiveSVT</td>
<td>1</td>
<td>Ω_{SVT} = 1 - q[i]</td>
<td>0</td>
<td>N/A</td>
<td>10</td>
<td>Manual</td>
</tr>
</tbody>
</table>

CheckDP for the incorrect algorithms. For all incorrect algorithms, CheckDP is able to provide a counterexample (validated by PSI [57]) in 15 seconds and 8 iterations.₅

Notably, both StatDP and DP-Finder fail to find the privacy violations in BadSmartSum and BadAdaptiveSVT, as well as the violation of ϵ = 1-privacy in Imprecise SVT after hours of searching.₆

This is due to the limitations of sampling-based approaches. In certain cases, we can help these sampling-based algorithms by manually providing proper values for the extra arguments that some of the mechanisms require (4ᵗʰ column of Table 5-1). This extra advantage (labeled Semi-Manual in the table) sometimes allows the sampling-based methods to find counterexamples. We note that CheckDP, in contrast, generates all inputs automatically.

Verification Table 5-2 lists the automatically generated proofs (i.e., alignments) for each random variable in the correct algorithms. Due to the soundness of CheckDP, all returned proofs are valid. We note that correct algorithms on average take more iterations (and hence, time) to verify; still all of them are verified within 70 seconds. Notably, CheckDP in fact provides a more precise alignment

₅We note that the counterexample of BadSmartSum is validated on a slightly modified algorithm since PSI does not support modulo operation.

₆For StatDP, we use 10000 of the default number of samples to confirm the failure.
\(q[i] + \eta_2 \geq T_\star ? 1 - \bar{q}[i] \ldots 0\) (same as the one in GapSVT) than the less precise (though still correct) alignment \(q[i] + \eta_2 \geq T_\star \ldots 2 \ldots 0\) manually generated in [14]. Report Noisy Max is the only example that uses shadow execution; the selector generated is \(S = q[i] + \eta_2 \geq bq \lor i = 0 \land \hat{\omega} = 0\), the same as the manually generated one in Chapter 4.

**Performance** We note that all examples finish within 10 iterations. We contribute the efficiency to the reduced search space of Algorithm 1 (e.g., the alignment template for GapSVT only contains 7 “holes”) as well as our novel verify-invalidate loop that allows verification and counterexample generation components to communicate in meaningful ways.

Compared with StatDP and DP-Finder, CheckDP is more efficient on the cases where they do find counterexamples. Compared with an automatic verifier [16], we note that the performance is comparable in most cases, but CheckDP is much faster on SmartSum and SVT. In summary, CheckDP is mostly more efficient compared to counterexample detectors and automated provers.

### 5.8 Summary

We proposed CheckDP, an integrated tool based on static analysis for automatically proving or disproving that a mechanism satisfies differential privacy. Compared with all existing verification work, CheckDP is the first that provides useful information (i.e., counterexamples) for incorrect mechanisms. CheckDP also simplifies the previous type systems and defers all privacy-related checks to later stages. Both changes are important for automatically generating proofs and counterexamples.

Evaluation shows that CheckDP is able to provide proofs for a number of algorithms, as well as counterexamples for their incorrect variants within 2 to 70 seconds. Moreover, all generated proofs and counterexamples are validated.
Chapter 6

Automated Program Synthesis for Differential Privacy

6.1 Introduction

Most current tools focus on checking the privacy properties of algorithms. For example, verification tools have been developed to mechanically (and sometimes, automatically) prove that a (correct) privacy mechanism satisfies differential privacy [14, 15, 16, 17, 18, 19, 20, 21]. Counterexample detectors for differential privacy [29, 30, 31, 41] can find evidence that an (incorrect) privacy mechanism fails to satisfy its claimed privacy levels. Moreover, a few tools can combine both functionalities: either proving a mechanism is correct or finding a counterexample [37, 42, 43]. While these tools are invaluable for ensuring correctness of privacy mechanisms, they all require a putative differentially private algorithm as a starting point.

Recently, Roy et al. [33] took a step further: they proposed a tool called KOLAHAL for automatically learning an accurate and differentially private mechanism given a mechanism sketch provided by a domain expert. In other words, their approach synthesizes how much noise should be added in pre-specified locations. Note that it does not determine where the noise should be added. It also cannot synthesize mechanisms that use their privacy budget adaptively. An example of such a mechanism is a recently proposed variant of SVT, called Adaptive Sparse Vector with Gap [44]. This mechanism has extra flexibility for saving privacy budget on some queries, allowing it to keep iterating until its privacy budget is exhausted.

In this chapter, we present DPGen, the first fully automated approach that can synthesize an accurate and differentially private program from a given non-private (noiseless) program. Significantly, DPGen employs a novel inference algorithm to automatically generate a mechanism sketch from a non-private program. We formalize the synthesis problem as a constrained optimization problem: maximizing utility while simultaneously satisfying privacy constraints in a transformed version of the mechanism
sketch. DPGen then uses a counterexample-guided synthesis (CEGIS) loop [59] and an optimizer like Particle Swarm Optimization (PSO) [67] to synthesize and optimize the mechanism. Compared with KOLAHAL [33], the new optimization approach is shown to be more efficient. In some cases, KOLAHAL can take 900 to 5460 seconds to synthesize a mechanism, while DPGen can successfully synthesize an equivalent or more accurate version in 10 to 120 seconds.

Moreover, DPGen is equipped with a novel feature called a while-private loop, written as

\[
\text{while-priv } e \text{ do } \ldots
\]

Semantically, the while-private loop (after synthesis) executes \( c \) whenever \( e \) evaluates to \text{true} and \text{as long as the dynamically tracked privacy budget has not been depleted}. Notably, this feature allows DPGen to synthesize sophisticated mechanisms such as Adaptive Sparse Vector with Gap [44] that try to minimize the amount of privacy budget spent in each loop iteration, and hence keep iterating until the privacy budget has been depleted. To the best of our knowledge, DPGen is the first program synthesizer that can automatically generate such sophisticated mechanisms.\(^1\)

We evaluated DPGen on standard benchmarks that consist of various privacy mechanisms. For each privacy mechanism, we removed the randomness in it and asked DPGen to automatically synthesize a differentially private version. In all cases, DPGen was able to synthesize an equivalent or even more accurate version compared with the baseline. For adaptive mechanism that uses while-private loop, program synthesis is more complicated. But DPGen was still able to synthesize private and accurate mechanisms.

In summary, this chapter highlights the following contributions:

1. DPGen, the first fully automated tool that can synthesize an accurate and differentially private mechanism from a noiseless non-private program.

2. A novel inference algorithm that automatically generates a mechanism sketch (i.e., code with noise of unknown scales added to automatically selected program locations) from a non-private program (Section 6.4).

3. A customized CEGIS loop that incrementally optimizes the tentative mechanism while generating its privacy proof (Section 6.5.2).

\(^1\)Note that while-private is a programmer hint that the while loop should be executed in a best-effort way (a hallmark of the sparse vector family of privacy mechanisms) rather than exactly as many times as the non-private version would execute.
4. A novel *while-private* feature that allows DPGen to synthesize adaptive privacy mechanisms (Section 6.5.3).

5. Case studies and experimental comparisons between DPGen and KOLAHAL [33]. In addition to being able to synthesize more programs, DPGen also shows improvements on mechanisms that both approaches can synthesize. In the benchmarks, DPGen generated identical or more accurate mechanisms within a considerably shorter amount of time (Section 6.6).

### 6.2 Background

#### 6.2.1 Particle Swarm Optimization (PSO)

Prior tools using the Randomness Alignment technique (e.g., [14, 15, 37]) focus on privacy only; they model privacy proof as a constraint-solving problem which is solved by an external SMT solver. However, synthesizing DP mechanism is better described as a *constrained optimization* problem: maximizing *utility* among various candidates that have the same overall differential privacy parameter $\epsilon$.

In this chapter, we use Particle Swarm Optimization (PSO) [67] to help with the synthesis. PSO is a meta-heuristic optimization algorithm that is inspired by swarm behaviors such as birds in nature. It deploys a large population of candidate solutions (“particles”) in the search space and the particles move around iteratively to find the best location. For each iteration, each particle updates its position and velocity according to a mathematical formula consisting of its own local best position, the swarms’ best position and its previous velocity. By adopting this strategy, the entire swarm is guided towards the best solutions. PSO makes no assumption about the problem being optimized and is suitable for very large search spaces. This is well suited for our complex, non-differentiable optimization problem, which makes other gradient-based optimization methods inapplicable. Specifically for the synthesis task, each candidate mechanism in the search space corresponds to a particle in PSO, and the instantiations of the sketch mechanism serves as its position. For each iteration, every candidate explores the search space by changing itself slightly according to the current global best candidate (with the best utility), its own local best in history and the amount of changes from previous iterations. The global best solution is returned
function SVTBase (T,N,size: num, q: list num*)
returns (out: list num, bound(ε))
precondition ∀ i. –1 ≤ (q[i]) ≤ 1 ∧ N < size / 5

1. i := 0; count := 0;
2. while (i < size ∧ count < N)
   3. if (q[i] ≥ T) then
      4. out := true::out;
      5. count := count + 1;
   6. else
      7. out := false::out;
   8. i := i + 1;

function SVT (T,N,size: num, q: list num)
returns (out: list num)

1. i := 0; count := 0;
2. \[ \eta_1 := \text{Lap}(3/\epsilon) \]
3. \[ T_* := T + \eta_1; \]
4. while (i < size ∧ count < N)
   5. \[ \eta_2 := \text{Lap}(3N/\epsilon); \]
   6. if (q[i] + \eta_2 ≥ T_*) then
      7. out := true::out;
      8. count := count + 1;
   9. else
      10. out := false::out;
   11. i := i + 1;

Figure 6-1: Sparse Vector Technique.

after a number of iterations.

6.2.2 Sparse Vector Technique (SVT)

In this chapter, we use Sparse Vector Technique (SVT) [28] and its variants as running examples. Given a sequence of queries, SVT tries to find the first \( N \) queries whose query answers are likely\(^2\) to be above a publicly known threshold \( T \). When privacy is not a concern, the pseudo code of SVT’s basic

\(^2\)The uncertainty is introduced by privacy requirements.
functionality is shown in Figure 6-1 (we call it SVTBase). For now, we can safely ignore the function signature. SVTBase checks each exact query answer: it outputs \texttt{true} (resp. \texttt{false}) if the query answer is above (resp. below) the threshold until \( N \) \texttt{true} outputs are produced.

To enforce differential privacy, SVT adds \textit{carefully calibrated} independent Laplace noise both to the threshold (\( T \)) and each query answer (\( q[i] \)). The pseudo code is shown in Figure 6-1 (we call it SVT), where the changes are highlighted. The sampling instruction \texttt{Lap(r)} draws one sample from the Laplace distribution with mean 0 and scale factor of \( r \in \mathbb{R} \). For each query, the mechanism outputs \texttt{true} if the \textit{noisy} query answer is above the \textit{noisy} threshold; otherwise it outputs \texttt{false}. It is well-known that SVT satisfies \( \epsilon \) differential privacy [28].

### 6.3 Overview

#### 6.3.1 Challenges

The goal of this dissertation is to \textit{automatically synthesize} a differentially private program (e.g., function SVT) from a base program that is not necessarily differentially private (e.g., function SVTBase). Like other program synthesis techniques [59, 68], the synthesized program must implement similar functionality to the original program / specification. Since a privacy mechanism injects noise to offer privacy, this can be more precisely stated as: any output of the original program is still possible for the synthesized program.

What makes DPGen distinguished from other program synthesizers is its capability of synthesizing a \textit{private} and \textit{useful} counterpart of the original program:

- **Privacy**: the synthesized program needs to inject sufficient noise in the right places to satisfy pure differential privacy, as formally defined in Definition 2.1.

- **Utility**: the synthesized program needs to carefully calibrate the injected noise to make the randomized outputs useful (i.e., to make the outputs “close” to the ones from the original program). This involves choosing the correct noise scales (including using no noise wherever it is safe to do so).

Next, we highlight the main challenges in both aspects.
function SVT-ALT (T,N,size: num, q: list num) returns (out: list num)

1  i := 0; count := 0;
2  while (i < size ∧ count < N)
3    \eta_2 := \text{Lap} \left( \frac{\text{size}}{\epsilon} \right);
4    if (q[i] + \eta_2 ≥ T) then
5      out := true::out;
6      count := count + 1;
7    else
8      out := false::out;
9    i := i + 1;

Figure 6-2: An alternative way of making SVTBase \( \epsilon \)-private.

**Privacy**  Developing differentially private mechanisms is a nontrivial task: injecting sufficient amount of noise in the right places and then proving correctness is notoriously tricky. For instance, Lyu et al. [12] catalog several incorrect variants of SVT, where each variant slightly modifies the functionality and/or injected noise of function SVT in Figure 6-1 (for now, safely ignore the annotations in the function signature). While the changes are minimal, the incorrect variants fail to meet their claimed differential privacy guarantees. For example, one variant tweaks the mechanism to output the noisy query answer when it is above the threshold. That is, it changes Line 7 of SVT by replacing \text{true} with \( q[i] + \eta_2 \). As a result, it fails to satisfy \( \epsilon \)-differential privacy for any value of \( \epsilon \) [12].

**Utility**  What makes synthesizing differentially private mechanisms even more challenging is that we also need to add as little noise as possible while maintaining the desired privacy levels (otherwise the noisy outputs may not be useful). For example, in the simplest case, if we increase the scale of noise injected at Lines 2 and 5 in SVT (Figure 6-1), the mechanism is still \( \epsilon \)-differentially private. However, the extra randomness reduces the accuracy of SVT. Furthermore, utility is also affected by where the noise is added. For example, an alternative way of making function SVTBase \( \epsilon \)-private is shown in Figure 6-2. Compared with SVT, SVT-ALT does not add any noise to the threshold \( T \); instead, it injects Laplace noise \( \text{Lap} \left( \frac{\text{size}}{\epsilon} \right) \) (rather than \( \text{Lap} \left( \frac{3N}{\epsilon} \right) \)) to each query answer. This provides the same privacy guarantees (SVT and SVT-ALT both satisfy \( \epsilon \)-differential privacy for the same value of \( \epsilon \)).
However, since $N$ is typically much smaller than size (the total number of queries), SVT-ALT injects significantly more noise into its computation.

Handling these kinds of decisions during the synthesis process is a highly non-trivial task and requires deep understanding of the privacy cost introduced by each sampling instruction. For example, SVT and its correct variants [28, 12, 13, 44] have the interesting property that outputting false does not incur any privacy cost (i.e., the costs\(^3\) are only incurred for making the threshold noisy and for outputting true). On the other hand SVT-ALT is too naive and incurs a privacy cost of $\epsilon$/size for each iteration of the while loop (for a total cost of $\epsilon$).

Finally, in many mechanisms (including SVT) and its variants, one needs to decide how to divide up a total privacy budget $\epsilon$ among different parts of the mechanism (i.e., what should the privacy cost of each part of the mechanism be). In the case of SVT, a synthesizer would decide how much of the budget should be consumed by adding noise to threshold $T$ and how much should be consumed by the while loop. This is equivalent to deciding how much noise should be used for the threshold and how much should be used for the noisy query answers. In Figure 6-1, the noise scale for the threshold is $\sigma_1 = 3/\epsilon$ while the noise scale for each query answer is $\sigma_2 = 3N/\epsilon$. However, any choice of $\sigma_1, \sigma_2$ that satisfies $1/\sigma_1 + 2N/\sigma_2 = \epsilon$ will result in $\epsilon$-differential privacy [12]. As shown by Lyu et al. [12], an approximately optimal ratio of $\sigma_1 : \sigma_2$ is $1 : (2N)^{2/3}$.

### 6.3.2 Approach Overview

To synthesize a privacy mechanism, DPGen adds proper amount of noise to the original program. This naturally involves two tasks: (1) finding program locations to add random noise to, and (2) finding

---

\(^3\)The privacy cost of the threshold is $\epsilon/3$ and each of the $N$ true outputs incurs a privacy cost of $2\epsilon/(3N)$.
the amount (scale) of each noise. Accordingly, DPGen synthesizes a privacy mechanism as shown in Figure 6-3.

**Phase 1: Sketch Generation (Section 6.4)** In Phase 1, DPGen generates a *sketch mechanism* with candidate locations for noise. The sketch mechanism might contain more locations for noise than needed, as the unnecessary ones will eventually be optimized away in Phase 2. Moreover, each noise location $\eta_i$ is paired with a scale template $S_i$ which consists of a set of unknown scale holes $\lambda$ to be synthesized in Phase 2. We use $M'(inp, \lambda)$ to denote such a sketch mechanism with unknown scale holes.

**Phase 2: Synthesis Loop (Section 6.5)** Due to the tension between privacy and utility, mechanism synthesis cannot proceed without privacy in mind. Hence, DPGen next generates a transformed relational program with both scale templates $S$ containing holes $\lambda$, and proof templates (in the form of alignments) $\mathcal{A}$ containing holes $\theta$ to be synthesized. Next, DPGen employs a customized CEGIS loop that iteratively refines a candidate mechanism (i.e., an instantiation of $\theta$ and $\lambda$) by generating more and more counterexamples (i.e., inputs that violates privacy constraints).

The CEGIS loop consists of two components. The counterexample generation component starts with a null mechanism (with $\theta = \tilde{0}$ and $\lambda = \tilde{1}$) and first searches for a counterexample (i.e., inputs) that *maximizes* the total number of privacy violations. The reason behind the optimization goal is the following: CEGIS benefits greatly from a good set of counterexamples; intuitively, a counterexample that violates maximum number of privacy constraints serves as better guides than others.

With a set of counterexamples, the mechanism generation component searches for a mechanism (i.e., an instantiation of the mechanism template) that *maximizes* utility while still being private. More specifically, the utility is defined both for privacy and accuracy:

- **Privacy.** A mechanism must be private for all previously seen counterexamples. Hence, any mechanism that is deemed as non-private on counterexamples has a negative utility score.

- **Accuracy.** DPGen is parameterized by either a default utility function (sum of variances), or a user-provided one. The utility function is used as the quality metric of each private candidate.

Once DPGen finds a mechanism where no counterexamples can be found, the CEGIS loop terminates and DPGen sends the mechanism to a verifier (we use CPAChecker [47]). Note that although we did not
Syntax of Source Language

Reals \( r \in \mathbb{R} \)
Booleans \( b \in \{\text{true}, \text{false}\} \)
Vars \( x \in V \)
Linear Ops \( \oplus ::= + \mid - \)
Other Ops \( \otimes ::= \times \mid / \)
Comparators \( \odot ::= <\mid >\mid =\mid \geq \)
Bool Exprs \( \mathbb{B} ::= \text{true} \mid \text{false} \mid x \mid \neg b \mid x_1 \odot x_2 \)
Num Exprs \( n ::= r | x | n_1 \oplus n_2 | n_1 \otimes n_2 | b \mid n_1 : n_2 \)
Expressions \( e ::= n \mid \mathbb{B} \mid e_1 :: e_2 \mid e_1[e_2] \)
Commands \( c ::= \text{skip} | x := e | c_1 ; c_2 \mid \text{return } e \mid \text{if } e \text{ then } (c_1) \text{ else } (c_2) \mid \text{while } e \text{ do } (c) \mid \text{while-priv } e \text{ do } (c) \)

Syntax of Target Language

Rand Vars \( \eta \in H \)
Num Exprs \( n ::= \cdots | \eta \)
Commands \( c^* ::= \text{skip} | x := e | c_1^* ; c_2^* \mid \text{return } e \mid \text{if } e \text{ then } (c_1^*) \text{ else } (c_2^*) \mid \text{while } e \text{ do } (c^*) \mid \eta ::= \text{Lap } (n) \)

Figure 6-4: DPGen: source and target language syntax.

encounter any incorrect synthesized mechanism in our experiments, verification is needed in general as an optimizer might miss a solution when one exists.

6.4 Sketch Generation

As discussed in Section 6.3, DPGen synthesizes a DP mechanism in two phases. In this section, we first show the syntax of its source and target languages. Then, we propose novel algorithms to identify potential violations of privacy in the source code, and then, to inject noise at proper locations to form a program sketch to be further analyzed in Phase 2 (Section 6.5).
6.4.1 Syntax of Source and Target Program

Source Language The syntax of DPGen source code is listed in Figure 6-4. The source language models an expressive imperative language with the following standard features:

- Values of real numbers, Booleans and operations on them;
- Ternary expressions \( b ? n_1 : n_2 \), which returns \( n_1 \) (resp. \( n_2 \)) when \( b \) evaluates to \text{true} (resp. \text{false});
- List of values as well as append (\( :: \)) and projection (\([\]) operations on lists. Note that all lists are initialized to be empty.
- No-op commands (\text{skip}), assignments, sequential commands (\( c_1; c_2 \)), return commands, if branches and while loops.

One novel feature of the source language is a while-private loop written as \text{while-priv} \( e \) \text{do} \( c \); it requests the synthesizer to synthesize an adaptive privacy mechanism (e.g., Adaptive Sparse Vector with Gap [44]) that runs \text{while} \( e \) \text{do} \( c \) until the privacy budget is exhausted. This powerful feature allows the synthesized privacy mechanism to adaptively control the number of outputs based on the remaining privacy budget, in order to increase the amount of queries that they can process. We show how to synthesize the Adaptive Sparse Vector with Gap mechanism in Section 6.5.3.

Finally, the source language requires a few user-provided privacy specifications that the synthesizer should obey, including private inputs and their sensitivity\(^4\), the desired privacy bound (i.e., \( \epsilon \) in \( \epsilon \)-differential privacy), as well as assumptions on the query answers. While we do not formalize the syntax of such specification, we use \text{type}\(^*\) to denote private input of some \text{type}, \text{bound}(\epsilon) to denote the privacy budget, and specify sensitivity on private inputs (\( \widehat{x} \) represents the sensitivity of \( x \)) and other assumptions on inputs as program precondition. For example, the source program \text{SVTBase} in Figure 6-1 specifies that query answers \( q \) are the only private inputs and their sensitivity is 1. Moreover, the mechanism assumes that \( N \) is much smaller than \( \text{size} \), and the goal is to synthesize an \( \epsilon \)-differentially private mechanism.

\(^4\)Determining the sensitivity of queries is crucial to produce an appropriate noise scale. Here, we assume that this information is provided by the user, as the sensitivities of simple queries, such as sum, mean and median, are fairly easy to compute as demonstrated in [28]. For more complex queries, users can either derive manually or use sensitivity analysis tools (e.g., [2, 7]) to calculate sensitivity.
**Target Language** The goal of DPGen is to synthesize a randomized mechanism that both preserves the source program’s semantics and offers \( \epsilon \)-differential privacy (where \( \epsilon \) is annotated in the source program). Hence, the target language (shown in Figure 6-4) is similar to the source language, with a few important changes:

- The target language is probabilistic: it extends the (deterministic) source language with random variables \( \eta \) and sampling commands, written as \( \eta := \text{Lap}(\pi) \).
- The target language excludes the (non-executable) while-private loops; such loops in the source code are replaced by fully synthesized standard loops that terminate the loop whenever the privacy budget is exhausted.

Consider Figure 6-1. Function \( SVT \) is the target program synthesized from the source program \( SVT\text{-Base} \). Note that they are very similar, but function \( SVT \) properly injects noise at various locations to satisfy \( \epsilon \)-differential privacy.

### 6.4.2 Adding Noise Locations to Source Code

The first step of DPGen is to find a set of program locations in the source program where extra noise is needed. In this step, the primary concern is privacy; in other words, the lack of randomness in the source program violates differential privacy. Hence, we use static program analysis to (1) identify where privacy is violated in the source code, (2) infer a set of variables that might require randomness, and (3) instrument the source code to inject noise to the identified variables.

**Identify Violations of Differential Privacy** Recall that DPGen is built on the Randomness Alignment technique (Section 2.2) to reason about privacy. Hence, instead of analyzing properties on distributions directly, as stated in Definition 2.1, we over-approximate “Violations of Differential Privacy” as “Violations of Alignment Requirements”. Recall that randomness alignment requires that when running on a pair of adjacent private inputs, a program will produce identical outputs. Since the source code has no randomness, this requirement can be formalized as the standard non-interference property [69]. Hence, we use a static taint analysis (e.g., [70, 71, 50]) to identify violations in the source code:
• Initially, only the private inputs are tainted.

• The analysis tracks all explicit flows in the program.

• The analysis does not track, but reports all implicit flows, where a tainted value is used in a branch condition.

• The analysis reports all outputs with a tainted value.

For example, since query answers $q$ are the only tainted inputs in SVTBase (Figure 6-1), the taint analysis finds one violation of privacy at Line 3, where the branch condition uses a tainted value $q[i]$. Since the taint analysis is standard, we omit the details here.

**Identify Offending Variables** The static taint analysis returns a set of offending assignments $x := e$ and offending branches $\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$, where $e$ is tainted. We use $E$ to represent the set of expressions that are either on the RHS of offending assignments, or in the branch condition of offending branches. Next, we need to infer a set of variables, that when randomized, will allow randomness alignment to exist on the randomized code. We call such a set of variables offending variables.

Consider the offending branch in our running example:

```
if q[i] ≥ T then ... else ...
```

where $q[i]$ is tainted while $T$ is not. To make the branch outcome identical on two adjacent inputs $q[i] \sim q'[i]$, we can either inject noise to $q[i]$, or to $T$, or to both. While all options can allow the offending branch to be aligned, the difference will show up when we analyze their corresponding utility. For example, adding noise to $T$ is crucial to make SVT useful; intuitively, it allows the noisy $T$ to be reused across different loop iterations, which results in a less noisy program. We defer the discussion on utility to Section 6.5.2.2.

Based on the insight above, we define all variables used in any $e \in E$ as offending variables. Note that by definition, the set of tainted variables is always a subset of offending variables.

**Instrument Source Code with Extra Noise** Finally, DPGen injects noise with unknown scales (to be synthesized in later stages) to the source code. In particular, it injects Laplacian noise both at the definition of an offending variable, as well as right before its corresponding uses in an offending
function SVT-Sketch ($\epsilon$,$T$,$N$,$\text{size:}$num,$q$:list num.$\lambda$:list num) returns (out: list num)

1. $\eta_1 := \text{Lap} \left((\lambda_0 + \lambda_1 \times N + \lambda_2 \times T + \lambda_3 \times \text{size})/\epsilon\right)$
2. $T_0 := T + \eta_1$
3. $i := 0$; count := 0;
4. while ($i < \text{size}$ $\land$ count $< N$)
5.   $\eta_3 := \text{Lap} \left((\lambda_4 + \lambda_5 \times N + \lambda_6 \times T + \lambda_7 \times \text{size})/\epsilon\right)$;
6.   $T_0 := T_0 + \eta_3$
7.   $\eta_2 := \text{Lap} \left((\lambda_8 + \lambda_9 \times N + \lambda_{10} \times T + \lambda_{11} \times \text{size})/\epsilon\right)$;
8.   $q_0 := q[i] + \eta_2$
9. if ($q_0 \geq T_0$) then
10.   out := true::out;
11.   count := count + 1;
12. else
13.   out := false::out;
14.   i := i + 1;

Figure 6-5: Sketch of SVT-Base with extra noise.

command. While adding noise to both locations might seem unnecessary at this point, DPGen eventually uses a utility optimizer (Section 6.5.2.2) to remove unnecessary noise in the code sketch.

Moreover, as the scale of each Laplacian noise is unknown at this point, we replace them with scale templates as follows:

\[
(\lambda_0 + \sum_{v_i \in \mathcal{V}} \lambda_i \times v_i)/\epsilon \text{ with fresh } \lambda_i
\]

where $\mathcal{V}$ contains all non-private function parameters (as making scale private could violate privacy directly by revealing distribution statistics). Return to our running example of SVT, the code sketch with extra noise is shown in Figure 6-5 where all changes are highlighted. Notably, the sketched function explicitly adds scale parameters $\lambda$ (we use $\lambda_i$ instead of $\lambda[i]$ for better readability) as extra inputs to be optimized later. No noise is injected at Line 2 for $q[i]$, essentially an iterator of $q$, as it is not in scope at that point.

Hereafter, we use $M(inp)$ and $M'(inp, \lambda)$ to represent the original program with inputs $inp$ and mechanism sketch with scale parameters $\lambda$ respectively.
6.5 Synthesis and Optimization

In Phase 2, DPGen completes program synthesis with two sub-goals:

- It synthesizes and optimizes the randomness alignment of each sampling instruction; a sampling instruction with alignment 0 implies that the instruction can be removed without violating differential privacy.

- It synthesizes and optimizes the scales \( \lambda \) in the sketch code from Phase 1 to offer good utility.

The main challenge is that instead of synthesizing some privacy proof (as done in prior work with proof synthesis [37, 16]) or optimize scales with given randomness locations (as done in [33]), our goal is to synthesize and optimize both the proof (with fewest randomness locations) and scales.

We first introduce the optimization problem without any while-private loop in source code and assume a default utility function that minimizes sum of variances. Then, we propose a synthesis loop to optimize alignments and scales simultaneously. Finally, we generalize the approach to optimize sketch code with while-private loops and customized utility functions.

6.5.1 Mechanism Synthesis Problem

Reasoning about Privacy To reason about privacy, DPGen uses a syntax-directed transformation from the sketch program to non-probabilistic relational code with explicit alignments and proof obligations (i.e., assertions to ensure privacy). The complete transformation rules are shown in Figure 6-6 and Figure 6-7 for expressions and commands, respectively. For commands, each transformation rule has the following format:

\[ \vdash \Gamma \{ c \rightarrow c' \} \Gamma' \]

where a typing environment \( \Gamma \) tracks for each program variable \( x \) its data type with its distance written as \( \widehat{x} \). Recall that in the Randomness Alignment technique, the distance of a variable is defined as its value difference across two executions on adjacent query answers (Section 2.2). Moreover, \( c \) and \( c' \) are the sketch code and relational code respectively, and the flow-sensitive type system also updates typing environment to \( \Gamma' \) after command \( c \).
Transformation Rules for Expressions with Form $\Gamma \vdash e : B_\varnothing$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash r : \text{num}_0 \mid \text{true}$</td>
<td>(T-Num)</td>
</tr>
<tr>
<td>$\Gamma \vdash b : \text{bool} \mid \text{true}$</td>
<td>(T-Boolean)</td>
</tr>
<tr>
<td>$\Gamma, x : B_\varnothing \vdash x : B_\varnothing$</td>
<td>(T-VarZero)</td>
</tr>
<tr>
<td>$\Gamma, x : B, \vdash x : B_\chi \mid \text{true}$</td>
<td>(T-VarStar)</td>
</tr>
<tr>
<td>$\Gamma \vdash e : \text{bool} \mid C$</td>
<td>(T-NEG)</td>
</tr>
<tr>
<td>$\Gamma \vdash \neg e : \text{bool} \mid C$</td>
<td>(T-NEG)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : B_{\varnothing_1} \mid C_1 \Gamma \vdash e_2 : B_{\varnothing_2} \mid C_2$</td>
<td>(T-Plus)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 \oplus e_2 : B_{\varnothing_1 \oplus \varnothing_2} \mid C_1 \land C_2$</td>
<td>(T-Plus)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 \otimes e_2 : \text{num} \mid C_1 \land C_2 \land (\varnothing_1 = \varnothing_2 = 0)$</td>
<td>(T-Times)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 \otimes e_2 : \text{bool} \mid C_1 \land C_2 \land (e_1 \otimes e_2) \leftrightarrow (e_1 \oplus e_2) \otimes (e_2 + e_2)$</td>
<td>(T-ODot)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : B_{\varnothing_1} \mid C_1 \Gamma \vdash e_2 : \text{list} B_{\varnothing_2} \mid C_2$</td>
<td>(T-Cons)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 :: e_2 : \text{list} B_{\varnothing_1} \mid C_1 \land C_2 \land (\varnothing_1 = \varnothing_2 = 0)$</td>
<td>(T-Cons)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : \text{list} \tau \mid C_1 \Gamma \vdash e_2 : \text{num} \mid C_2$</td>
<td>(T-Index)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1[e_2] : \tau \mid C_1 \land C_2 \land (\tau = 0)$</td>
<td>(T-Index)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : \text{bool} \mid C_1 \Gamma \vdash e_2 : B_{\varnothing_1} \mid C_2 \Gamma \vdash e_3 : B_{\varnothing_2} \mid C_3$</td>
<td>(T-Select)</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 ? e_2 : B_{\varnothing_1} \mid C_1 \land C_2 \land C_3 \land (\varnothing_1 = \varnothing_2)$</td>
<td>(T-Select)</td>
</tr>
</tbody>
</table>

Figure 6-6: Program transformation rules for expressions.

Most importantly, the transformation inserts assertions to ensure the following (informal) soundness property:

if $M'(\text{inp}, \lambda)$ is transformed to $M''(\text{inp}, \tilde{\text{inp}}, \text{sample}, \theta, \lambda)$, then

$$\exists \theta, \lambda. \forall \text{inp}, \tilde{\text{inp}}, \text{sample}. \text{all assertions in } M'' \text{ pass}$$

$$\implies M'(\text{inp}, \lambda) \text{ is differentially private}$$

The most interesting transformation rule is for the sampling commands in the sketch code (T-LAPLACE) shown in Figure 6-7. It performs the following important tasks:

1. Each sampling command is replaced by a non-probabilistic counterpart ($\eta := \text{sample}[\text{idx}]; \text{idx} :=$
Transformation Rules for Commands with Form $\vdash \Gamma \{c \rightarrow c'\} \Gamma'$

$$\begin{align*}
\Gamma \vdash e : B_n \mid C & \quad \langle d, c \rangle = \begin{cases} (0, \text{skip}), & \text{if } n = 0, \\ (e, \tilde{x} := v), & \text{otherwise} \end{cases} & (T-\text{ASGN}) \\
\vdash \Gamma \{x := e; \rightarrow \text{assert } (C); x := e; c\} \Gamma [x \mapsto B_d] & (T-\text{SEQ}) \\
\vdash \Gamma \{c_1 \rightarrow c'_1\} \Gamma_1 \quad \vdash \Gamma \{c_2 \rightarrow c'_2\} \Gamma_2 & (T-\text{SEQ}) \\
\vdash \Gamma \{c_1 ; c_2 \rightarrow c'_1 ; c'_2\} \Gamma_1 \Gamma_2 & (T-\text{SEQ}) \\
\vdash \Gamma \{\text{return } e \rightarrow \text{assert } (C \land n = 0); \text{return } e\} \Gamma & (T-\text{RETURN}) \\
\vdash \Gamma \{\text{skip } \rightarrow \text{skip}\} \Gamma & (T-\text{SKIP}) \\
\vdash \Gamma \{\text{while } e \text{ do } c \rightarrow c_x; \{\text{while } e \text{ do } (\text{assert } ((e, \Gamma)^o); c'; c''))\}\} \Gamma \sqcup \Gamma_f & (T-\text{WHILE}) \\
\vdash \Gamma \{\text{if } e \text{ then } c_1 \text{ else } c_2 ; \text{if } e \text{ then } (\text{assert } ((e, \Gamma)^o); c'; c'') \text{ else } (\text{assert } ((e, \Gamma)^o); c'_1 ; c''))\} \Gamma_1 \sqcup \Gamma_2 & (T-\text{IF}) \\
\mathcal{A} = \text{GenerateTemplate}(\Gamma, \text{All Assertions}) \quad c_a = \text{assert } (((\eta + \mathcal{A}(\eta_\eta) = (\eta + \mathcal{A}(\eta_\eta) = \eta_1 = \eta_2)) & (T-\text{LAPLACE}) \\
\vdash \Gamma \{\eta \leftarrow \text{Lap } \mathcal{S} ; c_a; \eta := \text{sample}[idx]; idx := idx + 1; v_e := v_e + |\mathcal{A}| / |\mathcal{A}_\tilde{x} := \mathcal{A}_\tilde{x} | \Gamma[\eta \mapsto \text{num}_a] &
\end{align*}$$

Transformation Rules for Merging Environments

$$\begin{align*}
\Gamma_1 \sqcup \Gamma_2 & \quad c = \{\tilde{x} := 0 \mid \Gamma_1(x) = \text{num}_a \land \Gamma_2(x) = \text{num}_a\} \\
\Gamma_1, \Gamma_2 & \Rightarrow c
\end{align*}$$

Figure 6-7: Program transformation rules for commands. $\mathcal{S}$ represents the scale template instrumented in Phase 1. Distinguished variable $v_e$ and assertions are added to ensure differential privacy.

1. $\text{idx } + 1$ that reads a sample from the instrumented function input $\text{sample}$.

2. An alignment template (i.e., $\mathcal{A}$) is generated for each sampling command; each template contains a few holes, i.e., $\theta$, which is also instrumented as function input. Here, we reuse the $\text{GenerateTemplate}$ function proposed by CheckDP (Algorithm 1 in Chapter 5). Intuitively, the alignments serve as a way to satisfy all inserted assertions in the transformed program. To do so, each alignment template $\mathcal{A}_i$ for random variable $\eta_i$ contains distance variables of program variables that (1) appear in assertions, and (2) depend on $\eta_i$. Hence, $\text{GenerateTemplate}$ takes the typing environment at the sampling command and all assertions as input, and properly calculates an alignment template, a linear function on a set of relevant distance variables as stated above.

3. The transformed code uses a distinguished variable $v_e$ to track the overall privacy cost. Moreover,
\(v_e\) is updated to \(v_e + |A|/S\), where \(S\) is the scale template instrumented in Phase 1. As discussed in Section 2.2, the update soundly accounts for the privacy cost of aligning the Laplace noise with alignment \(A\) and scale \(S\).

4. Assertions are inserted in the transformed code to ensure the (informal) soundness property stated above. In particular, it inserts an assertion \(c_a\) that checks if the alignment function \(\phi(r) = r + A\) is injective (i.e., \(\forall r_1, r_2. \phi(r_1) = \phi(r_2) \implies r_1 = r_2\)). This a fundamental requirement of alignment-based proof [14].

For example, the transformed program of the sketch mechanism in Figure 6-5 is shown in Figure 6-8 with the instrumented code highlighted. Here, each random variable \(\eta_i\) is paired with a corresponding alignment template \(A_i\) computed by GenerateTemplate:

- \(A_1: \theta_0\)
- \(A_2: (\Omega \ ? \ \theta_1 + \theta_2 \times \tilde{T}_b + \theta_3 \times \tilde{q}[i] + \theta_4 + \theta_5 \times \tilde{T}_b + \theta_6 \times \tilde{q}[i])\)
- \(A_3: (\Omega \ ? \ \theta_7 + \theta_8 \times \tilde{T}_b + \theta_9 \times \tilde{q}[i] + \theta_{10} + \theta_{11} \times \tilde{T}_b + \theta_{12} \times \tilde{q}[i])\)

where \(\Omega\) represents the branch condition at Line 13. Note that the privacy cost of each alignment is soundly tracked at Lines 3, 8 and 10. Moreover, the distances of variables (e.g., \(T_b\) and \(q_b\)) are properly updated after each assignment. Finally, the transformed code contains assertions to ensure that (1) two related execution of the sketch mechanism will follow the same control flow (e.g., Lines 14 and 18); (2) The distances of output expressions must be zero (not present in Figure 6-8 since the output values are already zero-distance literals; and (3) the overall privacy cost of the program does not exceed the privacy budget (e.g., Line 21).

Since the other transformation rules are mostly identical to those introduced in CheckDP (Chapter 5) and the soundness property is a direct implication of Theorem 5.2 in Chapter 5, we omit the formal statement of the soundness property and its proof in this dissertation.

**Reasoning about Utility** Note that utility is a property of an instantiation of the mechanism sketch (i.e., fully synthesized program with concrete scales). Hence, reasoning about utility is relatively easy on the mechanism sketch \(M'(inp, \lambda)\). The only interesting part is that utility computation should also take
function Transformed SVT \((T, N, \text{size}, q, \tilde{q}, \text{sample}, \theta, \lambda)\) returns \(\text{out}\)

1. \(v_e := 0; \ idx = 0;\)
2. \(\eta_1 := \text{sample}[\ idx]; \ idx := \ idx + 1; \ \hat{\eta}_1 := \mathcal{A}_1;\)
3. \(v_e := |\mathcal{A}_1|/S_1;\)
4. \(T_0 := T + \eta_1; \ \tilde{T}_0 := \hat{\eta}_1;\)
5. \(\text{count} := 0; \ i := 0;\)
6. while \((\text{count} < N \land i < \text{size})\)
7. \(\eta_3 := \text{sample}[\ idx]; \ idx := \ idx + 1; \ \hat{\eta}_3 := \mathcal{A}_3;\)
8. \(v_e := v_e + |\mathcal{A}_3|/S_3;\)
9. \(\eta_2 := \text{sample}[\ idx]; \ idx := \ idx + 1; \ \hat{\eta}_2 := \mathcal{A}_2;\)
10. \(v_e := v_e + |\mathcal{A}_2|/S_2;\)
11. \(T_0 := T + \eta_3; \ \tilde{T}_0 := \tilde{T}_0 + \hat{\eta}_3;\)
12. \(q_\circ := q[i] + \eta_2; \ \tilde{q}_\circ := \tilde{q}[i] + \hat{\eta}_2;\)
13. if \((q_\circ \geq T_0)\) then
14. \(\text{assert} \ (q_\circ + \tilde{q}_\circ \geq T_0 + \tilde{T}_0);\)
15. \(\text{out} := \text{true}:\text{out};\)
16. \(\text{count} := \text{count} + 1;\)
17. else
18. \(\text{assert} \ (\neg(q_\circ + \tilde{q}_\circ \geq T_0 + \tilde{T}_0));\)
19. \(\text{out} := \text{false}:\text{out};\)
20. \(i := i + 1;\)
21. \(\text{assert} \ (v_e \leq \epsilon);\)

Figure 6-8: Transformed mechanism of SVT-Sketch by DPGen. The instrumented parts are underlined. For better readability, the proof and scale templates are represented by \(\mathcal{A}_i\) and \(S_i\), respectively.

into account the alignments \(\theta_i\), as a random variable with \(\theta_i = 0\) implies that the variable is unnecessary from the privacy perspective; hence, it will be removed in the final synthesized code.

In general, the particular metrics of utility might be application- and data-specific. DPGen is designed to be modular: users can plug in their customized utility metrics, and even sample data to optimize the utility of the synthesized privacy mechanism. Hence, in general, DPGen is parameterized by a utility function \(\text{Utility}(M', \theta, \lambda)\), where \(M'\) is mechanism sketch and \(\theta, \lambda\) are the synthesized alignments and scales respectively. By default, DPGen uses the sum of variances of all random variables.
as the utility function (note that DPGen currently only supports Laplace noise):

$$ \text{Utility}(M', \theta, \lambda) = - \left( \sum_{\{S_i: A_i \neq 0\}} 2S_i^2 \right) $$

where $A_i$ and $S_i$ denote the synthesized scale and alignment for random variable $\eta_i$. As discussed earlier, we explicitly exclude the ones with 0 alignments, since they are unnecessary.

Note that to compute utility based on the default utility function, there is no need to execute $M'$. Hence, synthesizing privacy mechanisms with the default utility function is very efficient. Moreover, despite its simplicity, it allows us to synthesize many privacy mechanisms (Section 6.6). For now, we assume the default utility function is in use; how to synthesize with more complicated utility function is deferred to Section 6.5.3.

### 6.5.2 Mechanism Optimization Problem

Recall that the goal of DPGen is to generate an *accurate* and *private* mechanism. That is, for a search space of alignment holes $\Theta$ and scale holes $\Lambda$, the constrained optimization problem is defined follows:

$$ \max_{(\theta, \lambda) \in \Theta \times \Lambda} \text{Utility}(M', \theta, \lambda) $$

s.t. $\text{Vinp. all assertions in } M'' \text{ pass}$

To find alignment holes $(\theta)$ and scale holes $(\lambda)$ according to the optimization problem above, DPGen

---

5This is inspired by Lyu et al. [12] who derived the approximately optimal budget allocation of SVT by minimizing the variance of the branch (Line 3 in Figure 6-1).
uses a customized Counterexample-Guided Inductive Synthesis (CEGIS) [59] loop, as illustrated in Figure 6-9. Each synthesis iteration contains two steps:

- With a candidate mechanism (initialized with null mechanism of $\theta_0 = \bar{0}, \lambda_0 = \bar{1}$), the “counterexample generation” component tries to find inputs $inp$ that “break” the privacy requirements (i.e., assertion violations in $M''$).

- With a set of counterexamples seen so far, the “mechanism generation” component synthesizes a privacy mechanism by optimizing the utility objective function (we use PSO as a black-box optimization technique in this chapter) while satisfying all previously-generated counterexamples.

The CEGIS loop terminates when no counterexamples can be generated; then, the final privacy mechanism is returned.

Compared with the “bi-directional” search loop of CheckDP (Chapter 5) that improves both privacy proof and counterexamples simultaneously, the CEGIS loop in Figure 6-9 is more standard, as there is no need to improve counterexamples for DPGen. Hence, the use of “bi-directional” CEGIS loop is not necessary.

**Discussion on Soundness** Note that since most optimizers (including PSO [67] that DPGen uses) are unsound (i.e., they might miss a solution when one exists), the synthesized privacy program might be (in rare cases) non-private. To ensure soundness, the synthesized mechanism can be further verified by sound tools like CheckDP (Chapter 5). If verification fails, the counterexamples generated from CheckDP can be passed back to the CEGIS loop to continue the search. In practice, we did not experience any such unsound cases by running separate verification passes in CheckDP; we leave the integration of DPGen and CheckDP as future work.

**6.5.2.1 Counterexample Generation**

Given a candidate mechanism instantiated with some $\theta, \lambda$, as well as a transformed mechanism with explicit alignments $M''(inp, \widehat{inp}, sample, \theta, \lambda)$, a counterexample $C$ is defined as a solution of the following term:

$$\exists inp, \widehat{inp}, sample. \text{ some assertions in } M''(inp, \widehat{inp}, sample, \theta, \lambda) \text{ fail.}$$
We note that this naive definition treats all counterexamples equally: two distinct counterexamples which violate 1 and 100 assertions respectively are both acceptable. To quantify and optimize the qualities of counterexamples (for better performance), we slightly modify the mechanism $M'$ to return the total number of assertion violations and use an optimizer to find a counterexample according to the following metric:

$$\max_{\text{inp, inp, sample}} M''(\text{inp, inp, sample, } \theta, \lambda)$$

Consider the transformed program of our running example in Figure 6-8 with a null mechanism $(\theta = 0, \lambda = 1)$ for bootstrapping the process. The optimizer tries to find a counterexample that fails as many assertions as possible. Since no alignments are set to offset $q[i]$ (the differences introduced by the query variable $q[i]$) in the assertions, a counterexample is found by making all queries fall in the true branch (i.e., query answers $q[i]$ are all above the threshold $T$). Suppose later, an improved alignment, which properly aligns the branch by $\tilde{q}[i]$, is fed in, which makes the false branch also incur a privacy loss. Therefore a counterexample will then be generated with query answers below the threshold, to make privacy cost exceed the total privacy budget (the last assertion in code).

6.5.2.2 Mechanism Generation

In general, mechanism generation runs on both the transformed program $M''$ and the sketch mechanism $M'$ as follows:

- For any candidate solution (of $\theta, \lambda$) that fails to satisfy any privacy constraint in $M''$ given any previously-generated counterexample, we assign a negative utility score to the solution.

- Otherwise, we use the utility function $\text{Utility}(M', \theta, \lambda)$ as its utility score.

Based on the utility scores defined above, DPGen uses an optimizer to find a privacy mechanism that optimizes the utility function while remaining differentially private.

Returning to our running example. The initial few discovered counterexamples likely include ones that go to different branches to cover all code paths. They can serve as good guides to lead the optimizer towards finding a more general solution, by aligning true and false branch differently, using a conditional alignment in the form of $\Omega ? \bullet : \bullet$, as other solutions will result in a negative utility score since they violate privacy.
function AdaptiveSVT-Base (T,N, size, \( \sigma \)) returns (out: list num, \( q \): list num')

precondition \( \forall i. -1 \leq (q[i]) \leq 1 \)

1. \( i := 0; \)
2. while-private (i < size)
3. \( \text{if } (q[i] - T \geq \sigma) \text{ then} \)
4. \( \text{out} := (q[i] - T)::\text{out}; \)
5. \( \text{else} \)
6. \( \text{if } (q[i] - T \geq 0) \text{ then} \)
7. \( \text{out} := (q[i] - T)::\text{out}; \)
8. \( \text{else} \)
9. \( \text{out} := \emptyset::\text{out}; \)
10. \( i := i + 1; \)

Figure 6-10: AdaptiveSVT-Base, while-private feature is used to enable the synthesis of adaptive mechanisms.

Among the solutions that do satisfy all privacy constraints, the mechanism generation component ranks them based on their utility scores. Here, a solution that assigns a large noise (e.g., \( size/\epsilon \)) to the queries, although private, will have smaller utility scores than one which assigns \( 3N/\epsilon \) (since \( N \geq size/5 \) in precondition). Moreover, a solution that assigns three random variables (two for the threshold, and one for the queries) will be less favorable due to larger sum of variances. This shows the power of our utility metric function in selecting good candidate solutions.

6.5.3 Handling while-priv Loop and User-Provided Utility Function

Next, we explore the full-fledged version of DPGen, with advanced features of while-private loop and user-provided utility function. We use a recently proposed variant of SVT that we call AdaptiveSVT (i.e., Adaptive Sparse Vector with Gap in [44]) as an example; its pseudo-code without noise is shown in Figure 6-10. Compared with SVT, there are three major changes:

- The mechanism uses while-private loop (Line 2) to request the synthesizer to adaptively answer as many queries as possible (the input \( N \) specifies the minimum number of above-threshold queries that the algorithm should output).
function `ADAPTIVE_SVT(T,N,size,\sigma):\text{num,q: list num,}`

returns (out: list num)

precondition $\forall i. -1 \leq (q[i]) \leq 1$

\[
\begin{align*}
    i & := 0; \\
    \eta_1 & := \text{Lap}(4/\varepsilon); \\
    v_e & := v_e + 4/\varepsilon; \\
    T_{\leq 1} & := T + \eta_1; \\
    \text{while } (i < \text{size} \land v_e \leq \varepsilon - 2\varepsilon/(2N + 3)) & \text{ do} \\
    \quad \eta_2 & := \text{Lap}((4N + 6)/\varepsilon); \\
    \quad v_e & := v_e + (\Omega_{Top} ? 2 : 0) \times \varepsilon/(4N + 6); \\
    \quad q_{\leq 1} & := q[i] + \eta_2; \\
    \quad \text{if } (q_{\leq 1} - T_{\leq 1} \geq \sigma) & \text{ then} \\
    \quad \quad \text{out} := (q_{\leq 1} - T_{\leq 1})::\text{out}; \\
    \quad \text{else} & \text{ do} \\
    \quad \quad \eta_3 & := \text{Lap}((2N + 3)/\varepsilon); \\
    \quad \quad v_e & := v_e + (\Omega_{Middle} ? 2 : 0) \times \varepsilon/(2N + 3); \\
    \quad \quad q_{\leq 2} & := q[i] + \eta_3; \\
    \quad \quad \text{if } (q_{\leq 2} - T_{\leq 1} \geq 0) & \text{ then} \\
    \quad \quad \quad \text{out} := (q_{\leq 2} - T_{\leq 1})::\text{out}; \\
    \quad \quad \text{else} & \text{ do} \\
    \quad \quad \quad \text{out} := 0::\text{out}; \\
    \quad \quad i & := i + 1; \\
\end{align*}
\]

Figure 6-11: Synthesized AdaptiveSVT based on AdaptiveSVT-Base. $\Omega_{Top}$ and $\Omega_{Middle}$ stand for the branch condition at Line 9 and Line 15, respectively.

- The mechanism partitions query answers into three ranges: $(-\infty, T)$, $[T, T + \sigma)$ and $[T + \sigma, +\infty)$ and requests DPGen to automatically allocate the total privacy budget among queries in each range.

- When $q[i] \geq T$, the mechanism releases the gap between $q[i]$ and $T$, instead of a constant.

Overall, the mechanism improves over SVT since it can use less privacy budget (i.e., add more noise) for queries that are much larger than the threshold $T$ (i.e., in range $[T + \sigma, +\infty)$), in order to increase the amount of queries that it can process. Moreover, it is shown that the gap information can be released for free [44].

From program synthesis perspective, it poses two challenges for DPGen: (1) to synthesize executable code for while-private loop, and (2) to adopt a user-specified utility function.
Synthesizing while-priv Loop  

In order to properly handle the synthesis of while-priv loops, a new transformation rule (T-WHILE-PRIV) (Figure 6-12) should be added to the rules (Figure 6-7). Recall that in the transformed program $M''$, there is an distinguished variable $v_\epsilon$ that tracks the consumed privacy cost at each program point. The transformation of while-private loop uses $v_\epsilon$ to ensure that the loop terminates if $v_\epsilon$ might exceed $\epsilon$ after one more iteration: it inserts an unknown bound on the privacy cost of running one iteration ($\bigcirc$) and ensures that the actual cost of each iteration never exceeds the bound with the assertion inserted at the end. We note that while-private (while-priv) is a new feature of DPGen; it enables DPGen to automatically infer and even optimize the loop termination conditions that are previous manually annotated in CheckDP (Chapter 5).

Discussion on the Soundness of while-priv  

Although while-priv is a new feature of DPGen, we note that this feature is transformed to a normal while loop by the transformation rule in Figure 6-12. By construction, the unknown bound on the privacy cost of each loop iteration ($\bigcirc$) is sound. Moreover, as a synthesized mechanism only contains normal while loops, a synthesized mechanism can further be verified by tools like CheckDP.

User-Specified Utility Function  

Consider the default utility function that minimizes the sum of variances of all random variables (Equation 6.1). A solution that outputs no queries at all always beats other solutions since it injects no noise ($\text{Utility} = 1$). However, the solution fails the requirement of outputting at least $N$ queries in total, where $N$ is a parameter of the mechanism. Therefore, a more informative utility function is required for Adaptive SVT.

Recall that the family of SVTs are designed to report whether a query answer is above a certain threshold or not. Hence, a natural utility measurement is the number of true positives and false positives of the above-threshold queries. Moreover, the design of Adaptive SVT assumes that many queries are well-above the threshold; this allows mechanism to add relatively large noise to the outliers without
impacting number of false positives. Finally, by definition, the synthesized privacy mechanism should output at least $N$ queries in total, where $N$ is a parameter of the mechanism.

Hence, we use a sample input $inp_{ex}$ where many queries are well-above the threshold, create a modified sketch mechanism $M'_\theta$ that removes $\eta_i$ from $M'$ whose alignment is 0, and returns the number of true positives ($#tp$) and false positives ($#fp$). Hence, the user-specified utility function is defined as follows:

$$\text{Utility}(M', \theta, \lambda) = (#tp - #fp) - p \times \max(N - (#tp + #fp), 0)$$

where $p$ is the penalty of outputting less than $N$ outputs, which we set as 1 to guide the search to favor a solution that answers at least $N$ above-threshold queries.

**Choice of Utility Functions** The quality of the synthesized mechanism is dependent on the quality of the utility function, as the latter defines “utility” in the search. In general, a proper utility function of a privacy mechanism might be both data- and application-specific, such as the data- and application-specific utility function that we derived for Adaptive SVT. Nevertheless, for a variety of mechanisms, as showcased in our evaluation, the default utility function (i.e., the sum of variances of all random variables) already allows DPGen to synthesize high quality privacy mechanisms.

### 6.6 Implementation and Evaluation

We implemented a prototype\(^6\) of DPGen in Python. The prototype uses the pyswarms package [72] for PSO optimization. For each component in the CEGIS loop (Figure 6-9), we run the optimization for 500 iterations. To speedup searching, DPGen stops early if the best value stays within tolerance $t = 1$ for 50 iterations. By default, the search space for each hole in the alignments and scales is set to $[-10, 10]$ and $[0, 10]$ respectively. This is chosen based on the typical values of those parameters in correct privacy mechanisms. Moreover, the number of query answers is set to 100. DPGen automatically expands the search space for the holes and the number of query answers until a mechanism is successfully generated.

We note that the use of the optimizer is to *discover* a solution; the generated mechanism is eventually

Table 6-1: Synthesized random variables with corresponding alignment proof. $\Omega_*$ stands for the branch condition in each mechanism. Unnecessary random variables that are removed in the optimization are omitted.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>$\eta_1$</th>
<th>$\eta_2$</th>
<th>$\eta_3$</th>
<th>Time (s)</th>
<th>KOLAHAL [33]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReportNoisyMax</td>
<td>$2/e$</td>
<td>$\Omega_{\text{ReportNoisyMax}}$ $\geq 0$</td>
<td>N/A</td>
<td>N/A</td>
<td>120</td>
</tr>
<tr>
<td>PartialSum</td>
<td>$1/e$</td>
<td>$-\text{sum}$</td>
<td>N/A</td>
<td>N/A</td>
<td>10</td>
</tr>
<tr>
<td>SmartSum</td>
<td>$2/e$</td>
<td>$-\text{sum} - q[i]$</td>
<td>$2/e$</td>
<td>$-q[i]$</td>
<td>25</td>
</tr>
<tr>
<td>SVT</td>
<td>$3/e$</td>
<td>$1$</td>
<td>$3N/e$</td>
<td>$\Omega_{\text{SVT}}$ $\geq 0$</td>
<td>29</td>
</tr>
<tr>
<td>SVT-Inverse</td>
<td>$3/e$</td>
<td>$-1$</td>
<td>$3N/e$</td>
<td>$\Omega_{\text{SVT}}$ $\geq 0$</td>
<td>N/A</td>
</tr>
<tr>
<td>SVT-All</td>
<td>N/A</td>
<td>N/A</td>
<td>$\text{size}$/e</td>
<td>$\Omega_{\text{SVT}}$ $\geq 1$</td>
<td>N/A</td>
</tr>
<tr>
<td>SVT (N = 1)</td>
<td>2.587343/e</td>
<td>1</td>
<td>3.259884/e</td>
<td>$\Omega_{\text{SVT}}$ $\geq 0$</td>
<td>N/A</td>
</tr>
<tr>
<td>GapSVT</td>
<td>$3/e$</td>
<td>$1$</td>
<td>$3N/e$</td>
<td>$\Omega_{\text{SVT}}$ $\geq 0$</td>
<td>N/A</td>
</tr>
<tr>
<td>NumSVT</td>
<td>$4/e$</td>
<td>$1$</td>
<td>$4N/e$</td>
<td>$\Omega_{\text{SVT}}$ $\geq 2$</td>
<td>N/A</td>
</tr>
<tr>
<td>SVTWhilePriv</td>
<td>$3/e$</td>
<td>$1$</td>
<td>$3N/e$</td>
<td>$\Omega_{\text{SVT}}$ $\geq 2$</td>
<td>N/A</td>
</tr>
<tr>
<td>AdaptiveSVT</td>
<td>$4/e$</td>
<td>$1$</td>
<td>$(4N + 6)/e$</td>
<td>$\Omega_{\text{AdaptiveSVT}}$ $\geq 0$</td>
<td>(2N + 3)/e</td>
</tr>
</tbody>
</table>

$\Omega_{NM} = q_0 > bq \land i = 0$  
$\Omega_{SVT} = q_1 \geq T_0$  
$\Omega_{Top} = q_0 - T_0 \geq \sigma$  
$\Omega_{Middle} = q_2 - T_0 \geq 0$

* The ideal solution was ranked $4^{th}$ among the candidates generated by KOLAHAL.

verified by an off-the-shelf sound verifier CPAChecker [47] with arbitrary array lengths. Moreover, to speed up the synthesis of adaptive mechanisms, we split the mechanism sketch into multiple sketches each with a unique combinations of different random variable locations, and run all sketches in parallel. To make the generated mechanism easier to read and more friendly for off-the-shelf verifier, we round up the scales of generated mechanism to nearest integer. However, this can be switched off if the user wants a more refined mechanism.

We evaluate DPGen on an Intel® Xeon® E5-2620 v4 CPU machine with 64 GB memory. Table 6-1 lists the synthesized scales, alignments and synthesis time for each mechanism that we introduce next.

6.6.1 Case Studies

To illustrate the expressiveness of DPGen and its capability of synthesizing privacy mechanisms of different characteristics, we used a standard benchmark as seen in prior works [15, 37, 29, 30, 16], including SVT under different conditions, other variants of SVT such as NumSVT and GapSVT, the Report Noisy Max mechanism [28], Partial Sum and Smart Sum [56]. All of the mechanisms that DPGen synthesized are proved to satisfy $\varepsilon$-differential privacy.

SVT Under Different Conditions  As discussed earlier (Section 6.3.1), the SVT-Base program can be made private in multiple ways, and its utility might depend on the characteristics of the data being
function SVTBASE-INV (T,N,size: num, q: list num*)
returns (out: list num), bound(ε)
precondition ∀ i. −1 ≤ (q[i]) ≤ 1 ∧ N < size / 5

1. i := 0; count := 0;
2. while (i < size ∧ count < N)
3.   if (q[i] ≥ T) then
4.     out := true::out;
5.   else
6.     out := false::out;
7.     count := count + 1;
8.     i := i + 1;

1. v_ε := 0; idx = 0;
2. η_1 := sample[idx]; idx := idx + 1; ̂η_1 := A_1;
3. v_ε := |A_1|/S_1;
4. T₀ := T + η_1; ̂T₀ := ̂η_1;
5. count := 0; i := 0;
6. while (count < N ∧ i < size)
7.   η_2 := sample[idx]; idx := idx + 1; ̂η_2 := A_2;
8.   v_ε := v_ε + |A_2|/S_2;
9.   η_3 := sample[idx]; idx := idx + 1; ̂η_3 := A_3;
10. v_ε := v_ε + |A_3|/S_3;
11. T₀ := T + η_2; ̂T₀ := ̂T₀ + ̂η_2;
12. q₀ := q[i] + η_3; ̂q₀ := ̂q[i] + ̂η_3;
13. if (q₀ ≥ T₀) then
14.   assert (q₀ + ̂q₀ ≥ T₀ + ̂T₀);
15.   out := q[i]::out;
16. else
17.   assert (¬ (q₀ + ̂q₀ ≥ T₀ + ̂T₀));
18.   out := false::out;
19.   count := count + 1;
20. i := i + 1;
21. assert (v_ε ≤ ε);

Figure 6-13: SVT-Inverse and its transformed code.

analyzed as well.

For example, the standard SVT mechanism makes the use of the fact that the number of above-
threshold queries to answer ($N$) is relatively small (hence the name “sparse vector”). This is specified by the precondition $N < \text{size}/5$ in the function signature. Given this assumption on data, DPGen successfully synthesizes the privacy mechanism shown in Figure 6-1, which is the standard SVT mechanism.

In the SVT-All case, we change the assumption to be most queries answers are above the threshold. Under this assumption, the standard SVT is no longer preferred, as intuitively, the privacy cost paid for the threshold can no longer be offset by its gain from paying no cost for the below-threshold queries. As expected, DPGen synthesizes a privacy mechanism that only injects noise to query answers but not to the threshold, which is the same as the mechanism shown in Figure 6-2.

In the SVT-Inverse case, we flip SVT to answer at most $N$ below-threshold queries, rather than to answer at most $N$ above-threshold queries. Accordingly, the same precondition $N < \text{size}/5$ in the function signature now specifies that the number of below-threshold queries to answer is relatively small. Not surprisingly, DPGen successfully synthesizes the dual of standard SVT, with flipped alignments on the true and false branches but the same scales and random variables.

**Numerical SVT and Gap SVT** Numerical Sparse Vector Technique [28], along with Gap Sparse Vector Technique [15], are variants of the standard SVT and have served as benchmark algorithms in ShadowDP (Chapter 4) and CheckDP (Chapter 5). They all extend the standard SVT in smart ways to release extra information without sacrificing more privacy. For completeness, their pseudo-codes and the transformed codes by DPGen are listed in Figure 6-14 and Figure 6-15, respectively. DPGen is able to synthesize the standard private version of them from sketch programs without noises.

**Finding Approximately Optimal Budget Allocation For SVT** As shown in [12], the approximately optimal budget allocation between the threshold and queries for SVT is $1 : (1 + (2N)^{\frac{1}{2}})$. Although DPGen currently lacks the ability to solve the optimization problem with a symbolic $N$, we analyze a case where input $N$ of SVT is fixed to 1. Also, we disabled integer rounding for synthesizing the approximately optimal allocation for this particular instance of SVT. DPGen is able to synthesize a solution with scale $2.587430/\epsilon$ on $\eta_1$, the noise added to the threshold, and scale $3.259844/\epsilon$ on $\eta_2$, the noise added to each query answer; while the approximately optimal ones are $2.587401/\epsilon$ and $3.259960/\epsilon$ respectively when $N = 1$. 

function \texttt{NumSVT} (T, N, size: num, q: list num*)
returns (out: list num), \texttt{bound}(\epsilon)

precondition \forall i. -1 \leq (\overline{q}[i]) \leq 1 \land N < size / 5

1 count := 0; i := 0;
2 while (count < N \land i < size)
3 \textbf{if } (q[i] \geq T) \textbf{ then}
4 \hspace{1em} out := (q[i])::out;
5 \hspace{1em} count := count + 1;
6 \textbf{else}
7 \hspace{1em} out := false::out;
8 \hspace{1em} i := i + 1;

function \texttt{Transformed NumSVT} (T, N, size, q, \overline{q}, \texttt{sample}, \theta, \lambda)
returns (out)

1 \hspace{1em} v_e := \emptyset; \hspace{1em} idx = 0;
2 \hspace{1em} \eta_1 := \texttt{sample}[idx]; \hspace{1em} idx := idx + 1; \hspace{1em} \widehat{\eta}_1 := \mathcal{A}_1;
3 \hspace{1em} v_e := |A_1|/S_1;
4 \hspace{1em} T_0 := T + \eta_1; \hspace{1em} \widehat{T}_0 := \widehat{\eta}_1;
5 \hspace{1em} count := 0; \hspace{1em} i := 0;
6 while (count < N \land i < size)
7 \hspace{1em} \eta_4 := \texttt{sample}[idx]; \hspace{1em} idx := idx + 1; \hspace{1em} \widehat{\eta}_4 := \mathcal{A}_4;
8 \hspace{1em} v_e := v_e + |A_4|/S_4;
9 \hspace{1em} T_0 := T + \eta_4; \hspace{1em} \widehat{T}_0 := \widehat{T}_0 + \widehat{\eta}_4;
10 \hspace{1em} \eta_2 := \texttt{sample}[idx]; \hspace{1em} idx := idx + 1; \hspace{1em} \widehat{\eta}_2 := \mathcal{A}_2;
11 \hspace{1em} v_e := v_e + |A_2|/S_2;
12 \hspace{1em} q_o := q[i] + \eta_2; \hspace{1em} \widehat{q}_o := \widehat{q}[i] + \widehat{\eta}_2;
13 \textbf{if } (q_o \geq T_o) \textbf{ then}
14 \hspace{1em} \eta_3 := \texttt{sample}[idx]; \hspace{1em} idx := idx + 1; \hspace{1em} \widehat{\eta}_3 := \mathcal{A}_3;
15 \hspace{1em} v_e := v_e + |A_3|/S_3;
16 \hspace{1em} q_o := q[i] + \eta_3; \hspace{1em} \widehat{q}_o := \widehat{q}[i] + \widehat{\eta}_3;
17 \hspace{1em} \textbf{assert } (q_o + \widehat{q}_o \geq T_o + \widehat{T}_0);
18 \hspace{1em} out := (q_o))::out;
19 \hspace{1em} count := count + 1;
20 \textbf{else}
21 \hspace{1em} \textbf{assert } (\neg(q_o + \widehat{q}_o \geq T_o + \widehat{T}_0));
22 \hspace{1em} out := false::out;
23 \hspace{1em} i := i + 1;
24 \hspace{1em} \textbf{assert } (v_e \leq \epsilon);

Figure 6-14: Numerical Sparse Vector Technique and its transformed code.

\textbf{Variants of SVT Using while-priv Loop} To showcase the power of while-private loop and user-provided utility function, we evaluate on two mechanisms that use these features. The first is Adaptive
Figure 6-15: GapSVT and its transformed code.

### SVT, introduced in Section 6.5.3. The second, called SVT-WhilePriv, is a modified version of SVT where the user simply uses a while-private loop and asks the synthesizer to adaptively adjust the privacy.
$i := 0$

$\textbf{while}-\text{priv} (i < \text{size})$

$\text{if } (q[i] \geq T) \text{ then}$

$out := \text{true}::out;$

$\text{else}$

$out := \text{false}::out;$

$i := i + 1;$

Figure 6-16: Modified part of SVT to form SVT-WhilePriv.

cost paid to the above and below threshold answers respectively (Figure 6-16)

In both cases, we use the user-provided utility function of Equation 6.1 (Section 6.5.3). The utility function requires the user provide an example input for evaluation. To capture the characteristics of a typical usage of SVT, where the amount of above-threshold query answers is small, we designed an input as follows: we use a sample set of 100 query answers where 75 are well below the threshold ($\leq T - 1000$), 10 are well above the threshold ($\geq T + 1000$) and 15 close to the threshold ($= T + 50$). In both cases, the input $N$ (the minimum number of above-threshold queries to answer) is set to 20 in order to avoid answering queries that are well-below the threshold.

Moreover, due to the nature of the utility function, which computes utility based on true positives and false positives, we need to run the sketch mechanism (with randomness) many iterations for a good estimation of the utility. In the evaluation, we set the number of iterations to 2500.

For SVT-WhilePriv, DPGen successfully synthesizes a privacy mechanism that is identical to standard SVT: the synthesized mechanism adds noise with scale $3/\epsilon$ (resp. $3N/\epsilon$) to the threshold (resp. each query answer). The synthesized while condition is $\textbf{while} (i < \text{size} \land v_\epsilon \leq \epsilon - 2\epsilon/(3N))$. The synthesized program increments $v_\epsilon$ by $\epsilon/3$ before the branch, increments it by $2\epsilon/(3N)$ in the true branch and leaves it unchanged in the false branch (as the alignment in that case is 0). Note that although the synthesized code is syntactically different from standard SVT, they have exactly the same semantics.

For AdaptiveSVT, DPGen synthesizes a version (last row in Table 6-1) that is different from the one proposed in [44]. However, we confirmed that the average utility score for the synthesized mechanism across 2500 iterations is 24.4, meaning that it almost answers all above-threshold queries in an accurate way, with no false positives.
function NoisyMax(size: num, q: list num*)
returns max: num
precondition ∀ i. -1 ≤ \( \hat{q}_i \) ≤ 1

```
1  i := 0; bq := 0; max := 0;
2  while (i < size)
3      if (q[i] > bq ∧ i = 0)
4          max := i;
5      bq := q[i];
6      i := i + 1;
```

function Transformed NoisyMax(size, q, \( \hat{q} \), sample, \( \theta \), \( \lambda \))

```
8  \( v_e \) := 0; idx := 0;
9  i := 0; bq := 0; max := 0;
10 \( \eta_2 \) := sample[idx]; idx := idx + 1; \( \hat{\eta}_2 := \mathcal{A}_2 \);
11 \( v_e := v_e + |\mathcal{A}_2|/S_2 \);
12 bq\( ^\uparrow \) := bq + \( \eta_2 \);
13 \( \hat{b}q \) := \( \hat{\eta}_2 \);
14 \( \hat{b}q \) := 0; \( \hat{b}q \) := 0; \( \overline{\text{max}} \) := 0; \( \overline{\text{max}} \) := 0;
15 while (i < size)
16      if (\( \mathcal{L}_1 \)) \( bq \) := \( \hat{b}q \); \( \overline{\text{max}} \) := \( \overline{\text{max}} \);
17      \( q_0 := q[i] + \eta_1 \);
18      \( \hat{q}_0 := \hat{\eta}_1 \);
19 \( \eta_3 := \text{sample[idx]} \); idx := idx + 1; \( \hat{\eta}_3 := \mathcal{A}_3 \);
20 \( v_e := (\mathcal{L}_2 ? v_e : 0) + |\mathcal{A}_3|/S_3 \);
21 \( bq \) := \( bq + \eta_3 \);
22 \( \hat{b}q \) := \( \hat{b}q + \hat{\eta}_3 \);
23      if (\( \mathcal{L}_2 \)) \( \hat{b}q \) := \( \hat{b}q \); \( \overline{\text{max}} \) := \( \overline{\text{max}} \);
24      \( q_\neg \) := \( q[i] + \eta \);
25      \( \hat{q}_\neg := \hat{\eta} \);
26      if (\( q_\neg > bq \) ∨ \( i = 0 \))
27          \assert (q[i] + \( \hat{q}[i] \) + \( \eta \) + \( \eta^\neg \) > bq + \( \hat{b}q \) ∨ \( i = 0 \));
28      \( \max := i \);
29      \( \overline{\text{max}} := 0 \);
30 \( \hat{b}q \) := \( bq + \hat{b}q - (q[i] + \eta) \);
31 \( bq := q[i] + \eta \);
32 \( \overline{\text{max}} := \overline{\text{max}} + \eta^\neg \);
33      else
34          \assert (-(q[i] + \( \hat{q}[i] \) + \( \eta \) + \( \eta^\neg \) > bq + \( \hat{b}q \) ∨ \( i = 0 \)));
35 // shadow execution
36      if (\( q[i] + \( \hat{q}[i] \) + \( \eta \) > bq + \( \hat{b}q \) ∨ \( i = 0 \))
37          \( \hat{b} \) := \( q[i] + \( \hat{q}[i] \) + \( \eta \) - bq \);
38 \( \overline{\text{max}} := i - \max \);
39      i := i + 1;
40 \assert (v_e \leq e) ;
```

Figure 6-17: Report Noisy Max and its transformed code. \( \mathcal{L}_1 \) stands for shadow execution selectors.
Figure 6-18: PartialSum and its transformed code.

In this case, DPGen has successfully synthesized a private solution that offers better utility (as measured by the user-provided utility function) on the sample data compared with the original mechanism in [44]. In practice, a user could provide more sample data to avoid over-fitting, with the cost of a longer synthesis time.

**Report Noisy Max** Another well-known privacy mechanism is Report Noisy Max: it finds the identity of the item with the maximum score in the database. We present this mechanism in a simplified manner: given a series of query answers as inputs, the mechanism returns the index of the query with maximum answer.
function SmartSum (M, T, size: num, q: list num)
returns (out: list num), bound(ε)
precondition ∀i. -1 ≤ (q[i]) ≤ 1 ∧ (∀i. (q[i]) ≠ 0 ⇒ (∀j. q[j] = 0))

1 i := 0; next := 0; sum := 0;
2 while (i < size ∧ i ≤ T)
3 if ((i + 1) mod M = 0) then
4 next := sum + q[i];
5 sum := 0;
6 out := next::out;
7 else
8 next := next + q[i];
9 sum := sum + q[i];
10 out := next::out;
11 i := i + 1;

function Transformed SmartSum (M, T, size, q, ̃q, sample, θ, λ)
returns (out)

14 ṽe := 0; idx := 0;
15 i := 0; next₀ := 0; sum₀ := 0;
16 ̃sum₀ := 0; next₀ := 0;
17 while (i < size ∧ i ≤ T)
18 if ((i + 1) mod M = 0) then
19 ̃η₁ := sample[idx]; idx := idx + 1;
20 ṽe := ṽe + |A₁| / S₁; ̃η₁ := A₁;
21 next₀ := sum₀ + q[i] + ̃η₁;
22 next₀ := sum₀ + ̃q[i] + ̃η₁;
23 assert (next₀ = 0);
24 out := next₀::out;
26 else
27 ̃η₂ := sample[idx]; idx := idx + 1;
28 ṽe := ṽe + |A₂| / S₂; ̃η₂ := A₂;
29 next₀ := next₀ + q[i] + ̃η₂;
30 next₀ := next₀ + ̃q[i] + ̃η₂;
31 ̃η₃ := sample[idx]; idx := idx + 1;
32 ṽe := ṽe + |A₃| / S₃; ̃η₃ := A₃;
33 sum₀ := sum₀ + q[i] + ̃η₃;
34 ̃sum₀ := sum₀ + ̃q[i] + ̃η₃;
35 assert (next₀ = 0);
36 out := next₀::out;
37 i := i + 1;
38 assert (ṽe ≤ ε);

Figure 6-19: SmartSum and its transformed code.
The synthesis of Report Noisy Max requires an extension of the alignment-based proof technique called Shadow Execution [15], which is also supported by DPGen. While the synthesis time for Report Noisy Max is slightly longer than other mechanisms without while-private loop, DPGen synthesizes a private mechanism that is the same as the standard Report Noisy Max.

**Partial Sum and Smart Sum** These two mechanisms release aggregate statistics. Partial Sum simply sums up all query answers and directly releases the final sum. A more advanced mechanism [56] is proposed by Chen et al. to release the prefix sum of a series query answers: \( \sum_{i=0}^{T} q[i] \). Notably, these two mechanisms rely on a slightly different adjacency definitions: at most one of the query answers can differ by at most 1:

\[
\forall i. -1 \leq (\tilde{q}[i]) \leq 1 \wedge (\forall i. (\tilde{q}[i]) \neq 0 \Rightarrow (\forall j. \tilde{q}[j] = 0))
\]

Despite such difference, DPGen is able to synthesizes both Partial Sum and Smart Sum.

**Comparison with KOLAHAL** As the implementation of KOLAHAL [33] is not publicly available, we were unable to make a direct comparison with KOLAHAL on all benchmarks. The last column of Table 6-1 shows data that we collect from [33] when the corresponding mechanism is also evaluated on KOLAHAL (N/A is listed if the mechanism is not part of the experiments in [33]). We note that one difference between DPGen and KOLAHAL is that the latter synthesizes a set of candidate solutions instead of one; sometimes, the ideal solution might not be a top candidate: for example, the ideal solution for Smart Sum is ranked as the 4th one. Moreover, KOLAHAL requires manually-provided mechanism sketches where noise locations are annotated, while DPGen automatically generates sketches as discussed in Section 6.4.

**6.6.2 Performance**

We note that the synthesis time with the default utility function is significantly smaller than the one with a user-provided utility function (used for SVT-WhilePriv and AdaptiveSVT). The reason is that the default utility function does not need to execute the sketch mechanism at all. Moreover, among the ones using the default utility function, Report Noisy Max takes longer to synthesis, since it needs to use the

**Comparison with KOLAHAL** We note that for the same mechanisms, the synthesis time of DPGen is considerably smaller than that of KOLAHAL. While this is not an apple-to-apple comparison, we contribute the efficiency to the reduced search space of our sketch generation algorithm and the qualities of the counterexamples generated in the search loop.

### 6.7 Summary

In this chapter, we present DPGen, an automated differential privacy mechanism synthesizer that is able to synthesize sophisticated DP mechanisms such as adaptive mechanisms. DPGen employs a novel approach to automatically generate sketch mechanisms with potential random variables, and uses an enhanced CEGIS loop to fill the holes in the sketch according to customizable utility functions. Compared with recent synthesis work, DPGen is reasonably faster in synthesizing non-adaptive mechanisms, and is the only tool that is powerful enough to synthesize sophisticated adaptive ones. Evaluations show DPGen synthesizes a variety of non-adaptive mechanisms within minutes and adaptive ones within an hour. Compared with state-of-the-art synthesizer KOLAHAL, DPGen (1) automatically generates the locations of random variables, (2) is more efficient in synthesizing non-adaptive mechanisms due to reduced search space of the templates, and (3) is able to synthesize sophisticated mechanisms such as AdaptiveSVT.
Chapter 7

Future Works

In this chapter, we sketch interesting future directions that can further improve the frameworks proposed in this dissertation.

7.1 Analyzing Other Variants of Differential Privacy

In this dissertation, we mainly focus on “pure” differential privacy (Definition 2.1), a classic definition that underpins many systems for data privacy. Meanwhile, a number of variants of differential privacy have been proposed with different targets. One popular variant is approximate differential privacy:

**Definition 7.1** (Approximate Differential Privacy [34]). Let $\epsilon \geq 0$ and $\delta \in [0, 1]$. A probabilistic computation $M : D \rightarrow O$ is $(\epsilon, \delta)$-differentially private if $\forall D \sim D'$ (where $D, D' \in \mathcal{D}$) and every possible (measurable) output set $E \subseteq O$, we have

$$
\mathbb{P}[M(D) \in E] \leq e^\epsilon \mathbb{P}[M(D') \in E] + \delta
$$

The most notable difference is $\delta$, an extra parameter that allows relaxations of pure differential privacy. Intuitively, this parameter can be viewed as a “failure” probability such that the mechanism satisfies pure differential privacy with a probability of $1 - \delta$. This opens up many choices in designing mechanisms for DP. For example, it allows the use of the Gaussian noise, which is often a more favorable choice in data analyses than Laplace noise.

The proposed analysis frameworks in this dissertation are all built on randomness alignment, a simple yet powerful proving technique that enables analysis on complex pure DP mechanisms. However, it relies on the nice point-wise property of pure differential privacy to track privacy cost for verification. The privacy cost reasoning for approximate differential privacy is more subtle: a mechanism satisfies
approximate differential privacy with a curve of \((\epsilon, \delta)\) pairs instead of a single point. In order to properly track and reason about the privacy loss for approximate differential privacy, a typical way in manual proofs is to track via \textit{privacy loss random variable}:

**Definition 7.2** (Privacy Loss Random Variable [34]). Let \(D \sim D'\) be two adjacent databases, and \(M\) be a privacy mechanism. The privacy loss for \(M\) under output \(o\) is defined as:

\[
\ell_{M,D,D'}(o) = \ln \left( \frac{P[M(D) = o]}{P[M(D') = o]} \right)
\]

The privacy loss random variable \(\mathcal{L}_{M,D,D'} = \ell_{M,D,D'}(O)\) is the transformation of the output random variable \(O = M(D)\) by the function \(\ell_{M,D,D'}\).

However, it is not trivial to track the privacy loss random variable in program verification due to the complexities of privacy loss random variables. One future direction is to extend randomness alignment to precisely track the privacy loss under this setting.

### 7.2 Improving Usability of Analysis Frameworks

The usability of the program analysis frameworks proposed in this dissertation (Chapter 4, 5, 6) can be further enhanced in a few aspects. One direction is to extend them to support more noise distributions. We focus on Laplace distribution due to its adoption in a variety of mechanisms as shown in our benchmark. However, in practice, other noise distributions such as Exponential and Uniform noise distributions have been used in mechanisms to satisfy pure differential privacy. In general, new random distributions can be added to alignment-based proofs in a modular way via extra typing rules, as showcased in [14]. Extend our existing frameworks for more noise distributions in future iterations is another future direction.

Moreover, our analysis frameworks are designed for DP mechanisms, rather than larger programs built on top of them. An interesting area is integrating our existing frameworks with tools like DFuzz [36], which are more efficient on programs built on top of DP mechanisms using composition theorems (but don’t verify the mechanisms themselves).
Chapter 8

Conclusion

In this dissertation, we presented novel programming frameworks for automated analysis of differential privacy mechanisms.

For formal verification of differential privacy, we presented a novel proving technique “Shadow Execution” and embed it in a programming framework named ShadowDP. By keeping track of a separate cost-free execution, ShadowDP is able to express more flexible randomness alignments. This is a key to enabling efficient verification of sophisticated differential privacy mechanisms such as Report Noisy Max. Compared with prior works, we showed that ShadowDP is both efficient in verification and powerful in expressing novel mechanisms with very little annotation burden on mechanism designers.

To enable full automations in the verification and provide useful information when proofs fail, we presented an automated prover and disprover CheckDP. The framework extends the program analysis in ShadowDP and employs a novel bidirectional counterexample-guided proof synthesis loop to simultaneously generate proofs or counterexamples. Furthermore, we provided comprehensive evaluations on CheckDP: it is able to automatically and efficiently generate proofs for a battery of correct mechanisms, while providing precise counterexamples for challenging incorrect ones.

Finally, to further aid programmers in designing private and useful differential privacy mechanisms, we presented an automated synthesizer called DPGen. DPGen employs a novel approach to automatically generate sketch mechanisms with potential random variables, and uses an enhanced CEGIS loop to fill the holes in the sketch according to customizable utility functions. Evaluations suggest DPGen improves the performance of synthesizing non-adaptive mechanisms by orders of magnitude compared with existing work. Meanwhile, it is the only framework that synthesizes adaptive mechanisms.

In summary, we developed a set of programming frameworks to aid programmers design better differential privacy mechanisms. This includes novel approaches capable of proving and disproving challenging mechanisms, as well as synthesizing new mechanisms based on noise-free sketch programs.
The frameworks were evaluated on a variety of mechanisms, including published ones and novel variants. All of them are reasonably fast and useful results are generated within minutes in most cases.
Appendix

Complete Soundness Proof for ShadowDP

We first prove a couple of useful lemmas. Following the same notations, we use \( m \) for original memory and \( m' \) for extended memory with aligned and shadow variables but not the distinguished privacy tracking variable \( v_e \). Moreover, we assume that memory tracks the entire list of sampled random values at each point.

**Lemma A.1** (Numerical Expression). \( \forall e, m', \Gamma, \text{ such that } \Gamma \vdash e : \text{num}_{(\langle v^\circ, v^\dagger \rangle,)}, \text{ we have} \)

\[
[e]_{m'} + [n^\circ]_{m'} = [e]_{\Gamma^\circ m'},
\]

\[
[e]_{m'} + [n^\dagger]_{m'} = [e]_{\Gamma^\dagger m'}.
\]

**Proof.** Induction on the inference rules.

- (T-\textsc{Num}): trivial.
- (T-\textsc{Var}): If \( \Gamma^\circ (x) \) is \( \ast \), \( v^\circ \) is \( \bar{x}^\circ \). By Definition 4.2, \( \Gamma^\circ m'(x) = m'(x) + m' (\bar{x}^\circ) \). If \( \Gamma^\circ (x) = v^\circ \), \( v^\circ \) is \( v^\circ \). Again, we have \( \Gamma^\circ m'(x) = m'(x) + [v^\circ]_{m'} \) by Definition 4.2. The case for \( \Gamma^\dagger \) is similar.

- (T-\textsc{Plus}, T-\textsc{Times}, T-\textsc{Ternary}, T-\textsc{Index}): by the induction hypothesis (list is treated as a collection of variables of the same type).

\( \square \)

**Lemma A.2** (Boolean Expression). \( \forall e, m', \Gamma, \text{ such that } \Gamma \vdash e : \text{bool}, \text{ we have} \)

\[
[e]_{m'} = [e]_{\Gamma^\circ m'} = [e]_{\Gamma^\dagger m'}.
\]

**Proof.** Induction on the inference rules.
• (T-Boolean): trivial.

• (T-Var): the special case where \( \delta^o = \delta^i = 0 \). Result is true by Definition 4.2.

• (T-ODot): by Lemma A.1 and induction hypothesis, we have \([e_i]_{m'} + [\pi^o_i]_{m'} = [e_i]_{\Gamma^m} \) for \( i \in \{1, 2\} \). By the assumption of (T-ODot), we have

\[
[e_1 \circ e_2]_{m'} \Leftrightarrow [(e_1 + \pi^o_1) \circ (e_2 + \pi^o_2)]_{m'} = [e_1 \circ e_2]_{\Gamma^m}
\]

The case for \( \Gamma^\dagger \) is similar.

• (T-Ternary, T-Index): by the induction hypothesis.

\[\square\]

## A.1 Injectivity

We first prove that the type system maintains injectivity.

**Lemma A.3.** \( \forall c, c', pc, m', m'_1, m'_2, \Gamma_1, \Gamma_2. pc \vdash \Gamma_1 \{c \rightarrow c'\} \Gamma_2 \land [c']_{m'} m'_1 \neq 0 \land [c']_{m'} m'_2 \neq 0, \) then we have

\[(m'_1 = m'_2) \lor (\exists \eta. \Gamma^\star \Gamma_1 \eta \neq \Gamma^\star \Gamma_2 \eta)\), \( \star \in \{\circ, \dagger\} \)

**Proof.** By structural induction on \( c \).

• Case **skip**: trivial since it is non-probabilistic.

• Case **x**: trivial since it is non-probabilistic.

• Case \( c_1; c_2 \): Let \( pc \vdash \Gamma_1 \{c_1 \rightarrow c'_1\} \Gamma, pc \vdash \Gamma \{c_2 \rightarrow c'_2\} \Gamma_2 \). There exists some \( m'_3 \) and \( m'_4 \) such that

\[
[c'_1]_{m'} (m'_3) \neq 0 \land [c'_2]_{m'_3} (m'_4) \neq 0
\]
\[ [c'_1]_{m'}(m'_4) \neq 0 \land [c'_2]_{m'}(m'_2) \neq 0 \]

If \( m'_3 = m'_4 \), results is true by the induction hypothesis on \( c'_2 \). Otherwise, we have \( \exists \eta . \Gamma^*_2 m'_3(\eta) \neq \Gamma^*_2 m'_4(\eta) \). Hence, \( \Gamma^*_1 m'_4(\eta) \neq \Gamma^*_2 m'_4(\eta) \); contradiction.

- **Case if e then c1 else c2:** let \( pc' \vdash \Gamma \{ c_i \rightarrow c'_i \} \Gamma'_i \). By typing rule, we have \( \Gamma_i' \sqcup \Gamma_i' = \Gamma_2 \). Given initial memory \( m' \), both executions take the same branch. Without loss of generality assume \( c_1 \) is executed. Let \( [c'_i]_{m'} m'_3 \neq 0, [c'_i]_{m'} m'_4 \neq 0, [c'_i]_{m'} m'_3 \neq 0 \) and \( [c'_i]_{m'} m'_4 \neq 0 \). There are two cases:

  1. \( m'_3 = m'_4 \): in this case, \( c'_i \) only changes the value of \( x^* \) to \( n \) when \( \Gamma_2(x) = * \) and \( \Gamma_1'(x) = n \).
     Moreover, we have \( m'_1(x^*) = \Gamma[[n]] m'_4 = \Gamma'_1 m'_2(x^*) \) for those variables. Hence, \( m'_1 = m'_2 \).

  2. \( m'_3 \neq m'_4 \): by induction hypothesis, \( \exists \eta . m'_3(\eta) \neq m'_4(\eta) \). Hence, \( m'_1 \neq m'_2 \), and \( m'_1(\eta) \neq m'_2(\eta) \) for the same \( \eta \).

When \( pc = T \), \( c' \) also includes \( c^\dagger \). In this case, result still holds after \( c^\dagger \) since \( c^\dagger \) is deterministic by construction.

- **Case while e do c:** let \( pc' \vdash \Gamma \sqcup \Gamma_f \{ c \rightarrow c' \} \Gamma_f \). We proceed by induction on the number of iterations:

  1. \( c \) is not executed: \( m'_1 = m' = m'_2 \).

  2. \( c \) is executed \( n + 1 \) times: similar to the \( c_1;c_2 \) case.

When \( pc = T \), \( c' \) also includes \( c^\dagger \). In this case, result still holds after \( c^\dagger \) since \( c^\dagger \) is deterministic by construction.

- **Case \( (\eta := \text{Lap} \ r; S, v_\eta) \):** if \( m'_1 \neq m'_2 \), we know that \( m'_1(\eta) \neq m'_2(\eta) \) since \( c' \) only modifies the value of \( \eta \). In this case, it must be true that \( \Gamma^*_1 m'_1(\eta) \neq \Gamma^*_2 m'_2(\eta) \); otherwise, \( m'_1 = m'_2 \) by the injectivity check in rule \( (T\text{-}\text{LAPlace}) \).

\[ \square \]

**Proof of Lemma 4.2**

*Proof.* Direct implication of Lemma A.3.  \[ \square \]
A.2 Instrumentation

Next, we show a property offered by the auxiliary function $\Gamma_1, \Gamma_2, pc \Rightarrow c'$ used in the typing rules. Intuitively, they allow us to freely switch from one typing environment to another by promoting some variables to star type.

**Lemma A.4 (Instrumentation).** $\forall pc, \Gamma_1, \Gamma_2, c'$, if $(\Gamma_1, \Gamma_2, pc) \Rightarrow c'$, then for any memory $m'_1$, there is a unique $m'_2$ such that $\llbracket c' \rrbracket_{m'_1}(m'_2) \neq 0$ and

$$
\begin{cases}
\Gamma_1^0 m'_1 = \Gamma_2^0 m'_2 \land \Gamma_1^1 m'_1 = \Gamma_2^1 m'_2 & \text{if } pc = \bot \\
\Gamma_1^0 m'_1 = \Gamma_2^0 m'_2 & \text{if } pc = \top
\end{cases}
$$

**Proof.** By construction, $c'$ is deterministic. Hence, there is a unique $m'_2$ such that $\llbracket c' \rrbracket_{m'_1}(m'_2) \neq 0$.

Consider any variable $x \in \text{Vars}$. By the construction of $c'$, we note that $m'_1(x) = m'_2(x)$ and $\tilde{x}^0$ differs in $m'_1$ and $m'_2$ only if $\Gamma_1^0(x) = \nu$ and $\Gamma_2^0(x) = \ast$. In this case, $\Gamma_1^0 m'_1(x) = m'_1(x) + [\nu] m'_1 = m'_2(x) + m'_2(\tilde{x}^0) = \Gamma_2^0 m'_2(x)$. Otherwise, $\Gamma_1^0(x) = \Gamma_2^0(x)$ (since $\Gamma_1 \subseteq \Gamma_2$). When $\Gamma_1^0(x) = \Gamma_2^0(x) = \ast$, $\Gamma_1^0 m'_1(x) = m'_1(x) + m'_1(\tilde{x}^0) = m'_2(x) + m'_2(\tilde{x}^0) = \Gamma_2^0 m'_2(x)$. When $\Gamma_1^0(x) = \Gamma_2^0(x) = \nu$ for some $\nu$, $\Gamma_1^0 m'_1(x) = m'_1(x) + [\nu] m'_1 = m'_2(x) + [\nu] m'_2 = \Gamma_2^0 m'_2(x)$.

When $pc = \bot$, the same argument applies to the case of $\Gamma_1^1 m'_1 = \Gamma_2^1 m'_2$. \hfill $\square$

A.3 Shadow Execution

Next, we show the main properties related to shadow execution.

**Lemma A.5.** $\forall e, \Gamma, m'$, if $e$ is well-typed under $\Gamma$, we have

$$
\llbracket (e, \Gamma)^{\dagger} \rrbracket_{m'} = \llbracket e \rrbracket_{\Gamma^1 m'}
$$

**Proof.** By structural induction on the $e$.

- $e$ is $r$, $true$ or $false$: trivial.
Therefore, we write

By structural induction on

Proof.

for

not apply to sampling instructions. Hence, Lemma A.6

program executions.

Lemma A.6

\[
\Gamma(x) = \text{num}_{(d, n)} \Rightarrow \langle x, \Gamma \rangle^\dagger_{m'} = m'(x) + m'(\hat{\Gamma}) = \langle x \rangle_{\Gamma^\dagger_{m'}}.
\]

When \(\Gamma(x) = \text{num}_{(d, n)}\), we have \(\langle x, \Gamma \rangle^\dagger_{m'} = \langle x + \hat{\Gamma} \rangle_{m'} = \langle \hat{\Gamma} \rangle_{\Gamma^\dagger_{m'}}\) (by Lemma A.1). Otherwise, \(\langle x, \Gamma \rangle^\dagger_{m'} = m'(x) = \Gamma^\dagger_{m'}(x)\).

• \(e\) is \(e_1\) op \(e_2\): by induction hypothesis.

• \(e\) is \(e_1[e_2]\): By the typing rule (T-INDEX), \(\Gamma \vdash e_2 : \text{num}(0, 0)\). Hence, \([e_2]_{\Gamma^\dagger_{m'}} = [e_2]_{m'}\) by Lemma A.1.

Let \(\Gamma \vdash e_1 : \text{list}(d, d')\). When \(e_1 = q\) and \(d = \ast\), \([q[e_2], \Gamma \rangle^\dagger_{m'} = [q[e_2] + \hat{\Gamma}[e_2]]_{\Gamma^\dagger_{m'}}\). When \(d = \ast\) and \(e_1\) is not a variable, \([e_1[e_2], \Gamma \rangle^\dagger_{m'}\) is defined as \([e_1[\Gamma] + [e_2]]_{\Gamma^\dagger_{m'}}\). By induction hypothesis, this is the same as \([e_1[\Gamma]]_{\Gamma^\dagger_{m'}}\).

When \(d = n\), \([e_1[e_2], \Gamma \rangle^\dagger_{m'}\) is defined as \([e_1[e_2] + n\hat{\Gamma}]_{\Gamma^\dagger_{m'}}\), which is the same as \([e_1[e_2]]_{\Gamma^\dagger_{m'}}\) (by Lemma A.1).

Otherwise, result is true by Lemma A.2.

• \(e\) is \(e_1 :: e_2\) and \(\neg e\): by induction hypothesis.

• \(e\) is \(e_1?e_2 : e_3\): by induction hypothesis, we have \([e_1, \Gamma \rangle^\dagger_{m'} = [e_1]_{\Gamma^\dagger_{m'}}\). Hence, the same element is selected on both ends. Result is true by induction hypothesis.

□

Next, we show that shadow execution simulates two executions on \(\Gamma^\dagger\)-related memories via two program executions.

**Lemma A.6** (Shadow Execution). \(\forall c, c^\dagger, \Gamma. (\forall x \in \text{Asgnd}(c). \Gamma^\dagger(x) = \ast) \land \langle c, \Gamma \rangle^\dagger = c^\dagger\), we have

\[
\forall m_1', m_2'. [c^\dagger]_{m_1'}(m_2') = [c]_{\Gamma^\dagger_{m_1'}}(\Gamma^\dagger_{m_2'}).
\]

**Proof.** By structural induction on \(c\). First, we note that the construction of shadow execution does not apply to sampling instructions. Hence, \(c\) and \(c^\dagger\) fall into the deterministic portion of programs. Therefore, we write \(m_2 = [c]_{m_1}\) when \(m_2\) is the unique memory such that \([c]_{m_1}(m_2) = 1\), and likewise for \(c'\). Then this lemma can be stated as

\[
m_2' = [c^\dagger]_{m_1'} \implies \Gamma^\dagger_{m_2'} = [c]_{\Gamma^\dagger_{m_1'}}.
\]
• **c** is **skip**: we have \( m'_1 = m'_2 \) in this case. Hence, \( \Gamma^\dagger m'_2 = \Gamma^\dagger m'_1 = [\textbf{skip}]_{\Gamma^\dagger m'_1} \).

• **c** is \((c_1; c_2)\): let \( \langle c_1, \Gamma \rangle^\dagger = c_1^\dagger \) and \( \langle c_2, \Gamma \rangle^\dagger = c_2^\dagger \) and \( m' = [c_1^\dagger]_{m'_1} \). By induction hypothesis, we have \( \Gamma^\dagger m' = [c_1]_{\Gamma^\dagger m'_1} \). By language semantics, \( m'_2 = [c_2^\dagger]_{m'_1} \). By induction hypothesis, with \( \Gamma^\dagger \) as both the initial and final environments, we have \( \Gamma^\dagger m'_2 = [c_2]_{\Gamma^\dagger m'_2} = [c_1; c_2]_{\Gamma^\dagger m'_1} \).

• **c** is \( x := e \): we have \( c^\dagger = (\hat{x} := \langle e, \Gamma \rangle^\dagger - x) \) in this case. Moreover, \( m'_2 = m'_1 \{ \langle \langle e, \Gamma \rangle^\dagger \rangle - x \} \). By Lemma A.5, we have \( \Gamma^\dagger m'_2 = m'_1 \{ \langle \langle e, \Gamma \rangle^\dagger \rangle - x \} \). For variable \( x \), we know that \( \Gamma(x)^\dagger = * \) by assumption. So \( \Gamma^\dagger m'_2(x) = m'_2(x) + m'_2(\hat{x}) = m'_1(x) + \{ \langle e, \Gamma \rangle^\dagger - x \} \). By induction hypothesis, we have \( \Gamma^\dagger m'_2 = \Gamma^\dagger m'_1 \). For a variable \( y \) other than \( x \), the result is true since both \( c \) and \( c^\dagger \) do not modify \( y \) and \( y^\dagger \), and \( y \)'s distances does not change due to the well-formedness check in typing rule (T-ASGN).

• **c** is **if** \( e \) **then** \( c_1 \) **else** \( c_2 \): let \( \langle c_1, \Gamma \rangle^\dagger = c_1^\dagger \) and \( \langle c_2, \Gamma \rangle^\dagger = c_2^\dagger \). In this case,

\[
c^\dagger = \textbf{if} \langle e, \Gamma \rangle^\dagger \textbf{ then } c_1^\dagger \textbf{ else } c_2^\dagger.
\]

By Lemma A.5, we have \( \Gamma^\dagger m'_1 = [\langle e, \Gamma \rangle^\dagger]_{m'_1} \). Hence, both \( c \) and \( c^\dagger \) will take the same branch under \( \Gamma^\dagger m'_1 \) and \( m'_1 \) respectively. The desired results follow from induction hypothesis on \( c_1 \) or \( c_2 \).

• **c** is **while** \( e \) **do** \( c \): again, since \( \Gamma^\dagger m'_1 = [\langle e, \Gamma \rangle^\dagger]_{m'_1} \). The desired result follows from induction on the number of loop iterations.

\( \square \)

Next, we show that when \( pc = T \) (i.e., when the shadow execution might diverge), the transformed code does not modify shadow variables and their distances.

**Lemma A.7 (High PC).** \( \forall c, c', \Gamma_1, \Gamma_2. T \vdash \Gamma_1 \{ c \rightarrow c' \} \Gamma_2 \) then we have

1. \( (\forall x \in \text{Asgn}(c). \Gamma_2^\dagger(x) = * \land (\Gamma_1^\dagger(x) = * \implies \Gamma_2^\dagger(x) = *) \)

2. \( \forall m'_1, m'_2, [c']_{m'_1}(m'_2) \neq 0 \implies \Gamma_1^\dagger m'_1 = \Gamma_2^\dagger m'_2 \)

**Proof.** By structural induction on \( c \).

• **c** is **skip**: trivial.
• $c$ is $x := e$: When $pc = \top$, $\Gamma^\downarrow_2(x) = \ast$ by the typing rule. ($\Gamma^\downarrow_1(x) = \ast \Rightarrow \Gamma^\downarrow_2(x) = \ast$) since $\Gamma^\downarrow_2(y) = \Gamma^\downarrow_1(y)$ for $y \neq x$.

In this case, $c'$ is defined as $(\overline{x}^\downarrow := x + \nu^\downarrow - e; x := e)$ where $\Gamma^\downarrow_1 + x : \nu^\downarrow$. By the semantics, we have $m'_2 = m'_1([x + \nu^\downarrow - e]_m/\overline{x})$. Hence, $\Gamma^\downarrow_2m'_2(x) = m'_2(x) + [x + \nu^\downarrow - e]_m = [e]_m + [x + \nu^\downarrow]_m - [e]_m = \Gamma^\downarrow_1m_1(x)$.

For any $y \neq x$, both its value and its shadow distance do not change (due to the wellformedness check in rule (T-Asgn)). Hence, $\Gamma^\downarrow_1m'_1(y) = \Gamma^\downarrow_2m'_2(y)$.

• $c$ is $c_1;c_2$: by induction hypothesis.

• $c$ is $\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$: when $pc = \top$, we have

$c' = \textbf{if } e \textbf{ then } (\textbf{assert } (e^0); c'_1;c'_2') \textbf{ else } (\textbf{assert } (\neg e^0); c'_2;c''_2)$.

By the induction hypothesis, we know that $c'_1$ and $c''_2$ does not modify any shadow variable and their ending typing environments, say $\Gamma'_1, \Gamma'_2$, satisfy condition 1. Hence, the ending environment $\Gamma'_1 \sqcup \Gamma'_2$ satisfies condition 1 too.

To show (2), we assume $c_1$ is executed without losing generality. Let $m'_3$ be the memory state between $c'_1$ and $c''_2$. By induction hypothesis, $\Gamma^\downarrow_1m'_1 = (\Gamma'_1)^\downarrow m'_3$. By the definition of $\Gamma_i, \Gamma_1 \sqcup \Gamma_2, \top \Rightarrow c''_2$, and $c'_1$ only modifies $\overline{x}^\circ, x \in Vars$. Hence, $\Gamma^\downarrow_2m'_2 = (\Gamma'_2)^\downarrow m'_3 = \Gamma^\downarrow_1m'_1$.

• $c$ is $\textbf{while } e \textbf{ do } c'$: by the definition of $\sqcup$ and induction hypothesis, condition 1 holds.

Moreover, by the definition of $\Gamma, \Gamma \sqcup \Gamma_f, \top \Rightarrow c, c_3$ only modifies $\overline{x}^\circ, x \in Vars$. Hence, $\Gamma^\downarrow_2m'_2 = \Gamma^\downarrow_1m'_1$ by induction on the number of iterations.

• $c$ is sampling instruction: does not apply since $pc = \bot$ in the typing rule.

$\square$

**Lemma A.8.** $\forall c, c', c^\uparrow, \Gamma_1, \Gamma_2, (\forall x \in \text{Asgnd}(c). \Gamma^\downarrow_2(x) = \ast) \land ([c, \Gamma_2]^\uparrow = c^\uparrow \land c'$ is deterministic, and $(\forall m'_1, m'_2. [c']_m_1(m'_2) \neq 0 \Rightarrow \Gamma^\downarrow_1m'_1 = \Gamma^\downarrow_1m'_2)$, then we have

$$\forall m'_1, m'_2. [c'; c^\uparrow]_m_1(m'_2) = [c]_m_1(\Gamma^\downarrow_2m'_2).$$
Proof. \( c \) and \( c' \) are deterministic. Hence, this lemma can be stated as

\[
\forall m_1', m_2'. m_2' = \llbracket c'; c^+ \rrbracket m_1' \implies \Gamma^+_2 m_2' = \llbracket c \rrbracket \Gamma^+_1 m_1'.
\]

Let \( m_3' = \llbracket c' \rrbracket m_1' \) and \( m_2' = \llbracket c^+ \rrbracket m_1' \). By assumption, we have \( \Gamma^+_1 m_3' = \Gamma^+_1 m_1' \). Moreover, by Lemma A.6, we have \( \Gamma^+_2 m_2' = \llbracket c \rrbracket \Gamma^+_1 m_1' \). Hence, \( \Gamma^+_2 m_2' = \llbracket c \rrbracket \Gamma^+_1 m_1' \).

\[\square\]

A.4 Soundness

Finally, we prove the Pointwise Soundness Lemma.

Proof of Lemma 4.3

Proof. By structural induction on \( c \). Note that the desired inequalities are trivially true if \( \llbracket c' \rrbracket m_1' (m_2') = 0 \). Hence, in the proof we assume that \( \llbracket c' \rrbracket m_1' (m_2') > 0 \).

• Case (skip): trivial.

• Case \( (x := e) \): by (T-Asgn), we have \( \max(c'' \Gamma^+_1 m_1') = 0 \). This command is deterministic in the sense that when \( \llbracket c' \rrbracket m_1' (m_2') \neq 0 \), we have \( \llbracket c' \rrbracket m_1' (m_2') = 1 \) and \( m_2' = m_1' \llbracket e \rrbracket m_1' / x \). To prove (4.1) and (4.2a), it suffices to show that \( \Gamma^+_2 m_2' = \Gamma^+_1 m_1' \llbracket e \rrbracket \Gamma^+_1 m_1' / x \) and \( \Gamma^+_2 m_2' = \Gamma^+_1 m_1' \llbracket e \rrbracket \Gamma^+_1 m_1' / x \). We prove the latter one as the other (given \( pc = \bot \)) can be shown by a similar argument.

First, we show \( \Gamma^+_2 m_2'(x) = \Gamma^+_1 m_1' \llbracket e \rrbracket \Gamma^+_1 m_1' / x \)(x). Let \( \Gamma_1 \vdash e : (\nu^\circ, \nu^\dagger) \). By the typing rule, we have \( \Gamma^+_2(x) = \nu^\circ \), and

\[
\Gamma^+_2 m_2'(x) = m_2'(x) + \llbracket \nu^\circ \rrbracket m_2'
\]

\[= \llbracket e \rrbracket m_1' + \llbracket \nu^\circ \rrbracket m_2'
\]

\[= \llbracket e \rrbracket m_1' + \llbracket \nu^\circ \rrbracket m_1'
\]

\[= \llbracket e \rrbracket \Gamma^+_1 m_1'
\]

\[= \Gamma^+_1 m_1' \llbracket e \rrbracket \Gamma^+_1 m_1' / x \)(x). \]
The third equality is due to well-formedness, and the forth equality is due to Lemma A.1.

Second, we show that \( \Gamma^y_2m'_2(y) = \Gamma^y_1m'_1(\{e\}^r_1m'_1/x) \) for \( y \neq x \). First, by Rule (T-Asgn), \( \Gamma^2(y) = \Gamma^y_1(y) \), \( m'_1(y) = m'_2(y) \) and \( m'_1(\overline{y}) = m'_2(\overline{y}) \). If \( \Gamma^2_2(y) = \Gamma^y_1(y) = \star, \Gamma^y_2m'_2(y) = m'_2(y) + m'_2(\overline{y}) = m'_1(y) + m'_1(\overline{y}) = \Gamma^y_0m'_1(y) \). If \( \Gamma^2_2(y) = \Gamma^y_1(y) = \ast, \Gamma^y_2m'_2(y) = m'_2(y) + \lceil \nu \rceil_m = m'_1(y) + \lceil \nu \rceil_{m'_1} = \Gamma^y_0m'_1(y) \), where \( \lceil \nu \rceil_{m'_1} = \lceil \nu \rceil_{m'_1} \) due to well-formedness.

- Case \( (c_1; c_2) \): For any \( m'_2 \) such that \( \llbracket c'_1; c'_2 \rrbracket_{m'_1}(m'_2) \neq 0 \), there exists some \( m' \) such that

\[
\llbracket c'_1 \rrbracket_{m'_1}(m') \neq 0 \land \llbracket c'_2 \rrbracket_{m'}(m'_2) \neq 0.
\]

Let \( pc + \Gamma_1\{c_1 \rightarrow c'_1\} \Gamma, pc + \Gamma\{c_2 \rightarrow c'_2\} \Gamma_2 \) For (4.1), we have

\[
\llbracket c'_1; c'_2 \rrbracket_{m'_1}(m'_2) = \sum_{m'} \llbracket c'_1 \rrbracket_{m'_1}(m') \cdot \llbracket c'_2 \rrbracket_{m'}(m'_2)
\]

\[
\leq \sum_{m'} \llbracket c'_1 \rrbracket_{\Gamma^y_1m'_1}(\Gamma^{y'}m') \cdot \llbracket c'_2 \rrbracket_{\Gamma^{y'}m'}(\Gamma^y_2m'_2)
\]

\[
\leq \sum_{m'} \llbracket c'_1 \rrbracket_{\Gamma^y_1m'_1}(m') \cdot \llbracket c'_2 \rrbracket_{m'}(\Gamma^y_2m'_2)
\]

\[
= \llbracket c'_1; c'_2 \rrbracket_{\Gamma^y_1m'_1}(\Gamma^y_2m'_2).
\]

Here the second line is by induction hypothesis. The change of variable in the third line is due to Lemma 4.2.

For (4.2a) and (4.2b), let \( \epsilon_1 = \max(c''_{1'} \lceil \Gamma^y_1m'_1 \rceil), \epsilon_2 = \max(c''_{2'} \lceil \Gamma^y_2m'_2 \rceil) \) and \( \epsilon = \max(c''_{1'}; c''_{2'} \lceil \Gamma^y_1m'_1 \rceil) \). Note that \( \epsilon_1 + \epsilon_2 \leq \epsilon \) due to the fact that \( m'_2 = (\epsilon_1 + \epsilon_2) \in \llbracket c''_{1'}; c''_{2'} \rrbracket_{m'_1}(0) \). By induction hypothesis we have one of the following two cases.

1. \( \llbracket c'' \rrbracket_{m'}(m'_2) \leq \exp(\epsilon_2) \llbracket c'' \rrbracket_{\Gamma^{y'}m'}(\Gamma^y_2m'_2) \). We have

\[
\llbracket c'_1; c'_2 \rrbracket_{m'_1}(m'_2) = \sum_{m'} \llbracket c'_1 \rrbracket_{m'_1}(m') \cdot \llbracket c'_2 \rrbracket_{m'}(m'_2)
\]

\[
\leq \exp(\epsilon_2) \sum_{m'} \llbracket c'_1 \rrbracket_{\Gamma^y_1m'_1}(\Gamma^{y'}m') \cdot \llbracket c'_2 \rrbracket_{\Gamma^{y'}m'}(\Gamma^y_2m'_2)
\]

\[
\leq \exp(\epsilon_2) \sum_{m'} \llbracket c'_1 \rrbracket_{\Gamma^y_1m'_1}(m') \cdot \llbracket c'_2 \rrbracket_{m'}(\Gamma^y_2m'_2)
\]
\[ \leq \exp(\epsilon_2)[c_1; c_2]_{\Gamma_1^m} (\Gamma_2^m) \]
\[ \leq \exp(\epsilon)[c_1; c_2]_{\Gamma_1^m} (\Gamma_2^m). \]

The first inequality is by induction hypothesis. The change of variable in the third line is again due to Lemma 4.2. The last line is because \( \epsilon_1 + \epsilon_2 \leq \epsilon \).

2. \( [c_1']_{m'}(m') \leq \exp(\epsilon_2)[c_2]_{\Gamma_1^m} (\Gamma_2^m) \). By induction hypothesis on \( c_1 \), we have two more cases:

(a) \( [c_1']_{m'}(m') \leq \exp(\epsilon_1)[c_1]_{m'}(m') \). In this case,
\[
[c_1'; c_2']_{m'}(m') = \sum_{m'} [c_1']_{m'}(m') \cdot [c_2']_{m'}(m') \\
\leq \exp(\epsilon) \sum_{m'} [c_1]_{\Gamma_1} (\Gamma_2^m) \cdot [c_2]_{\Gamma_1} (\Gamma_2^m) \\
\leq \exp(\epsilon) \sum_{m'} [c_1]_{\Gamma_1} (m') \cdot [c_2]_{m'} (\Gamma_2^m) \\
\leq \exp(\epsilon) [c_1; c_2]_{\Gamma_1} (\Gamma_2^m).
\]

The second line is by induction hypothesis and the fact that \( \epsilon_1 + \epsilon_2 \leq \epsilon \). The change of variable in the third line is due to Lemma 4.2.

(b) \( [c_1']_{m'}(m') \leq \exp(\epsilon_1)[c_1]_{\Gamma_1} (\Gamma_2^m) \). We have
\[
[c_1'; c_2']_{m'}(m') \leq \exp(\epsilon) [c_1; c_2]_{\Gamma_1} (\Gamma_2^m)
\]
by a similar argument as above.

- Case (if \( e \) then \( c_1 \) else \( c_2 \)): If \( \Gamma_1 \vdash e : \text{bool} \), then by Lemma A.2 we have \( [e]_{m'} = [e]_{\Gamma_1^m} = [e]_{\Gamma_1^m} \). Hence, the same branch is taken in all related executions. By rule (T-Ir), the transformed program is
\[
\text{if } e \text{ then } \textbf{assert} \ (\langle e \rangle ^{\circ} ; c_1' ; c_2') \text{ else } \textbf{assert} \ (-\langle e \rangle ^{\circ} ; c_2' ; c_2')
\]
and \( \Gamma_1, \Gamma_1 \sqcup \Gamma_2, \perp \Rightarrow c_1' \), \( \Gamma_2, \Gamma_1 \sqcup \Gamma_2, \perp \Rightarrow c_2' \). Without loss of generality, suppose that \( c_1 \) is executed in all related executions. By Lemma A.4, there is a unique \( m' \) such that \( [c_1']_{m'}(m') \neq 0 \).
By induction hypothesis, we have

$$[c'_1]_{m'_1}(m') \leq [c_1]_{\Gamma_1 m'_1}(\Gamma_1^\dagger m')$$

Moreover, by Lemma A.4, $\Gamma_1^\dagger m' = \Gamma_2^\dagger m_2'$ and

$$[c'_1; c''_1]_{m'_1}(m'_2) = [c'_1]_{m'_1}(m').$$

Hence, we have

$$[c'_1; c''_1]_{m'_1} m'_2 \leq [c_1]_{\Gamma_1 m'_1}(\Gamma_1^\dagger m'_2) = [c_1]_{\Gamma_1 m'_1}(\Gamma_1^\dagger m'_2)$$

(4.2a) or (4.2b) can be proved in a similar way.

When $\Gamma_1 \not\vDash e : \text{bool}$, the aligned execution will still take the same branch due to the inserted assertions. Hence, the argument above still holds for (4.2a) or (4.2b).

For the shadow execution, we need to prove (4.1). By rule (T-If), $pc' = \top$, and the transformed program is

$$\text{if } e \text{ then } (\text{assert } ([e]^\circ); c'_1; c''_1) \text{ else } (\text{assert } (\neg[e]^\circ); c'_2; c''_2);$$

$$\langle \text{if } e \text{ then } c_1 \text{ else } c_2, \Gamma_1 \uplus \Gamma_2 \rangle^\dagger$$

By Lemma A.7 and the definition of $\Rightarrow$ under high pc, we have that $\forall m'_1, m_2'. [c'_1]_{m'_1}(m'_2) \neq 0 \Rightarrow \Gamma_2^\dagger m'_1 = \Gamma_2^\dagger m'_2$, and $\Gamma_1 \uplus \Gamma_2(x) = * \forall x \in \text{Asgnd}(c_1; c_2)$. Furthermore, the program is deterministic since it type-checks under $\top$. Therefore, $\max(W'' | m'_2) = 0$ and (4.1) holds by Lemma A.8.

- Case (while e do c): Let $W = \text{while } e \text{ do } c$. If $\Gamma_1 \vdash e : \text{bool}$, then $[e]_{m'_1} = [e]_{\Gamma_2 m'_1} = [e]_{\Gamma_2 m'_1}$. By rule (T-While) we have $pc' = \bot$, $\Gamma_2 = \Gamma \uplus \Gamma_f$ and the transformed program is

$$W' = c_\bot; \text{while } e \text{ do } (\text{assert } ([e]^\circ); c'; c'')$$

where $\bot \vdash \Gamma_1 \uplus \Gamma_f \{c \rightarrow c'\} \Gamma_f$, $(\Gamma_1, \Gamma_1 \uplus \Gamma_f, \bot) \Rightarrow c_\bot$ and $(\Gamma_f, \Gamma_1 \uplus \Gamma_f, \bot) \Rightarrow c''$. We proceed by natural induction on the number of loop iterations (denoted by $i$).
When \( i = 0 \), we have \([e]_{m'_1} = \text{false}\). By the semantics, we have \([W']_{m'_1} = \text{unit}([c_s]_{m'_2})\), \([W]_{\Gamma_1 m'_1} = \text{unit}(\Gamma_1 m'_2)\), and \([W]_{\Gamma_1 m'_2} = \text{unit}(\Gamma_1^2 m'_1)\). Furthermore, \(\max(W'')_{m'_1} = 0\). By Lemma A.4, \((\Gamma_1 m'_1) = (\Gamma_2^2([c_s]_{m'_2}))\) and \((\Gamma_1^2 m'_1) = (\Gamma_2^3([c_s]_{m'_2}))\). So desired result holds.

When \( i = j + 1 > 0 \), we have \([e]_{m'_1} = \text{true}\). By the semantics, we have \([W']_{m'_1} = [c_s; c'; c''; c'; c'']_{m'_1}\) and \([W]_{\Gamma_1 m'_1} = [c_s; c'; c''; c'; c'']_{\Gamma_1 m'_1}, \star \in \{\circ, \dagger\}\). For any \( m'' \) such that \([c_s; c'; c''; c'; c'']_{m'_1}(m'') = k_1 \neq 0 \) and \([c'; c'']_{m''}(m'_2) = k_2 \neq 0 \), we know that \([c; c']_{\Gamma_1 m'_1}(\Gamma_1^2 m') = k_1 \). Moreover, there is a unique \( m'_2 \) such that \([c''; c'']_{m''}(m'_2) = 1 \) by Lemma A.4. By structural induction on \( c \), we have \( k_1 \leq [c]_{\Gamma_1 m''}(\Gamma_1^2 m'_3) \) and either \( k_1 \leq \exp(\max(c'_{m'_1}(m'))[c]_{\Gamma_1 m''}(\Gamma_1^2 m'_3)) \) or \( k_1 \leq \exp(\max(c''_{m''}(m'_2))) \). By Lemma A.4, \( \Gamma_1 m'' = \Gamma_2^2 m'_2 \). Hence, we can replace \( \Gamma_f \) with \( \Gamma_2 \) in the inequalities. Finally, the desired result holds following the same argument as in the composition case.

Otherwise, by rule \((T-\text{While})\) we have \( pc' = \tau, \Gamma_2 = \Gamma \sqcup \Gamma_f \) and the transformed program is \( W' = c_s; W'' \) where

\[
W'' = \text{while } e \text{ do } (\text{assert}([e]_\circ; c'); \text{while } e \text{ do } c, \Gamma_2)\]

and \( T \vdash \Gamma_1 \sqcup \Gamma_f \{c \rightarrow c'\} \Gamma_f, (\Gamma_1, \Gamma_1 \sqcup \Gamma_f, \tau) \Rightarrow c_s \) and \( (\Gamma_f, \Gamma \sqcup \Gamma_f, \tau) \Rightarrow c'' \). Since we do not have sampling instructions in this case, the program is deterministic, thus \( \max(W''_{m''}) = 0 \). Note that by rule \((T-\text{Asgn})\), \( \forall x \in \text{Asgn}(c), \Gamma_f^x(x) = \ast \). Hence, \( \Gamma_f^x(x) = \ast \) for \( x \in \text{Asgn}(c) \). Moreover, let \( m'' \) be the unique memory such that \([c_s]_{m'_1}(m'') \neq 0 \), we have \( \Gamma_1^m m'_1 = \Gamma_2^3 m' \) by Lemma A.4. Therefore, by Lemmas A.7 and A.8, we have

\[
[W']_{m'_1}(m'_2) = [W'']_{m''}(m'_2) = [W]_{\Gamma_1 m''}(\Gamma_1^3 m'_2) = [W]_{\Gamma_1 m'_1}(\Gamma_2^3 m'_2)
\]

- Case \( (\eta := \text{Lap } r; S, v_\eta) \): let \( \mu_r \) be the probability density (resp. mass) function of the Laplace (resp. discrete Laplace) distribution with zero mean and scale \( r \). Since \( \mu_r(v) \propto \exp(-|v|/r) \) we have

\[
\forall v, d \in \mathbb{R}, \mu_r(v) \leq \exp(|d|/r)\mu_r(v + d). \tag{A.1}
\]
When \([c']_{m_1'}(m_2') \neq 0\), we have \(m_2' = m_1'\{v/\eta\}\) for some constant \(v\), and \([c']_{m_1'}(m_2') = \mu_r(v)\).

We first consider the case \(S = \emptyset\). By \((T\text{-}\text{LAPLACE})\), we have \(\Gamma_2(x) = \Gamma_1(x)\) for \(x \neq \eta\), and \(\Gamma_2(\eta) = (v_\eta, 0)\). Therefore, for \(x \neq \eta\), we have

\[
\Gamma_2^+ m_2'(x) = m_2'(x) + [\Gamma_2^+(x)]_{m_2'} = m_1'(x) + [\Gamma_1^+(x)]_{m_2'} = m_1'(x) + [\Gamma_1^+(x)]_{m_1'} = \Gamma_1^+ m_1'(x).
\]

In the second equation, we have \(m_2'(x) = m_1'(x)\) because \(m_2 = m_1\{v/\eta\}\) and \(x \neq \eta\). The third equation is due to the well-formedness assumption. Also, \(\Gamma_2^+ m_2'(\eta) = m_1'(\eta) = v\) since \(\Gamma_2^+(\eta) = 0\). Therefore, we have \(\Gamma_2^+ m_2' = \Gamma_1^+ m_1'(v/\eta)\) and thus \([c]_{\Gamma_1^+ m_1'}(\Gamma_2^+ m_2') = \mu_r(v) = [c']_{m_1'}(m_2')\).

Similarly, we can show that \(\Gamma_2^+ m_2' = \Gamma_1^+ m_1'(v + d/\eta)\) and therefore \([c]_{\Gamma_1^+ m_1'}(\Gamma_2^+ m_2') = \mu_r(v + d)\).

Furthermore, since \(v_e := v_e + |v_\eta|/r\), the set \((c'' \uparrow_{m_1'}^{m_2'})\) contains a single element \([|v_\eta|/r]_{m_2} = |d|/r\), and thus \(\max(c'' \uparrow_{m_1'}^{m_2'}) = |d|/r\). Therefore, we have

\[
[c']_{m_1'}(m_2') = \mu_r(v) \\
\leq \exp(|d|/r)\mu_r(v + d) \\
= \exp(\max(c'' \uparrow_{m_1'}^{m_2'}))\mu_r(v + d).
\]

The second inequality is due to \((A.1)\). Thus \((4.1)\) and \((4.2a)\) hold.

Next consider the case \(S = \dagger\). By the typing rule, we have \(\Gamma_2^\dagger(x) = \Gamma_1^\dagger(x) = \Gamma_1^\dagger(x)\) for \(x \neq \eta\), and \(\Gamma_2(\eta) = (v_\eta, 0)\). By a similar argument we have \(\Gamma_2^\dagger m_2' = \Gamma_1^\dagger m_1'(v/\eta)\) and \(\Gamma_2^\dagger m_2' = \Gamma_1^\dagger m_1'(v + d/\eta)\).

Furthermore, since \(v_e := 0 + |v_\eta|/r\), we have \(\max(c'' \uparrow_{m_1'}^{m_2'}) = |d|/r\). Therefore, we have

\[
[c']_{m_1'}(m_2') = \mu_r(v) = [c]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\dagger m_2') \\
\leq \exp(|d|/r)\mu_r(v + d) \\
= \exp(\max(c'' \uparrow_{m_1'}^{m_2'}))\mu_r(v + d).
\]
Thus (4.1) and (4.2b) hold.

Finally, consider $S = e \ ? \ S_1 : S_2$. For any $m'_1$, we have $S = S_1$ if $[e]_{m'_1} = \text{true}$ and $S = S_2$ if $[e]_{m'_1} = \text{false}$. By induction, $S$ evaluates to either $\circ$ or $\uparrow$ under $m'_1$. Thus the proof follows from the the two base cases above.
Bibliography


URL https://doi.org/10.1145/3341698

URL https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00060

URL http://dx.doi.org/10.1007/11787066_1

URL http://doi.acm.org/10.1145/1863543.1863568

URL http://doi.acm.org/10.1145/2429069.2429113

URL https://doi.org/10.1145/3372297.3417282


URL https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00081

URL https://doi.org/10.1145/3373718.3394796


Vita

Yuxin Wang

EDUCATION

Pennsylvania State University
Ph.D. candidate in Computer Science and Engineering
State College, PA
May 2022

Huazhong University of Science and Technology
B.E. in Computer Science and Technology
Wuhan, China
June 2017

EXPERIENCE

Uber Technologies Inc.
Ph.D. Software Engineer Intern
San Francisco, CA
May 2021 - August 2021

Pennsylvania State University
Graduate Research Assistant
State College, PA
Fall 2017 - Spring 2022

PUBLICATIONS


