Author/s:
Yan, Pengbo

Title:
Formally Verifying the Security of Probabilistic Oblivious Algorithms

Date:
2025

Persistent Link:
https://hdl.handle.net/11343/362834

Formally Verifying the Security of Probabilistic Oblivious Algorithms

by

Pengbo Yan

ORCID: 0000-0003-0396-8343

A thesis submitted in total fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering and Information Technology
School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

November 2025

THE UNIVERSITY OF MELBOURNE

Abstract

Faculty of Engineering and Information Technology
School of Computing and Information Systems

Doctor of Philosophy

by Pengbo Yan
ORCID: 0000-0003-0396-8343

Oblivious algorithms are designed to protect sensitive information by preventing side-channel attacks that exploit observable behavior such as memory access patterns. While deterministic oblivious algorithms offer strong security guarantees, they often incur significant performance overhead. Probabilistic oblivious algorithms mitigate this cost by introducing randomness, but verifying their security becomes substantially more complex. In this thesis, we develop a formal framework for systematically verifying the security of probabilistic oblivious algorithms.

Our contributions are threefold. First, we design a novel program logic that combines classical reasoning with probabilistic and independence-based reasoning. Built atop Probabilistic Separation Logic (PSL), our logic supports uniform distributions, probabilistic independence, and secret-dependent control flow, and we prove it sound in Isabelle/HOL. Second, we introduce a transformation-based verification framework for oblivious algorithms that use encryption, ensuring the correct use of encryption under both statistical and computational security definitions. Our framework allows the insertion of ghost code to record observable behavior while verifying that no unencrypted secrets are leaked. Third, we apply our approach to verify four real-world case studies—Oblivious Sampling, the Melbourne Shuffle, Path ORAM, and the Path Oblivious Heap—each posing unique verification challenges such as dynamic random choices, delayed leakage, and complex invariants.

To our knowledge, this is the first framework that enables formal verification of practical probabilistic oblivious algorithms involving encryption, providing a sound and expressive foundation for secure system design.

Declaration of Authorship

I, Pengbo Yan, declare that this thesis titled, 'Formally Verifying the Security of Probabilistic Oblivious Algorithms' and the work presented in it are my own. I confirm that:

- The thesis comprises only my original work towards the Doctor of Philosophy - Engineering and IT except where indicated in the preface;
- due acknowledgement has been made in the text to all other material used; and
- the thesis is fewer than the maximum word limit in length, exclusive of tables, maps, bibliographies and appendices as approved by the Research Higher Degrees Committee.

Signed: Pengbo Yan

Date: 02/09/2025

Preface

This thesis presents original research conducted during my candidature for the Doctor of Philosophy at the University of Melbourne. Unless otherwise indicated, the results and formal developments are my own.

Parts of Chapters 3 and 5 appeared in Formal Methods 2024 [Yan et al., 2025], where I was the primary contributor. Chapter 4 is unpublished material. The Isabelle/HOL formalisation accompanying this thesis has been released as an open-source artifact [Yan, 2025].

No part of this thesis has been submitted for any other qualification, nor was any material completed prior to my enrolment. No third-party editorial assistance was used.

This research was supported by the Melbourne Research Scholarship and the University of Melbourne, Faculty of Engineering and Information Technology.

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Toby Murray, Olya Ohrimenko, Rob Sison and Thuan Pham, for their invaluable guidance, encouragement, and patience throughout my PhD journey. Their expertise and thoughtful feedback have shaped both this thesis and my broader research perspective.

I am also grateful to the Formal Methods group and the Isabelle/HOL AFP community for exchanging ideas and sharing open-source artifacts, and to the Melbourne Research Scholarship and the University of Melbourne FEIT for their financial support.

Besides academics, I would like to thank my family—Zhen Yan, Zhong Wu, Pengcheng Yan, Hongzhen Wu, and Ying Liu—whose unwavering belief in me and support during difficult times sustained me through the ups and downs of the PhD.

I am thankful to my colleagues and friends:
– Dongge Liu and Wenhua Ling, for facing the difficult COVID-19 period together.
– Zhenzhi Lai and Tian Qiu, for our trusted discussions over varieties of topics.
– Wentao Gao, Faxing Wang, Wenqi Yan, for board games played with true enthusiasm.
– Lianglu Pan, for offering helpful guidance on university procedures and even on games.
– Zhiyuan Zhang, Chenlu Zhang, for traveling/meeting at opposite ends of the world.
– Louis Cheung and Aaron Bembenek, for warm and friendly conversations.

Many thanks to my friends in Australia:
– Qianyun Lin, for countless enjoyable moments that cannot be easily summarised.
– Jiahui Zhang, Haode Huang, Yu Xue, Yechen Tang, Jiaying Zhuang for our D&D sessions, which weekly gave me something to look forward to during my thesis writing.
– Xinning Huang, Saite Guo, Peter Zhu, Xiaowei Shang, Jiawei Yang and many others, for the games and other activities we enjoyed together.

Finally, I would like to thank my friends overseas:
– Yongqing Liao, Jianqin Lin, Sen Zheng, Yifan Shao, Zihao Huang, Junxian Wen; Yongan Chen, Jiayuan Zou, Haizhou You, Bo Chen, Jincheng Li; and my other middle school classmates. Our decade-long friendship has been my anchor in the river of time.
– Jiarong Yang, Zhenbang Ye, Bojing Zhou, Biaoyan Fang; Yingming Ma, JiaYu Xia and other university classmates. Our annual gatherings around the Spring Festival have been a much-needed source of relaxation throughout my PhD years.
– Zecheng Liao, Yutiao Chen, Xuan Zhang, Shihao Li, Dawei Si, Sida Chen, Qi Zhang, Yifan Wu, Kailin Ding, and other online friends, for the daily chats and jokes that were especially vital during the lockdown.

Contents

List of Figures

Chapter 1

Introduction

Side-channel attacks allow attackers to infer sensitive information by eavesdropping on a program's execution, when the sensitive data are not directly observable (e.g. because they are encrypted). For example, sensitive documents or secret images can be reconstructed by only observing a program's memory access pattern [Islam et al., 2012, Lee et al., 2017, Liu et al., 2015a].

Many algorithms are charged with the protection of secrets in application contexts where such attacks are realistic, for example, cloud computing [Sasy and Ohrimenko, 2019, Zheng et al., 2017], secure processors [Fletcher et al., 2014, Maas et al., 2013] and multiparty computation [Liu et al., 2015b].

The goal of an oblivious algorithm (e.g. path ORAM [Stefanov et al., 2018], Melbourne shuffle [Ohrimenko et al., 2014]) is to hide its secrets from an attacker that can observe memory accesses. Probabilistic oblivious algorithms aim to do so while achieving better performance than *deterministic* oblivious algorithms. The various programming disciplines to defend against such attacks for deterministic algorithms [Molnar et al., 2006, Almeida et al., 2016] often lead to poor performance: e.g. to hide the fact that an array is accessed at a certain position, one may have to iterate over the entire array [Cauligi et al., 2019]. Probabilistic oblivious algorithms avoid this inefficiency by performing random choices at runtime to hide their secrets from attackers more efficiently. Unfortunately, probabilistic methods for achieving obliviousness are error prone and some have been shown insecure, as a result requiring non-trivial fixes [Kushilevitz et al., 2012, Goodrich and Mitzenmacher, 2011].

```
   demo(S, x) :
1    r ←$ U_{1···100};
2    S[(x + r) % 2] ← enc(x);
3    S[1 − (x + r) % 2] ← enc(2 ∗ x);
```

FIGURE 1.1: Demonstration Algorithm

Compared to deterministic programs, the correctness/incorrectness of probabilistic programs are harder to prove or discover. Generally we want to guarantee a probabilistic program's output has a certain distribution. However, it is hard to define an oracle distinguishing the correct/incorrect output distributions especially when their difference (i.e. statistical distance) is small. Existing automatic testing tools (for example, ObliCheck [Son et al., 2021]) can check obliviousness only for deterministic algorithms.

This thesis concentrates on formally verifying the security of probabilistic oblivious algorithms in systematic ways. The contributions include a new program logic allowing the verification of complicated and practical probabilistic oblivious algorithms, an automatic transformation to verify that the use of encryption in those algorithms is correctly employed, and relevant case studies that apply these techniques to verify several practical oblivious algorithms.

1.1 Probabilistic Independence

We present a synthetic and simple example to illustrate the structure of our approach, aiming to facilitate understanding.

As shown in Fig. 1.1, our demonstration algorithm takes two inputs: an array S of length 2 and a secret integer x. Accesses to S are observable to attackers. The algorithm first samples a random integer r uniformly from the range 1 to 100. It then writes the encryption of the secret, $\mathsf{enc}(x)$, into S at index $(x + r)$ mod 2. Since r is chosen uniformly, the value $(x + r)$ mod 2 equals 0 or 1 with equal probability (50%). Finally, the algorithm writes the encryption of $2x$, i.e. $\mathsf{enc}(2x)$, into the other position of S.

We can informally argue that this program is oblivious—that is, its observable memory access pattern (reads and writes to the array S) does not leak information of the secret input x. This follows from the fact that $(x + r)$ mod 2 is uniformly distributed over $0, 1$ due to the uniform randomness of r. As a result, each index of S is equally likely to be

```
demo_ghost(S, x) :
    Trace ← [];
1   r ←$ U_{1···100};
2   S[(x + r) % 2] ← enc(x);
    Trace ← Trace + ("Write", (x + r) % 2);
3   S[1 − (x + r) % 2] ← enc(2 ∗ x);
    Trace ← Trace + ("Write", 1 − (x + r) % 2);
    {Trace has a fixed uniform distribution with different input}
```

FIGURE 1.2: Adding ghost codes to the algorithm in Fig. 1.1

written first, regardless of the value of x. Moreover, since the values written to S are encrypted, the attacker cannot infer any information about x from the contents of S.

A more systematic approach is to introduce *ghost code* that records the observable behavior of the program. We call the recorded observable behaviour the program's *trace* and then formally prove that the recorded *trace* is independent of the secret. This idea is illustrated in Fig. 1.2. Specifically, the *trace* variable Trace captures the memory access pattern observable to an attacker. The Probabilistic Separation Logic (PSL) [Barthe et al., 2019] formalizes this methodology and provides tools to prove that Trace follows a fixed uniform distribution, regardless of the secret input. For simplicity, PSL assumes that the attacker cannot distinguish the encrypted data and thus does not record the actual written values.

Although PSL works for this simple example, many oblivious algorithms have complex semantics and invariants that are beyond the reach of PSL and some other previous works [Barthe et al., 2019, Son et al., 2021, Ye and Delaware, 2022, Darais et al., 2019] to reason about. For example, path ORAM [Stefanov et al., 2018] maintains an invariant stating that virtual addresses are independent of each other and of the program's memory access patterns; whereas the oblivious sampling algorithm [Sasy and Ohrimenko, 2019] contains secret- or random-variable-dependent random choices, conditional branches and loops, whose details we introduce in Section 3.1.

To overcome the challenges about complex semantics and invariants, we propose the reasoning strategy of combining classical and probabilistic (and independence) reasoning over different parts of the program. Classical reasoning is mature and powerful but does not work on probabilistic programs, while probabilistic and independence reasoning is restrictive about the complex semantics (for example, hard to reason about loops with random number of iterations). Most parts of the oblivious algorithm are not

probabilistic and could be described by the classical reasoning, while some small but important parts of them are probabilistic and need probabilistic independence reasoning. By combining these two reasoning styles, we can make the reasoning more applicable. We developed a new program logic, constructed by situating these ideas in the context of the Probabilistic Separation Logic (PSL) [Barthe et al., 2019], whose details will be introduced in Chapter 3.

1.2 Encryption in Oblivious Algorithms

As previously discussed, PSL and other prior works [Sasy and Ohrimenko, 2019, Stefanov et al., 2018, Shi, 2019] simplify the treatment of encryption by assuming that the attacker can observe only the memory access patterns, but not the actual data values.

While this assumption streamlines the verification of obliviousness, it implicitly presumes that encryption is always correctly and consistently applied. As a result, programs that inadvertently write unencrypted secrets to attacker-observable locations—such as a flawed version of the algorithm in Fig. 1.1—may still be verified as oblivious under this model. This is because the trace Trace used in the proof does not record plaintext values, and thus fails to capture such violations of confidentiality.

To address this limitation, we develop a verification framework that can detect writes of secret data while still allowing non-secret writes, without sacrificing simplicity. To this end, our approach augments the trace recording mechanism to distinguish between encrypted and unencrypted data. Specifically, whenever encrypted data is written, we record a special constant symbol $\perp$ in Trace to represent an indistinguishable ciphertext from the attacker's perspective. In contrast, unencrypted values are recorded verbatim. Consequently, if a secret value is written without proper encryption, its actual content will appear in Trace, violating the security condition and causing the verification to fail.

As an example, Fig. 1.3 illustrates the transformed version of the original program, which serves as the new verification target in our framework. This transformation is systematically defined and will be detailed in Chapter 4.

From a technical perspective, the main challenge arises from the fact that oblivious algorithms often rely on CPA- or CCA-secure encryption schemes (see Section 2.3.3), whose

```
demo_transformed(S, x) :
  Trace ← [];
1 r ←$ U_{1···100};
2 S[(x + r) % 2] ← enc(x);
  Trace ← Trace + ("Write", (x + r) % 2, ⊥);
3 S[1 − (x + r) % 2] ← enc(2 ∗ x);
  Trace ← Trace + ("Write", 1 − (x + r) % 2, ⊥);
  {Trace has a fixed uniform distribution with different input}
```

FIGURE 1.3: Transformation of the algorithm in Fig. 1.1

security guarantees are weaker and more intricate than perfect probabilistic independence. These guarantees are typically formalized through computational assumptions and indistinguishability games, rather than through information-theoretic notions. As a result, they complicate the overall security reasoning and significantly increase the complexity of proving the soundness of the verification framework.

In Chapter 4, we introduce our new security definitions, formally define the automatic transformation procedure, and present the corresponding soundness proofs. These results cover both statistical security guarantees (see Section 2.3.2 and Section 4.3) and computational security guarantees (see Section 2.3.3 and Section 4.4).

1.3 Practical Oblivious Algorithm Verification

Finally, practical oblivious algorithms are often designed with significant ingenuity and structural complexity. Understanding their correctness and verifying their security guarantees is typically non-trivial and time-consuming. To evaluate the applicability and expressiveness of our verification framework, we apply it to four representative algorithms:

- **Oblivious Sampling** [Sasy and Ohrimenko, 2019], which includes dynamic random choices and secret-dependent loops, is verified using our logic's capability to reason about probabilistic independence and classical information in a single logic.
- **The Melbourne Shuffle** [Ohrimenko et al., 2014] is a randomised oblivious permutation algorithm used as a building block in larger protocols. Although it has deterministic memory access patterns, it incorporates randomization in its data content. Our framework verifies both its access trace determinism and the correct application of encryption.

- **Path ORAM** [Stefanov et al., 2018] is a foundational algorithm in oblivious RAM constructions. It maintains invariants over a random position map and a tree structure, which are critical to ensuring the uniformity and independence of its access pattern. Our verification encodes and checks these invariants using the assertion system and inference rules introduced in Chapter 3.

- **Path Oblivious Heap** [Shi, 2019] extends Path ORAM with heap operations such as insert and delete. It presents a verification challenge due to its delayed information release, where secrets chosen in earlier operations only affect observable behavior later in the execution. Our framework captures this delayed leakage by maintaining invariants across sequential operations and verifying that all observable access patterns remain uniformly distributed.

In each case, we rewrite the original algorithm into a structured form that complies with our transformation requirements, then apply the program transformation (Chapter 4) to instrument it with ghost code.

All case studies involve encryption, which is essential for protecting data. Our transformation (Chapter 4) verifies encryption usage explicitly by inserting ghost code that distinguishes encrypted from plaintext data. This allows us to formally verify that no unencrypted secrets are leaked at observable locations.

We finally verify the transformed version using our program logic (Chapter 3) and establish its security guarantees—statistical or computational—depending on the assumptions about the encryption scheme.

These case studies are elaborated in Chapter 5, demonstrating the effectiveness of our approach in reasoning about realistic, probabilistically oblivious programs.

1.4 Summary

To summarize, the previous three sections—Section 1.1, Section 1.2, and Section 1.3—each corresponds to a research question that motivates one of the three main contributions of this thesis. These questions, and their corresponding chapters, are as follows:

- **What is a suitable and systematic logic for verifying probabilistic oblivious algorithms?** This question is addressed in Chapter 3 through the development of a formal logic that combines *classical* and *probabilistic* reasoning, specifically designed to capture probabilistic independence and obliviousness properties.

- **How can we reason about encryption, which complicates the analysis of probabilistic oblivious algorithms?** This challenge is tackled in Chapter 4, where we introduce a novel verification framework that systematically handles encryption and its security implications.

- **How can we verify practical and sophisticated oblivious algorithms, such as Path ORAM [Stefanov et al., 2018]?** This is the focus of Chapter 5, in which we apply our framework to real-world case studies and demonstrate its practical effectiveness.

Chapter 2

Preliminaries and Background

In this chapter, we will first introduce classical Hoare Logic which was proposed by Hoare [1969] in Section 2.1, which provides the foundational structure for all program logics discussed in this thesis and whose details will be relevant to Chapter 3. Next, we will present Probabilistic Separation Logic (PSL) as a preliminary for Chapter 3 in Section 2.2, followed by an introduction to encryption in oblivious algorithms as a prerequisite for Chapter 4 in Section 2.3. Finally, in Section 2.4, we briefly review additional related literature that, while not essential to our approach, provides useful context and broader perspectives.

2.1 Classical Hoare Logic

Hoare Logic, introduced by Hoare [1969], is a formal system for reasoning about the correctness of computer programs. It provides a structured method for specifying and verifying program behavior through Hoare triples, which establish the relationship between preconditions, program statements, and postconditions. This logic forms the foundation of many program verification frameworks and plays a crucial role in the formal methods explored in this thesis.

In general, Hoare Logic and other program logics consist of the following components:

- *Memory Model*: A formal definition of all possible program states at a given point in execution.

- *Programming Language*: A set of commands that form programs, along with their corresponding operational semantics, often described as a state transition relation. Each command defines how a program state evolves into a new state.
- *Assertion System*: A collection of logical assertions that describe program states. Given an assertion P and a state σ, the state σ either satisfies P or does not.
- *Inference Rules*: A set of formal rules governing the logical reasoning process. These rules define how assertions propagate through program statements, enabling the derivation of correctness proofs.

In the following subsections, we will introduce these components in sequence, providing a detailed discussion of each.

2.1.1 Memory Model and Programming Language

The memory model of classical Hoare Logic is based on an abstract state machine, where program execution is defined as a sequence of state transitions. Formally, let:

- **Var** be the set of *variable* names.
- **Val** be the set of *values* that variables can take.
- $\sigma : \mathbf{Var} \to \mathbf{Val}$ be a specific *state*, represented as a mapping from *variable* names to *values*.
- Σ be the *set* of all possible program *states* (σ).

A program is defined over a simple imperative language consisting of:

- e is an *expression* that consists of arithmetic and logical operations over variables. For example, $a + b$, where $a, b \in \mathbf{Var}$. Given a specific state σ where $a = 1$ and $b = 2$, the evaluation of $a + b$ yields $[\![a + b]\!]_\sigma = 1 + 2 = 3$, assuming that integers are a subset of **Val**.
- **C** is a *command*, encompassing statements such as assignments, sequencing, conditionals, and loops. The semantics of these constructs are defined in Fig. 2.1. We

denote the resulting state after executing a command C from an initial state s as $[\![C]\!](s)$.

$$[\![x := e]\!](s) = s[x \mapsto [\![e]\!]_s] \tag{2.1}$$

$$[\![C_1; C_2]\!](s) = [\![C_2]\!]([\![C_1]\!](s)) \tag{2.2}$$

$$[\![\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2]\!](s) = \begin{cases} [\![C_1]\!](s) & \text{if } [\![e]\!]_s = \mathsf{true} \\ [\![C_2]\!](s) & \text{otherwise} \end{cases} \tag{2.3}$$

$$[\![\textbf{while } e \textbf{ do } C]\!](s) = \begin{cases} [\![\textbf{while } e \textbf{ do } C]\!]([\![C]\!](s)) & \text{if } [\![e]\!]_s = \mathsf{true} \\ s & \text{otherwise} \end{cases} \tag{2.4}$$

FIGURE 2.1: Semantics of Programming Language

The command 2.1 (Assign) updates the variable x by evaluating the expression e in the current state. Formally, executing the command $x := e$ in state s produces a new state s' where x is assigned the value of e evaluated by the state s, while all other variables remain unchanged.

The command 2.2 (Sequential Composition) represents the sequential execution of two commands. First, C_1 is executed in state s, resulting in a new state s', which is then used to execute C_2.

The command 2.3 (If Statement) executes C_1 if e evaluates to true, otherwise it executes C_2. The command 2.4 (While Loop) repeatedly executes C as long as e holds true. If e evaluates to false, execution terminates and the state remains unchanged.

2.1.2 Assertion System and Inference Rules

The assertion system in Hoare Logic defines logical properties over program states. Formally, an assertion is a predicate P that determines whether a given state satisfies a specified condition. Given a state s, we write $s \models P$ to denote that s satisfies P, where the semantics is formally defined in Fig. 2.2. Assertions are expressed using first-order logic with equality, arithmetic, and logical operators to specify constraints on program variables. The meanings of these symbols are classical and intuitive (e.g. '$\leq$' means 'smaller or equal to' and '$\vee$' means 'or').

Assertion Type	**Semantics**
Equality	$s \models e_1 = e_2 \quad \text{iff} \quad [\![e_1]\!]_s = [\![e_2]\!]_s$
Inequality	$s \models e_1 \neq e_2 \quad \text{iff} \quad [\![e_1]\!]_s \neq [\![e_2]\!]_s$
Other Relations	$s \models e_1 < e_2 \quad \text{iff} \quad [\![e_1]\!]_s < [\![e_2]\!]_s$
	$\ldots$
Negation	$s \models \neg P \quad \text{iff} \quad s \not\models P$
Conjunction	$s \models P \wedge Q \quad \text{iff} \quad s \models P \text{ and } s \models Q$
Disjunction	$s \models P \vee Q \quad \text{iff} \quad s \models P \text{ or } s \models Q$
Implication	$s \models P \Rightarrow Q \quad \text{iff} \quad s \not\models P \text{ or } s \models Q$
Biconditional	$s \models P \Leftrightarrow Q \quad \text{iff} \quad (s \models P \Leftrightarrow s \models Q)$
Universal Quantification	$s \models \forall x.P(x) \quad \text{iff} \quad \forall v, s[x \mapsto v] \models P(v)$
Existential Quantification	$s \models \exists x.P(x) \quad \text{iff} \quad \exists v, s[x \mapsto v] \models P(v)$

FIGURE 2.2: Semantics of Assertions in Hoare Logic

$$\textbf{Assignment Rule}: \quad \{P[x := e]\}\ x := e\ \{P\} \tag{2.5}$$

$$\textbf{Sequential Composition Rule}: \quad \frac{\{P\}\ C_1\ \{R\} \quad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}} \tag{2.6}$$

$$\textbf{Conditional Rule}: \quad \frac{\{P \wedge B\}\ C_1\ \{Q\} \quad \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}} \tag{2.7}$$

$$\textbf{While Loop Rule}: \quad \frac{\{P \wedge B\}\ C\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{P \wedge \neg B\}} \tag{2.8}$$

$$\textbf{Consequence Rule}: \quad \frac{P' \Rightarrow P \quad \{P\}\ C\ \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\ C\ \{Q'\}} \tag{2.9}$$

FIGURE 2.3: Hoare Logic Rules

At the core of Hoare Logic is the *Hoare triple*, which has the general form:

$$\{P\}\ S\ \{Q\}$$

where P (the *precondition*) specifies assumptions about the program state before executing statement S, and Q (the *postcondition*) describes the expected state after executing S. A Hoare triple asserts that if the precondition P holds before executing S, then the postcondition Q will hold afterward, assuming that S terminates.

For example, the Hoare triple $\{x > 0\}\ x := x + 1\ \{x > 1\}$ states that if x is positive before execution, then after incrementing x by 1, its value must be greater than 1. This assertion is indeed valid assuming x is a mathematical integer (no overflow).

In Fig. 2.3, we introduce the rules of Hoare Logic for establishing valid triples systematically:

The Assignment Rule states that if P is a postcondition, then the Hoare triple remains valid if we substitute all occurrences of x in P with the expression e. This rule reflects the principle of weakest preconditions, ensuring that P holds after assigning e to x.

The Sequential Composition Rule allows reasoning about the correctness of two consecutive statements. This rule ensures that if C_1 transforms P into an intermediate assertion R, and C_2 transforms R into Q, then executing $C_1; C_2$ establishes Q.

For conditional statements, correctness depends on evaluating the boolean guard B. This states that if Q holds after executing C_1 when B is true and after executing C_2 when B is false, then Q holds after the conditional statement.

The While Loop Rule enables reasoning about loops using an invariant P. Here, P serves as a loop invariant that holds before and after each iteration of C, ensuring correctness after loop termination when B is false.

The Consequence Rule allows strengthening preconditions and weakening postconditions. This rule is essential for establishing correctness when the given precondition is weaker than necessary or when the postcondition is stronger than required.

2.1.3 Summary and Relation to Our Work

Hoare Logic consists of two main components: the *Memory Model* and *Programming Language*, which serve as assumptions abstracting program execution, and the *Assertion System* and *Inference Rules*, which provide useful reasoning mechanisms. This structure is also followed by various extensions of Hoare Logic, including Probabilistic Separation Logic (PSL) proposed by Barthe et al. [2019], which will be introduced in Section 2.2, our extension of PSL in Chapter 3, and other notable frameworks such as Separation Logic by Reynolds [2002] and Relational Hoare Logic by Benton [2004].

The development of a program logic generally requires the following steps:

- Defining the programming language semantics, including its memory model, to formally describe program execution.
- Constructing an assertion system tailored to the memory model, allowing for expressive reasoning about memory states.
- Formulating a set of systematic and useful inference rules for proving Hoare triples.
- Establishing the soundness of these inference rules with respect to the programming language semantics. In more complex settings, additional rules governing assertion implication may be required, whose soundness must also be proved against the memory model. However, traditional Hoare Logic does not include such assertion rules, as it re-uses standard first-order logic.

Some program logics may deviate from this structure. For example, Relaxed Separation Logic (RSL) proposed by Vafeiadis and Narayan [2013] has a very different memory model, which is axiomatic rather than operational. However, such cases are out of the scope of this thesis.

It is worth noting that Hoare Logic's assertion system and inference rules extend naturally to non-deterministic programs. For instance, in a setting where an assignment command may assign a value non-deterministically from a given set, Hoare Logic remains applicable [Apt, 1986]. This property will be particularly useful in Chapter 3.

2.2 Probabilistic Separation Logic

Probabilistic Separation Logic (PSL), introduced by Barthe et al. [2019], is a program logic designed for reasoning about probabilistic programs using probabilistic independence. In contrast to standard Hoare logic, where assertions are interpreted over individual states, PSL assertions are, intuitively, interpreted over *probability distributions* of states.

It reinterprets the separation conjunction ('$*$') in a novel way, extending the ideas of Separation Logic proposed by Reynolds [2002]. In PSL assertions, '$P * Q$' signifies that the distributions over which P and Q hold respectively are probabilistically independent. PSL has been shown to allow elegant reasoning about simple probabilistic programs.

For instance, the PSL triple $\vdash \{\mathbf{U}_a[S]\}\ b \leftarrow_\$ \mathbf{U}_S\ \{\mathbf{U}_a[S] * \mathbf{U}_b[S]\}$ states that if initially variable a follows a uniform distribution over the set S, and subsequently b is randomly assigned from S, then in the final state both a and b independently follow uniform distributions over S.

We might expect that probabilistic independence is highly valuable for verifying oblivious algorithms. However, PSL cannot be directly applied to sophisticated oblivious algorithms due to a range of limitations. To address this, we extend PSL in Chapter 3. Readers may choose to skip the detailed definitions in this section and refer back to them when reading Chapter 3.

In this section, we first introduce the preliminaries on probabilistic distributions in Section 2.2.1, followed by the memory model and programming language of PSL in Section 2.2.2. We then discuss the assertion system in Section 2.2.3 and inference rules in Section 2.2.4 and conclude with a summary in Section 2.2.5.

2.2.1 Preliminary about Probabilistic Distributions

A probability distribution over a countable set A is a function $\mu : A \to [0, 1]$ where $\Sigma_{a \in A}\ \mu(a) = 1$. We also write $\mu(B)$ for $\Sigma_{b \in B}\ \mu(b)$ where B can be any subset of A and $\mathbf{D}(A)$ for the set of all distributions over A. For example, a unit distribution over a single element a, $\mathsf{unit}(a)$, is defined by $(\lambda x.\ \mathsf{If}\ a = x\ \mathsf{then}\ 1\ \mathsf{else}\ 0)$. A uniform distribution over a finite set S, Unif_S, is $(\lambda x.\ \mathsf{If}\ x \in S\ \mathsf{then}\ 1/|S|\ \mathsf{else}\ 0)$, where $|S|$ represents the cardinality of S.

The *support* of a distribution μ, $\mathsf{supp}(\mu)$, is the set of all elements with nonzero probability, $\{a \in A \mid \mu(a) > 0\}$. In probabilistic programs, program states are represented as distributions over memories. The support of a state thus corresponds to the set of all possible memories.

Given a distribution μ over A and a function f from elements of A to a distribution over B, $f : A \to \mathbf{D}(B)$, we define $\mathsf{bind}(\mu, f) = \lambda b.\ \Sigma_{a \in A}\ \mu(a) \cdot f(a)(b)$. This operation is fundamental in defining the semantics of random selections and assignments to random variables in probabilistic programs.

Given two distributions μ_A and μ_B over the sets A and B, we define their product distribution as $\mu_A \otimes \mu_B = \lambda a, b.\ \mu_A(a) \cdot \mu_B(b)$. For a distribution μ over $A \times B$, we

define the left marginal as $\pi_1(\mu) = \lambda a.\ \Sigma_{b \in B}\ \mu(a, b)$ and the right marginal as $\pi_2(\mu) = \lambda b.\ \Sigma_{a \in A}\ \mu(a, b)$. We say that parts A and B are *independent* in distribution μ if and only if $\mu = \pi_1(\mu) \otimes \pi_2(\mu)$.

For a distribution μ over a set A and a subset $S \subseteq A$ with $\mu(S) > 0$, the conditional distribution given S is defined as $(\mu|S) = \lambda E.\ \frac{\mu(E \cap S)}{\mu(S)}$. Given two distributions μ_1, μ_2 over the same set, and a weighting factor $p \in [0, 1]$, we define their probabilistic mixture as $\mu_1 \oplus_p \mu_2 = (\lambda x.\ p \cdot \mu_1(x) + (1 - p) \cdot \mu_2(x))$. For edge cases, we define $\mu_1 \oplus_p \mu_2$ to be μ_1 when $p = 1$ and μ_2 when $p = 0$ even if the other distribution may be undefined. These definitions are fundamental for the semantics of conditional statements in probabilistic programs.

2.2.2 Memory Model and Programming Language

PSL distinguishes between *deterministic* and *random* program variables, expressions, and even commands. Deterministic variables and expressions only have deterministic values whereas random counterparts could be deterministic or random. As a result, it can define specialized inference rules for deterministic and random components separately, given that their reasoning principles differ.

Additionally, PSL prohibits loops with random conditions in its syntax to simplify the semantics, albeit at the cost of introducing certain restrictions. However, this distinction also increases the complexity of the programming language's syntax.

PSL defines **DV** as any countable set of deterministic variables and **RV** as any countable set of random variables, disjoint from **DV**.

Let **Val** be the countable set of values, $\mathbf{DetM} = \mathbf{DV} \to \mathbf{Val}$ be the set of deterministic memories, and $\mathbf{RanM} = \mathbf{RV} \to \mathbf{Val}$ be the set of random variable memories. A *semantic configuration* is a pair (σ, μ), where $\sigma \in \mathbf{DetM}$ represents a deterministic memory and $\mu \in \mathbf{D}(\mathbf{RanM})$ is a probability distribution over random variable memories. Configurations correspond to states in Hoare Logic.

As with program variables, PSL defines sets of deterministic and random expressions, denoted **DE** and **RE**, respectively. The former cannot mention random variables. Similar to Hoare Logic, these expressions include standard arithmetic and logical operations over variables.

Definition 2.1 (Expressions)**.** Expressions are either deterministic or random, defined as follows, where the dots (...) below stands for productions for other binary operators over deterministic and random expressions, respectively:

Deterministic expressions: $\mathbf{DE} \ni e_d ::= \mathbf{Val} \mid \mathbf{DV} \mid \mathbf{DE} + \mathbf{DE} \mid \mathbf{DE} \wedge \mathbf{DE} \mid \ldots$

Random expressions: $\mathbf{RE} \ni e_r ::= \mathbf{Val} \mid \mathbf{DE} \mid \mathbf{RV} \mid \mathbf{RE} + \mathbf{RE} \mid \mathbf{RE} \wedge \mathbf{RE} \mid \ldots$

Note that **DE** is a subset of **RE** so any deterministic expression is also a random expression. Given a deterministic memory σ and a random variable memory $m \in \mathsf{supp}(\mu)$, where (σ, μ) is a configuration, we write $[\![e_r]\!](\sigma, m)$ to denote the evaluation of the expression e_r. This evaluation follows a standard procedure—substituting variables with their values from σ and m, and then computing the result.

Moreover, $[\![e_r]\!](\sigma, \mu)$ denotes the probability distribution of the value of expression e_r with respect to memory σ and distribution μ. For instance, consider a state σ in which $a = 1$, and let μ be a uniform distribution over two memories: one where $x = 0 \wedge y = 0$, and another where $x = 1 \wedge y = 0$. Evaluating the expression $a + x + y$ under these conditions, denoted as $[\![a + x + y]\!](\sigma, \mu)$, yields a uniform distribution over the set $\{1, 2\}$.

The evaluation of deterministic expressions e_d returns a single value (i.e. a **Val**) depends only on the deterministic memory σ, so we often abbreviate it as $[\![e_d]\!](\sigma)$.

Furthermore, PSL defines two sets of program commands for our language: **C**, the complete set of commands, and **RC**, a subset of **C**, containing so-called "random" commands, which cannot assign to deterministic variables. This restriction ensures that deterministic variables only store deterministic values. The subscripts of the **if** statement and **while** indicate whether the condition is deterministic or random, using D for deterministic conditions and R for random conditions, respectively.

Finally, this definition imposes a restriction on commands that appear under random conditions (i.e., in the bodies of random if statements or loops), ensuring that they do not assign to deterministic variables.

Definition 2.2 (Programming Language of PSL)**.**

$$\mathbf{RC} \ni c ::= \mathbf{skip} \mid \mathbf{RV} \leftarrow \mathbf{RE} \mid \mathbf{RV} \leftarrow_{\$} \mathbf{U_{RE}} \mid \mathbf{RC}; \mathbf{RC} \mid \mathbf{if}_D\ \mathbf{DE}\ \mathbf{then\ RC\ else\ RC} \mid \mathbf{if}_R\ \mathbf{RE}\ \mathbf{then\ RC\ else\ RC} \mid \mathbf{while}_D\ \mathbf{DE}\ \mathbf{do\ RC} \mid \mathbf{while}_R\ \mathbf{RE}\ \mathbf{do\ RC}$$

$$\mathbf{C} \ni c ::= \mathbf{skip} \mid \mathbf{DV} \leftarrow \mathbf{DE} \mid \mathbf{RV} \leftarrow \mathbf{RE} \mid \mathbf{RV} \leftarrow_{\$} \mathbf{U_{RE}} \mid \mathbf{C}; \mathbf{C} \mid \mathbf{if}_D\ \mathbf{DE}\ \mathbf{then\ C\ else\ C} \mid \mathbf{if}_R\ \mathbf{RE}\ \mathbf{then\ RC\ else\ RC} \mid \mathbf{while}_D\ \mathbf{DE}\ \mathbf{do\ C} \mid \mathbf{while}_R\ \mathbf{RE}\ \mathbf{do\ RC}$$

We write $\textbf{if}_D\ b\ \textbf{then}\ c$ to abbreviate $\textbf{if}_D\ b\ \textbf{then}\ c\ \textbf{else skip}$ and likewise for $\textbf{if}_R\ b\ \textbf{then}\ c$.

The semantics of a command $c \in \mathbf{C}$ is denoted $[\![c]\!]$, which is a configuration transformer of type $(\mathbf{DetM} \times \mathbf{D}(\mathbf{RanM})) \to (\mathbf{DetM} \times \mathbf{D}(\mathbf{RanM}))$ as defined in Fig. 2.4.

$$[\![\textbf{skip}]\!](\sigma, \mu) = (\sigma, \mu) \tag{2.10}$$
$$[\![x_d \leftarrow e_d]\!](\sigma, \mu) = (\sigma[x_d \mapsto [\![e_d]\!]\sigma], \mu) \tag{2.11}$$
$$[\![x_r \leftarrow e_r]\!](\sigma, \mu) = (\sigma, \mathsf{bind}(\mu, m \mapsto \mathsf{unit}(m[x_r \mapsto [\![e_r]\!](\sigma, m)]))) \tag{2.12}$$
$$[\![x_r \leftarrow_\$ \mathbf{U}_S]\!](\sigma, \mu) = (\sigma, \mathsf{bind}(\mu, m \mapsto \mathsf{bind}(\mathsf{Unif}_S, u \mapsto \mathsf{unit}(m[x_r \mapsto u])))) \tag{2.13}$$
$$[\![c;\ c']\!](\sigma, \mu) = [\![c']\!]([\![c]\!](\sigma, \mu)) \tag{2.14}$$
$$[\![\textbf{if}_D\ b\ \textbf{then}\ c\ \textbf{else}\ c']\!](\sigma, \mu) = \begin{cases} [\![c]\!](\sigma, \mu) & : [\![b]\!]\sigma \neq \mathsf{false} \\ [\![c']\!](\sigma, \mu) & : [\![b]\!]\sigma = \mathsf{false} \end{cases} \tag{2.15}$$
$$[\![\textbf{if}_R\ b\ \textbf{then}\ c\ \textbf{else}\ c']\!](\sigma, \mu) = [\![c]\!](\sigma, \mu \mid [\![b]\!]\sigma \neq \mathsf{false}) \oplus_{\mu([\![b]\!]\sigma \neq \mathsf{false})} [\![c']\!](\sigma, \mu \mid [\![b]\!]\sigma = \mathsf{false}) \tag{2.16}$$
$$[\![\textbf{while}_D\ b\ \textbf{do}\ c]\!](\sigma, \mu) = [\![c;\ \cdots ; c]\!](\sigma, \mu) \tag{2.17}$$

FIGURE 2.4: Programming Language Semantics of PSL

The **skip** command (2.10) preserves the configuration unchanged. Deterministic assignment (2.11) updates the deterministic memory by giving the evaluated value of expression e_d to variable x_d, similar to assignment in Hoare Logic.

Random assignment (2.12) updates the distribution of random memory using the *bind* function. This operation transforms each memory in the support to the corresponding updated random memory while preserving its probability.

Random choice (2.13) updates the distribution μ using two *bind* functions. This operation extends each memory in the support to a new distribution, corresponding to a random choice over a uniform distribution on set S.

Sequential composition (2.14) and deterministic **If** statements (2.15) execute in the standard way, similar to Hoare Logic.

A random **If** statement (2.16) combines the two branches based on the probability of the condition being true or false, where the condition could be probabilistic.

A deterministic loop (2.17) is unfolded into a sequence of loop bodies since PSL assumes that the program always terminates, ensuring that such an unfolding must exist.

2.2.3 Assertion System

Given the probabilistic programming language, PSL requires an assertion system to express probabilistic information. It introduces three important assertions that distinguish it from Hoare Logic: the uniform distribution of random expressions, the equality of random expressions, and the separating conjunction for probabilistic independence.

However, formalizing the relationship between assertions and program states (configurations) is nontrivial. To achieve this, PSL follows a standard structure known as *bunched implication* (BI), proposed by O'Hearn and Pym [1999], and an instantiating method called the Kripke resource monoid, introduced by Pym et al. [2004] and Galmiche et al. [2005], both of which are commonly used in separation logics.

BI provides a structural foundation for the assertion system, indicating that assertions include atomic assertions ($p \in \mathbf{AP}$ where p is an atomic assertion and $\mathbf{AP}$ denotes the set of all atomic assertion), as well as **True** ($\top$), **False** ($\bot$), **And** ($\phi \wedge \psi$), **Or** ($\phi \vee \psi$), **Implication** ($\phi \to \psi$), **Separating Conjunction** ($\phi * \psi$), and **Separating Implication** ($\phi \mathrel{-\!\!*} \psi$).

In this framework, atomic assertions are flexible and can be adapted to specific requirements, whereas the other assertions adhere to the standard structure of separation logic. Specifically for PSL, atomic assertions capture probabilistic information, such as uniform distributions, equality of random expressions, and standard relations on deterministic expressions.

Definition 2.3 (Assertions)**.** Assertions, ϕ, ψ etc. are defined as:
$\phi, \psi ::= p \mid \top \mid \bot \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \to \psi \mid \phi * \psi \mid \phi \mathrel{-\!\!*} \psi$
where $\mathbf{AP} \ni p ::= \mathbf{U}_S[\mathbf{RE}] \mid \mathbf{RE} \sim \mathbf{RE} \mid \mathbf{DE} = \mathbf{DE} \mid \mathbf{DE} \leq \mathbf{DE} \mid \cdots$

Leveraging the Kripke resource monoid, PSL introduces the following definitions to model the separation and combination of configurations.

For any $S \subseteq \mathbf{RV}$, the set of all corresponding memories is denoted as $\mathbf{RanM}(S) = S \to \mathbf{Val}$. Furthermore, for any probability distribution $\mu \in \mathbf{D}(\mathbf{RanM}(S))$, the *domain* of μ, written as $\mathsf{dom}(\mu)$, is defined as S. This notion formalizes the concept of partial configurations which are potentially seperated from a complete configuration.

Given two disjoint sets $S, S' \subseteq \mathbf{RV}$ and two distributions of memories over them, $\mu_S \in \mathbf{D}(\mathbf{RanM}(S))$ and $\mu_{S'} \in \mathbf{D}(\mathbf{RanM}(S'))$, PSL defines their product as follows: $\mu_S \otimes \mu_{S'} = (\lambda m.\ \mu_S(p_S(m)) \cdot \mu_{S'}(p_{S'}(m)))$, where m is a memory (of type $\mathbf{RV} \to \mathbf{Val}$), and $p_S(m)$ extracts the sub-map of m over S. If $v \notin S$, then $p_S(m)(v)$ is undefined.

For a subset $S' \subseteq S$ and a distribution μ over S, PSL defines $\pi_{S,S'}$ to yield its marginal distribution over S': $\pi_{S,S'}(\mu) = \lambda m'.\ \Sigma_{m \in \{m | p_{S'}(m) = m'\}} \mu(m)$, where $p_{S'}(m)$ extracts the sub-map of m over S'. For instance, if $S = \{x, y, z\}$, and $S' = \{x\}$, and if μ is a distribution over S talking about these three random variables, then $\pi_{S,S'}(\mu)$ is the marginal distribution of x. We sometimes omit S because it is always $\mathsf{dom}(\mu)$.

A partial binary operation, representing the combination of two separated configurations, is defined as follows: $(\sigma, \mu) \circ (\sigma', \mu') = \begin{cases} (\sigma \cup \sigma', \mu \otimes \mu') & : \sigma = \sigma' \text{ on } \mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma') \\ & \quad \text{and } \mathsf{dom}(\mu) \cap \mathsf{dom}(\mu') = \emptyset \\ \text{undefined} & : \text{otherwise} \end{cases}$

We write $(m \circ m') \downarrow$ to say $(m \circ m')$ is defined (which happens if and only if the domains of the m and m' do not overlap).

A partial order, indicating that one configuration is a marginal distribution of another, is defined as: $(\sigma, \mu) \sqsubseteq (\sigma', \mu')$ iff $\begin{cases} \mathsf{dom}(\sigma) \subseteq \mathsf{dom}(\sigma') \text{ and } \sigma = \sigma' \text{ on } \mathsf{dom}(\sigma) \\ \mathsf{dom}(\mu) \subseteq \mathsf{dom}(\mu') \text{ and } \mu = \pi_{\mathsf{dom}(\mu)}(\mu') \end{cases}$

Finally, let $m \models \phi$ denote that the partial configuration m satisfies assertion ϕ. The assertion semantics is then defined as Fig. 2.5.

The first part defines the semantics of atomic assertions. A configuration (σ, μ) satisfies $\mathbf{U}_S[e_r]$ if and only if all free variables in e_r are within the domain of the configuration and the value of e_r follows a uniform distribution over S. In PSL, $e_r \sim e'_r$ denotes equality for random variables, meaning that for every memory in the support of μ, the value of e_r and e'_r must be equal. The remaining semantics (some of which are omitted) for deterministic expressions only consider deterministic memory and thus follow standard conventions.

The second part defines the semantics of other structural assertions. The assertions **True** ($\top$), **False** ($\bot$), **And** ($\phi \wedge \psi$), **Or** ($\phi \vee \psi$), **Implication** ($\phi \to \psi$) have their intuitive meanings. A configuration m satisfies $\phi * \psi$ if and only if m can be separated into two configurations, m_1 and m_2, such that their combination $m_1 \circ m_2 \sqsubseteq m$, with

$(\sigma, \mu) \models \mathbf{U}_S[e_r]$	iff $\mathsf{FV}(e_r) \subset \mathsf{dom}(\sigma) \cup \mathsf{dom}(\mu) \wedge$ $[\![e_r]\!](\sigma, \mu)$ assigns probability $1/\lvert S\rvert$ to each element of S
$(\sigma, \mu) \models e_r \sim e'_r$	iff $\mathsf{FV}(e_r) \cup \mathsf{FV}(e'_r) \subset \mathsf{dom}(\sigma) \cup \mathsf{dom}(\mu) \wedge$ $(\forall m \in \mathsf{supp}(\mu).\ [\![e_r]\!](\sigma, m) = [\![e_r]\!](\sigma, m))$
$(\sigma, \mu) \models e_d = e'_d$	iff $\mathsf{FV}(e_d) \subset \mathsf{dom}(\sigma) \wedge [\![e_d]\!](\sigma) = [\![e'_d]\!](\sigma)$
$(\sigma, \mu) \models e_d \leq e'_d$	iff $\mathsf{FV}(e_d) \subset \mathsf{dom}(\sigma) \wedge [\![e_d]\!](\sigma) \leq [\![e'_d]\!](\sigma)$
$\ldots$	

$m \models \top$	always
$m \models \bot$	never
$m \models \phi \wedge \psi$	iff $m \models \phi$ and $m \models \psi$
$m \models \phi \vee \psi$	iff $m \models \phi$ or $m \models \psi$
$m \models \phi \to \psi$	iff for all $m \sqsubseteq m', m' \models \phi$ implies $m' \models \psi$
$m \models \phi * \psi$	iff exist m_1, m_2. $(m_1 \circ m_2)\downarrow$ and $m_1 \circ m_2 \sqsubseteq m$ and $m_1 \models \phi$ and $m_2 \models \psi$
$m \models \phi \mathrel{-\!\!*} \psi$	iff for all $m' \models \phi$, $(m \circ m')\downarrow$ implies $m \circ m' \models \psi$

FIGURE 2.5: Assertion Semantics of PSL

m_1 satisfying ϕ and m_2 satisfying ψ. The final assertion is **Separating Implication** $(\phi \mathrel{-\!\!*} \psi)$ and is defined but never used in PSL [Barthe et al., 2019].

2.2.4 Inference Rules

PSL introduces the concept of a triple in a manner that mirrors the intuition behind the Hoare Logic triple presented in Section 2.1.

Definition 2.4 (PSL Triple)**.** A PSL triple is written as $\vdash \{\phi\}\ c\ \{\psi\}$, where ϕ and ψ are assertions and c is a command. This triple is considered valid if and only if for every configuration (σ, μ) that satisfies ϕ, the final configuration $[\![c]\!](\sigma, \mu)$ produced by the execution of c satisfies ψ.

The inference rules are introduced in Fig. 2.6, with relevant definitions provided in Fig. 2.7. Additionally, we present key rules pertinent to our work in Chapter 3.

The Constant Rule (CONST) establishes that an assertion η remains valid after executing c if all free variables in η, denoted $\mathsf{FV}(\eta)$, are not modified. Here, $\mathsf{MV}(c)$ represents the set of **possibly** modified variables.

$$\textsc{DAssign}\quad \vdash \{\phi[e_d/x_d]\}\ x_d \leftarrow e_d\ \{\phi\} \qquad \textsc{Skip}\quad \vdash \{\phi\}\ \mathbf{skip}\ \{\phi\} \qquad \textsc{Seqn}\quad \frac{\vdash \{\phi\}\ c\ \{\psi\} \qquad \vdash \{\psi\}\ c'\ \{\eta\}}{\vdash \{\phi\}\ c; c'\ \{\eta\}}$$

$$\textsc{Weak}\quad \frac{\vdash \{\phi\}\ c\ \{\psi\} \qquad \models \phi' \to \phi \qquad \models \psi \to \psi'}{\vdash \{\phi'\}\ c\ \{\psi'\}} \qquad \textsc{Const}\quad \frac{\vdash \{\phi\}\ c\ \{\psi\} \qquad \mathsf{FV}(\eta) \cap \mathsf{MV}(c) = \emptyset}{\vdash \{\phi \wedge \eta\}\ c\ \{\psi \wedge \eta\}}$$

$$\textsc{RAssn}\quad \frac{x_r \notin \mathsf{FV}(e_r)}{\vdash \{\top\}\ x_r \leftarrow e_r\ \{x_r \sim e_r\}} \qquad \textsc{DCond}\quad \frac{\vdash \{\phi \wedge b = \mathsf{true}\}\ c\ \{\psi\} \qquad \vdash \{\phi \wedge b = \mathsf{false}\}\ c'\ \{\psi\}}{\vdash \{\phi\}\ \mathbf{if}_D\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \{\psi\}}$$

$$\textsc{RSamp}\quad \vdash \{\top\}\ x_r \leftarrow_\$ \mathbf{U}_S\ \{\mathbf{U}_S[x_r]\} \qquad \textsc{DLoop}\quad \frac{\vdash \{\phi \wedge b = \mathsf{true}\}\ c\ \{\phi\}}{\vdash \{\phi\}\ \mathbf{while}_D\ b\ \mathbf{do}\ c\ \{\phi \wedge b = \mathsf{false}\}}$$

$$\textsc{RDCond}\quad \frac{\vdash \{\phi \wedge b \sim \mathsf{true}\}\ c\ \{\psi\} \qquad \vdash \{\phi \wedge b \sim \mathsf{false}\}\ c'\ \{\psi\} \qquad \models \phi \to (b \sim \mathsf{true} \vee b \sim \mathsf{false})}{\vdash \{\phi\}\ \mathbf{if}_R\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \{\psi\}}$$

$$\textsc{RCond}\quad \frac{\vdash \{\phi * b \sim \mathsf{true}\}\ c\ \{\psi * b \sim \mathsf{true}\} \qquad \vdash \{\phi * b \sim \mathsf{false}\}\ c'\ \{\psi * b \sim \mathsf{false}\} \qquad \psi \in \mathbf{SP}}{\vdash \{\phi * \mathbf{D}(b)\}\ \mathbf{if}_R\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \{\psi * \mathbf{D}(b)\}}$$

$$\textsc{True}\quad \vdash \{\top\}\ c\ \{\top\} \qquad \textsc{Conj}\quad \frac{\vdash \{\phi_1\}\ c\ \{\psi_1\} \qquad \vdash \{\phi_2\}\ c\ \{\psi_2\}}{\vdash \{\phi_1 \wedge \phi_2\}\ c\ \{\psi_1 \wedge \psi_2\}} \qquad \textsc{Case}\quad \frac{\vdash \{\phi_1\}\ c\ \{\psi_1\} \qquad \vdash \{\phi_2\}\ c\ \{\psi_2\}}{\vdash \{\phi_1 \vee \phi_2\}\ c\ \{\psi_1 \vee \psi_2\}}$$

$$\textsc{RCase}\quad \frac{\vdash \{\phi * b \sim \mathsf{true}\}\ c\ \{\psi * b \sim \mathsf{true}\} \qquad \vdash \{\phi * b \sim \mathsf{false}\}\ c\ \{\psi * b \sim \mathsf{false}\} \qquad \psi \in \mathbf{SP}}{\vdash \{\phi * \mathbf{D}(b)\}\ c\ \{\psi * \mathbf{D}(b)\}}$$

$$\textsc{Frame}\quad \frac{\vdash \{\phi\}\ c\ \{\psi\} \qquad \mathsf{FV}(\eta) \cap \mathsf{MV}(c) = \emptyset \qquad \mathsf{FV}(\psi) \subseteq T \cup \mathsf{RV}(c) \cup \mathsf{WV}(c) \qquad \models \phi \to \mathbf{D}(T \cup \mathsf{RV}(c))}{\vdash \{\phi * \eta\}\ c\ \{\psi * \eta\}}$$

FIGURE 2.6: Inference Rules of PSL

$$\mathsf{RV}(x_r \leftarrow e_r) = \mathsf{FV}(e_r), \quad \mathsf{RV}(x_r \leftarrow_\$ \mathbf{U}_S) = \emptyset, \quad \mathsf{RV}(\mathbf{while}_D\ b\ \mathbf{do}\ c) = \mathsf{RV}(c)$$

$$\mathsf{RV}(c; c') = \mathsf{RV}(c) \cup (\mathsf{RV}(c') - \mathsf{WV}(c)), \quad \mathsf{RV}(\mathbf{if}_D\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{RV}(c) \cup \mathsf{RV}(c')$$

$$\mathsf{RV}(\mathbf{if}_R\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{RV}(c) \cup \mathsf{RV}(c') \cup \mathsf{FV}(b)$$

$$\mathsf{WV}(x_r \leftarrow e_r) = \{x_r\} - \mathsf{FV}(e_r), \quad \mathsf{WV}(x_r \leftarrow_\$ \mathbf{U}_S) = \{x_r\}, \quad \mathsf{WV}(\mathbf{while}_D\ b\ \mathbf{do}\ c) = \emptyset$$

$$\mathsf{WV}(c; c') = \mathsf{WV}(c) \cup (\mathsf{WV}(c') - \mathsf{RV}(c)), \quad \mathsf{WV}(\mathbf{if}_D\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{WV}(c) \cap \mathsf{WV}(c')$$

$$\mathsf{WV}(\mathbf{if}_R\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = (\mathsf{WV}(c) \cap \mathsf{WV}(c')) - \mathsf{FV}(b)$$

$$\mathsf{MV}(x_r \leftarrow e_r) = \{x_r\}, \quad \mathsf{MV}(x_r \leftarrow_\$ \mathbf{U}_S) = \{x_r\}, \quad \mathsf{MV}(\mathbf{while}_D\ b\ \mathbf{do}\ c) = \mathsf{MV}(c)$$

$$\mathsf{MV}(c; c') = \mathsf{MV}(c) \cup \mathsf{MV}(c'), \quad \mathsf{MV}(\mathbf{if}_D\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{MV}(c) \cup \mathsf{MV}(c')$$

$$\mathsf{MV}(\mathbf{if}_R\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{MV}(c) \cup \mathsf{MV}(c')$$

FIGURE 2.7: Auxiliary Functions

The Frame Rule is familiar from separation logics and allows adding irrelevant and independent information to both the precondition and postcondition. However, in PSL, it is subject to several side conditions. Specifically, $\mathsf{WV}(c)$ denotes all variables that **must** be written by c before potentially being read, while $\mathsf{RV}(c)$ represents the set of variables that c may read from. By definition, $\mathsf{WV}(c)$ is a subset of $\mathsf{MV}(c)$.

The side conditions of the Frame Rule ensure that:

- No variable in the frame η is modified, similar to the Constant Rule.
- η is initially independent of all read variables, implying its independence from all variables in ϕ as well as all written variables, which depend only on read variables and are thus initially independent of the framing condition.

The Random Sampling rule (RSAMP) ensures that after making a random choice over a uniform distribution on a static set S, the resulting distribution remains uniform—an intuitive property. Typically, this rule is used in conjunction with the Frame rule to incorporate additional information into the precondition and postcondition.

The Random Condition rule (RCOND) follows the classical form but introduces two additional side conditions. First, the distribution of the if condition b must be independent

of the precondition ϕ; otherwise, combining ϕ with $b \sim \mathsf{true}$ may lead to contradictions, for example, when $\phi = \mathbf{U}_{\mathsf{true},\mathsf{false}}[b]$. Second, the postcondition must be *supported*, denoted $\psi \in \mathbf{SP}$, ensuring that the composition of the two configurations resulting from the two if branches still satisfies ψ.

However, we discovered during our work that [Barthe et al., 2019]'s definition of *supported* contains certain oversights. We will introduce a corrected version and discuss these oversights in detail in Section 3.6.

All deterministic rules (prefixed with "D"), including the Skip rule, True rule, Sequence rule, Conjunction rule, and Case rule, are standard and have direct counterparts in Hoare Logic. The RCase rule is a generalization of the RCond rule, while the RDCond rule handles the special case where a random variable stores a deterministic value.

2.2.5 Summary and Relation to Our Work

PSL mirrors the structure of Hoare Logic but introduces additional complexity by incorporating random sampling, allowing reasoning about probabilistic programs. To capture the memory state (or configurations) of a probabilistic programming language, PSL develops its own assertion system. For simplicity, PSL does not use a complete assertion system, which means some configuration (e.g. non-uniform distribution) cannot be precisely described by PSL assertions. This simplification arises from the extensive detail inherent in probabilistic distributions, as PSL is designed to focus primarily on probabilistic independence. Consequently, identifying key features and crafting a suitable assertion system for them is a crucial step in developing program logic for complex programming languages.

As a separation logic, PSL includes a Frame Rule that enables local reasoning about programs. This means that assertions describe only specific parts of a configuration, and these parts can be composed when their domains do not overlap. To ensure the soundness of the Frame Rule, certain properties, such as monotonicity, are essential in the assertion system. Designing an assertion system that meets these requirements is non-trivial, making existing structures like BI [O'Hearn and Pym, 1999] and Kripke resource models [Galmiche et al., 2005] particularly useful.

As discussed in Section 2.2.4, certain PSL rules (e.g. RCond and RSamp) impose restrictions that limit their applicability to the verification of complex algorithms, such as the path ORAM scheme proposed by Stefanov et al. [2018]. To overcome these limitations, we will propose a novel reasoning strategy and integrate it with PSL to relax some of these restrictions in Chapter 3. Furthermore, we illustrate the practical usefulness of our approach by verifying the security of path ORAM in Chapter 5.

2.3 Encryption in Oblivious Algorithms

Oblivious algorithms often employ encryption to conceal secrets; however, practical encryption schemes do not yield uniformly distributed ciphertexts, which complicates the verification of such algorithms. We address this challenge in Chapter 4.

In this section, we present three levels of encryption guarantees—perfect security, statistical security, and computational security in the following three subsection respectively. Readers may choose to skip these subsections and return to them as needed when reading Chapter 4. We summarize them and introduce their relation to our work in Section 2.3.4.

2.3.1 Perfect Security

Perfect security, also known as *information-theoretic security*, is originally introduced by Shannon [1949] and discussed in standard cryptography textbooks [Katz and Lindell, 2014]. It represents the strongest form of confidentiality, ensuring that the ciphertext reveals no information about the plaintext, even to adversaries with unlimited computational power.

Definition 2.5 (Perfect Secrecy [Katz and Lindell, 2014, Section 2.1])**.** An encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over message space $\mathcal{M}$, key space $\mathcal{K}$, and ciphertext space $\mathcal{C}$ satisfies *perfect secrecy* if for every message $m_0, m_1 \in \mathcal{M}$ and every ciphertext $c \in \mathcal{C}$,

$$\mathbb{P}\left[\mathsf{Enc}_K(m_0) = c\right] = \mathbb{P}\left[\mathsf{Enc}_K(m_1) = c\right],$$

where the probabilities $\mathbb{P}[\cdot]$ are over choice of the key K by Gen and any randomness of Enc.

In this thesis, we often write $\mathsf{enc}(m)$ to denote $\mathsf{Enc}_K(m)$ as defined above, viewed as a distribution over ciphertexts induced by the random choice of the key K and any internal randomness used by Enc.

This definition implies that observing the ciphertext does not help distinguish between different plaintexts, as the ciphertext distribution is the same for every possible message.

Example: The One-Time Pad. A classical example of a perfectly secret encryption scheme is the *one-time pad*. Let $\mathcal{M} = \mathcal{K} = \mathcal{C} = \{0,1\}^n$, and define the scheme as follows:

- $\mathsf{Gen}()$: return a key $k \leftarrow \{0,1\}^n$ uniformly at random.
- $\mathsf{Enc}_k(m)$: return $c = m \oplus k$.
- $\mathsf{Dec}_k(c)$: return $m = c \oplus k$.

Assuming a truly random key used only once, the ciphertext is uniformly distributed and independent of the plaintext, thus satisfying perfect secrecy.

PSL can express the security of the one-time pad using the triple

$$\vdash \{\mathbf{D}(m)\}\ k \leftarrow_{\$} \mathbf{U}_{\{0,1\}^n};\ c = m\ \mathsf{xor}\ k;\ \{\mathbf{D}(m) * \mathbf{U}_{\{0,1\}^n}[c]\}$$

which states that for any distribution over messages m, the resulting ciphertext c is uniformly distributed over $\{0,1\}^n$, where n denotes the length of m. Moreover, c is probabilistic independent of m. Nevertheless, PSL captures only this specific instance of a perfectly secret encryption scheme. Reasoning about broader classes of encryption schemes will be discussed in the context of oblivious algorithms in Chapter 4.

Limitations. While theoretically appealing, perfect secrecy requires the key to be as long as the message, chosen uniformly at random, and used only once. These stringent constraints make such schemes impractical for general-purpose use. In the following subsections, we introduce weaker—but more practical—security notions that relax these requirements while still providing meaningful confidentiality guarantees.

2.3.2 Statistical Security

Statistical security, also known as *information-theoretic security with bounded leakage*, relaxes the strict requirements of perfect secrecy by allowing a small but quantifiable amount of information to leak from the ciphertext. The leakage is measured using the *statistical distance* between the ciphertext distributions of different messages.

Definition 2.6 (Statistical Distance)**.** Given two distributions D_0 and D_1 over a finite set $\mathcal{X}$, their *statistical distance* is defined as

$$\Delta(D_0, D_1) := \frac{1}{2} \sum_{x \in \mathcal{X}} \left| \mathbb{P}\left[D_0 = x\right] - \mathbb{P}\left[D_1 = x\right] \right|.$$

Definition 2.7 (Statistical Secrecy for Encryption [Vadhan, 2013, Definition 9])**.** An encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over message space $\mathcal{M}$, key space $\mathcal{K}$, and ciphertext space $\mathcal{C}$ is said to be ε-statistically secure if for all $m_0, m_1 \in \mathcal{M}$, the statistical distance between the ciphertext distributions satisfies

$$\Delta(\mathsf{enc}(m_0), \mathsf{enc}(m_1)) \leq \varepsilon,$$

where $\mathsf{enc}(m)$ denotes the distribution of $\mathsf{Enc}_K(m)$ over the randomness of $K \leftarrow \mathsf{Gen}()$ and any internal randomness of Enc, and $\Delta(\cdot, \cdot)$ denotes the statistical distance between distributions.

Intuitively, ε-statistical security ensures that an adversary cannot distinguish between the encryptions of any two messages with advantage greater than ε, regardless of their computational power. When $\varepsilon = 0$, the scheme satisfies perfect secrecy. As ε increases, the ciphertexts of different messages become more distinguishable.

Generally, we requires the ε to be a negligible function of a predefined security parameter.

Definition 2.8 (Negligible Function)**.** A function $\varepsilon : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is called *negligible*, denoted $\mathsf{negl}(\cdot)$, if for every positive polynomial $\mathsf{poly}(\cdot)$, there exists an integer N such that for all $\lambda > N$,

$$\varepsilon(\lambda) < \frac{1}{\mathsf{poly}(\lambda)}.$$

In other words, $\varepsilon(\lambda)$ vanishes faster than the inverse of any polynomial in λ as $\lambda \to \infty$.

If an encryption scheme is $\varepsilon(\lambda)$-statistically secure where ε is negligible, then the ciphertexts corresponding to any two messages are indistinguishable to any adversary, even with unbounded computational power, except with negligible advantage. This provides a strong, albeit slightly weaker, alternative to perfect secrecy.

Practical Considerations. Statistical security allows for shorter keys than perfect secrecy while still providing strong guarantees. For example, one can design schemes where the key is logarithmic in the message length, at the cost of allowing a small distinguishing advantage.

However, most modern cryptographic protocols adopt a weaker, yet widely accepted, notion of secrecy—*computational security*—in exchange for improved efficiency and flexibility, such as using keys of constant length. This notion assumes that adversaries are bounded by computational resources and defines security in terms of their inability to distinguish ciphertexts within feasible time bounds. We elaborate on this notion in the following subsection.

2.3.3 Computational Security

Computational security is the weakest among the three secrecy notions discussed in this chapter, yet it is the most practical and widely adopted in modern cryptography.

It relaxes the requirements of statistical secrecy by only protecting against adversaries that are computationally bounded—typically modeled as probabilistic polynomial-time (PPT) algorithms. The central idea is that while ciphertexts may, in principle, leak information about the plaintext, this information cannot be feasibly extracted by any efficient algorithm.

A foundational formalization of computational secrecy is *indistinguishability under chosen-plaintext attacks* (IND-CPA) [Goldwasser and Micali, 1984], where an adversary attempts to distinguish between the encryptions of two chosen plaintexts without access to a decryption oracle.

A strong and widely accepted formalization of computational secrecy is *indistinguishability under chosen-ciphertext attacks* (IND-CCA) [Bellare et al., 1998]. It strengthens

the notion of IND-CPA by allowing the adversary to access a decryption oracle, modeling scenarios where attackers may obtain decryptions of arbitrary ciphertexts (except the challenge) as part of their strategy.

Here we only introduce the definition of IND-CCA because this thesis will work on IND-CCA and it implies IND-CPA.

Definition 2.9 (IND-CCA Game for Multiple Messages [Katz and Lindell, 2014, Section 3.7])**.** Given a PPT adversary $\mathcal{A}$, an encryption oracle $\mathsf{enc}()$, a decryption oracle $\mathsf{dec}()$, the CCA indistinguishability game for multiple messages is defined as:

for $j \in \{1, \ldots, m_1\}$, where m_1 is in $\mathsf{poly}(k)$:

$\mathcal{A}$ picks an input for $\mathsf{enc}()$ or $\mathsf{dec}()$ and then get the result.

$\mathcal{A}$ outputs two sequences of messages M_0 and M_1 of the same length l which is in $\mathsf{poly}(k)$.

A uniform bit $b \in \{0, 1\}$ is chosen, then $\mathsf{enc}()$ is called on each message of sequence M_b. The results (ciphertexts) are stored in C and given to $\mathcal{A}$.

for $j \in \{1, \ldots, m_1\}$, where m_1 is in $\mathsf{poly}(k)$:

$\mathcal{A}$ picks an input for $\mathsf{enc}()$ or $\mathsf{dec}()$ and then get the result,

but $\mathcal{A}$ cannot pick any ciphertexts in C for $\mathsf{dec}()$.

Eventually, $\mathcal{A}$ outputs a bit b'. $\mathcal{A}$ wins the game if and only if $b' = b$.

The adversary's goal is to determine which of two chosen messages was encrypted, even when allowed to query a decryption oracle on arbitrary ciphertexts, excluding the challenge (ciphertext). IND-CCA security ensures that no efficient adversary can distinguish the ciphertexts with non-negligible advantage, even under such powerful attacks.

Definition 2.10 (IND-CCA Security)**.** Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an encryption scheme. We write $\mathsf{enc}(\cdot)$ and $\mathsf{dec}(\cdot)$ to denote the encryption and decryption oracles, respectively, which internally run Enc and Dec using a fixed secret key K sampled at random from $\mathsf{Gen}(1^\lambda)$. We say $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *IND-CCA secure for multiple messages* if for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in the IND-CCA game for multiple messages using $\mathsf{enc}(\cdot)$ and $\mathsf{dec}(\cdot)$ is negligible in the security parameter λ:

$$\left| \mathbb{P}\left[b' = b\right] - \frac{1}{2} \right| \leq \varepsilon(\lambda), \text{ where } \varepsilon \text{ is a negligible function.}$$

IND-CCA security is considered the gold standard for encryption in oblivious algorithms. Practical IND-CCA-secure schemes include authenticated encryption constructions such as `Encrypt-then-MAC` [Bellare and Namprempre, 2000] and `AES-GCM` [McGrew and Viega, 2007].

2.3.4 Summary and Relation to Our Work

In this section, we introduced three encryption standards. Among them, computational security is the most commonly used in practice. However, it is also the most challenging to formally reason about. This difficulty arises from its game-based definition (Definition 2.9), which lacks a precise specification for the distribution of ciphertexts. Instead, it defines security through a complex, interactive game that depends on the adversary's computational power.

As a result, many works opt to sidestep the associated proofs. For instance, several oblivious algorithms [Sasy and Ohrimenko, 2019, Stefanov et al., 2018, Shi, 2019] and verification frameworks [Son et al., 2021] assume that an attacker can observe only the memory access patterns, but not the actual data values. This assumption significantly simplifies formal verification. Such a simplification is often justified: if the data is encrypted, then in practice, an adversary cannot extract meaningful information from it—rendering it effectively equivalent to unobserved data.

However, this approach implicitly assumes that encryption is always correctly applied. If a buggy program omits encryption in certain cases, the verification may still succeed under this model, as the plaintext remains unobservable to the attacker by assumption.

To address this gap, we propose a formal verification approach that explicitly checks for the correct application of encryption within oblivious algorithms, as developed in Chapter 4. Our method introduces two precise security definitions—based on either computational or statistical secrecy—and provides a verification framework to establish when an oblivious algorithm satisfies this definition. This, in turn, ensures that encryption is correctly and consistently used to uphold the intended security guarantees.

2.4 Related Works

In the preceding sections, we presented several foundational works in detail, which are essential for understanding the main contributions of Chapters 3, 4, and 5. This section introduces related works that, while not the focus of our study, run parallel to our research direction.

2.4.1 Oblivious Algorithms

Oblivious algorithms are designed to prevent the leakage of sensitive information through observable side channels, such as memory access patterns. In particular, these algorithms ensure that their access behavior is independent of the underlying data [Goldreich and Ostrovsky, 1987], making them especially valuable in secure computation and privacy-preserving systems.

Classical oblivious algorithms are typically deterministic and are constructed to access memory in a fixed or data-independent manner. Notable examples include sorting networks such as Batcher's sort [Batcher, 1968] and some classic oblivious RAM (ORAM) constructions [Goldreich and Ostrovsky, 1996].

To improve performance while preserving security guarantees, recent research introduces *probabilistic oblivious algorithms*. These algorithms allow randomised access patterns whose distributions are designed to hide data-dependent behaviors. The key idea is that while individual executions may differ, the distribution over access patterns remains statistically or computationally indistinguishable for different inputs. This relaxation permits greater algorithmic flexibility and efficiency, especially in settings where full determinism is too restrictive [Pinkas and Reinman, 2010].

Prominent examples include Randomized Shellsort [Goodrich, 2010], Circuit ORAM [Wang et al., 2015], Optimal ORAM [Asharov et al., 2022] and the probabilistic oblivious algorithms described and formally verified in our work, including path ORAM [Stefanov et al., 2018], as presented in Chapter 5. Some oblivious applications depend on these oblivious building blocks, such as ObliDB [Eskandarian and Zaharia, 2019] and other data structures [Wang et al., 2014], which are built on ORAM.

A notable feature of probabilistic oblivious algorithms is that they typically incur a small *failure probability*—the chance that a particular execution reveals some information about the input due to insufficient obfuscation of access patterns. This probability arises from the inherent randomness used to simulate uniform access and is often negligible, typically bounded by an inverse polynomial in a security parameter.

In many designs, such failure cannot be completely eliminated without incurring significant costs in terms of memory or runtime. The trade-off between failure probability and auxiliary memory is well-studied: increasing the available memory space often enables better isolation between randomised traces and reduces the likelihood of collisions or leakage. In particular, many randomised schemes (e.g. randomised hash-based ORAMs or sampling-based oblivious algorithms) exhibit a failure probability that decays exponentially with respect to the memory buffer size or stash capacity [Goodrich and Mitzenmacher, 2012, Wang et al., 2015].

2.4.2 Related Program Logics and Verification

A variety of program logics have been proposed for reasoning about probabilistic programs and for verifying obliviousness. These approaches differ in their underlying assumptions and aims. In the following subsections, we review representative strands of this literature, highlighting both their strengths and their limitations in relation to our goals.

2.4.2.1 Probabilistic Coupling and EasyCrypt

Probabilistic coupling—formalised in probabilistic Relational Hoare Logic (pRHL) Barthe et al. [2012] and implemented in verification frameworks such as EasyCrypt Barthe et al. [2014a]—is a standard technique for establishing the security of probabilistic algorithms. In this setting, one proves that for any two executions with different secret inputs, there exists a relational proof (a pRHL judgment) that constructs a coupling between the two induced distributions. This coupling witnesses that every probabilistic choice in one execution can be matched with a corresponding choice in the other, ultimately ensuring that the resulting output distributions are identical (or indistinguishable, depending on the security notion).

However, for dynamic random choices (introduced in detail in Chapter 3), a bijective probabilistic coupling may fail to exist or may even be undefined (e.g., the sampling construction of [Sasy and Ohrimenko, 2019] discussed in Section 5.1). In such settings, identifying a suitable coupling can be substantially more difficult than proving the desired security property directly via probabilistic independence. Indeed, the original informal security arguments of many oblivious algorithms—including all of our case studies [Sasy and Ohrimenko, 2019, Stefanov et al., 2018, Shi, 2019, Ohrimenko et al., 2014]—rely on establishing appropriate independence properties, rather than constructing couplings.

While EasyCrypt can prove independence as a final conclusion, its proof architecture is not designed to use independence as an intermediate structuring principle. Instead, EasyCrypt's reasoning framework is centred around pRHL-style couplings, which guide the construction of relational proofs between program runs.

Moreover, EasyCrypt also supports reasoning about the statistical distance between the output distributions of algorithms, a capability that will be directly relevant to the questions introduced in Section 3.7.

Extensions to this method, including approximate probabilistic coupling [Barthe et al., 2013, 2016a,b], are widely used for the verification of differential privacy [Dwork et al., 2006].

2.4.2.2 Verification of Obliviousness

Several program logics have been developed for verifying obliviousness.

ObliCheck [Son et al., 2021] and λ_{OADT} [Ye and Delaware, 2022] can check or prove obliviousness but only for deterministic algorithms.

λ_{obliv} [Darais et al., 2019] is a type system for a functional language for proving obliviousness of probabilistic algorithms but it forbids branching on secrets, which is prevalent in many oblivious algorithms including those in Chapter 5. It also forbids outputting a probabilistic value (and all other values influenced by it) more than once, to avoid leaking information by the dependence of two outputs. Our approach (Chapter 3) suffers no such restriction.

Path ORAM (one of our case studies in Chapter 5) has received some verification attention [Sahai et al., 2020, Leung et al., 2023]. [Sahai et al., 2020] reason about this algorithm but in a non-probabilistic model, instead representing it as a nondeterministic transition system, and apply model counting to prove a security property about it. Their property says that for any observable output, there is a sufficient number of inputs to hide which particular input would have produced that output. This specification seems about the best that can be achieved for a nondeterministic model of the algorithm, but would also hold for an implementation that used biased choices (which would necessarily reveal too much of the input). Ours instead says that for each input the output is identically distributed (according to the adversary's observational power), and would not be satisfied for such a hypothetical implementation.

Hannah Leung et al. [Leung et al., 2023] recently proposed to verify this algorithm in Coq, but as far as we are aware ours is the first verification of Path ORAM via a probabilistic program logic.

2.4.2.3 Probabilistic Separation Logic Extensions

Several recent works extend or refer to PSL in different ways.

Jereb and Simpson [2025] revisits the ideas of Probabilistic Separation Logic (PSL) in the context of Separation Logic [Reynolds, 2002], whereas the original PSL was formulated atop Hoare Logic. In Hoare Logic and PSL: the triple $\vdash \{\mathsf{true}\}\ b \leftarrow a\ \{b = a \wedge c = c\}$ is valid. However, in Separation Logic and in the extension proposed by Jereb and Simpson [2025], this triple becomes invalid since the precondition does not mention the variables a, b, or c. This distinction arises from a fundamental semantic difference: in Hoare Logic and PSL, assignment commands cannot fail because every variable is assumed to exist, while in Separation Logic and in the extension under discussion, assignment may fail if it accesses variables or memory locations not specified in the precondition.

Consequently, the precondition must explicitly mention all variables accessed by the command (e.g. a and b in the example), as well as those referenced in the postcondition (e.g. c). This requirement implicitly enforces the two side-conditions present in PSL's frame rule, thereby allowing a simpler form of the rule to emerge naturally. Furthermore,

this extension enables reasoning about both probabilistic independence and memory safety within a unified framework.

Lilac [Li et al., 2023] also uses separating conjunction to model probabilistic independence. Crucially, it supports reasoning about conditional probability and conditional independence; [Li et al., 2024] validated the design decisions of Lilac. However, Lilac's programming language is functional whereas ours is imperative. Lilac does not support random loops or dynamic random choice, which are essential for our aim.

Bao et al. [2021a] extends Probabilistic Separation Logic (PSL) to reason about conditional independence by introducing a more expressive assertion system. However, the associated program logic lacks support for loops, limiting its applicability to practical oblivious algorithms despite its theoretical significance. [Lago et al., 2024] extended PSL to computational security, but it cannot deal with loops (neither deterministic nor probabilistic) so their target algorithms are very different to ours.

To reason about negative dependence, Bao et al. [Bao et al., 2021b] developed the LINA logic, which supports assertions about anti-correlated random variables. The BlueBell logic [Bao et al., 2025] combines relational reasoning and probabilistic independence, extending Lilac to support relational verification of probabilistic programs.

Tassarotti and Harper [Tassarotti and Harper, 2019] also proposed a concurrent probabilistic separation logic using coupling arguments to reason about randomised concurrency. BaSL [Ho et al., 2025] supports reasoning about Bayesian conditioning, providing a formal, compositional framework for verifying statistical properties of Bayesian probabilistic programs.

IVL [Schröer et al., 2023] reasons about probabilistic programs with nondeterminism. In doing so it supports classical reasoning (e.g. for the nondeterministic parts) and probabilistic reasoning for the probabilistic parts. Our work (Chapter 3) reasons only about probabilistic programs (with no nondeterminism) but allows using classical reasoning to reason about parts of the probabilistic program, and for the classical and probabilistic reasoning styles to interact and enhance each other.

Lastly, the Quantitative Separation Logic (QSL) [Batz et al., 2019] provides a probabilistic separation logic in which assertions denote real-valued quantitative predicates rather than Boolean ones. QSL enables reasoning about probabilistic pointer programs

by interpreting assertions as expectations, and supports quantitative reasoning principles such as expected resource usage. While QSL also extends classical separation logic toward probabilistic reasoning, its quantitative semantics and expectation-based assertions target different applications from ours: QSL focuses on reasoning about numerical properties of low-level pointer programs, whereas our work develops a qualitative, distributional, and relational reasoning framework aimed at verifying the security and independence properties of higher-level probabilistic algorithms.

2.4.2.4 Termination of Probabilistic Algorithms

Our program logic, introduced in Chapter 3, assumes that the target probabilistic algorithms always terminate with a fixed upper bound on the number of iterations (possibly dependent on the program input). This design choice is inherited from PSL [Barthe et al., 2019], where guaranteed termination ensures that every program execution yields a well-defined sub-distribution over outcomes, thereby simplifying both the semantic framework and the structure of verification rules. This assumption aligns well with our target class of algorithms, such as oblivious data structures and cryptographic primitives, which are typically designed with clear and bounded control flow.

In contrast, several other probabilistic program logics adopt a weaker assumption known as *almost-sure termination*—that is, the program terminates with probability 1, although individual non-terminating executions may exist with measure zero. This relaxation allows for the analysis of a broader class of programs, particularly those involving unbounded random processes, such as randomised retries, rejection sampling, or stochastic recursive procedures. For example, the verification works of pGCL by McIver and Morgan [2005] allow reasoning under almost-sure termination.

However, reasoning under almost-sure termination introduces substantial semantic and technical complexity. The soundness of such logics often requires reasoning about potentially infinite execution traces, and in some cases, constructing ranking supermartingales [Ferrer Fioriti and Hermanns, 2015, McIver et al., 2017] to argue termination.

2.4.3 Symbolic Methods and Computational Soundness

In Chapter 4, the framework we present can be seen as an example of a symbolic method with computational soundness proofs. In particular, it involves reasoning about ciphertexts symbolically (replacing them with the special symbol $\perp$), and the corresponding soundness proofs establish computational security results for this symbolic method (Section 4.3 and Section 4.4).

A common approach to verifying the security of applications—such as oblivious algorithms and network protocols that rely on encryption—is to first formally define their security properties and then prove that the specific algorithms satisfy these properties. However, this approach becomes significantly more challenging when the underlying encryption schemes provide strong computational security guarantees, such as those discussed in Section 2.3.3. These guarantees introduce additional complexity into the security properties of the target applications, making direct proofs difficult, tedious, and highly prone to errors [Delaune and Hirschi, 2017].

In contrast, some verification approaches—such as the Dolev-Yao model [Dolev and Yao, 2006]—adopt symbolic methods, which assume the correctness and indistinguishability of the encryption scheme. In these models, ciphertexts are treated as abstract symbols rather than concrete random values, significantly simplifying the reasoning process. This abstraction enables more automated and tractable verification, particularly for large-scale or complex systems.

However, symbolic methods do not account for the underlying computational assumptions and probabilistic behavior of cryptographic primitives. As a result, they may overlook subtle vulnerabilities that arise in concrete implementations. Therefore, while symbolic verification offers practical advantages in terms of automation and scalability, it is generally considered less reliable than computational approaches, which provide stronger and more realistic security guarantees by reasoning directly about the adversary's computational limitations.

To unify the advantages of both approaches, some works—following the framework proposed in [Abadi and Rogaway, 2007]—develop symbolic models and subsequently prove their *computational soundness*. This notion ensures that any security guarantee established within the symbolic model also holds under the standard computational model. By

doing so, these approaches retain the automation and tractability of symbolic reasoning while benefiting from the strong, concrete guarantees offered by computational security. Computational soundness results thus serve as a critical bridge between abstraction and cryptographic rigor, enabling verification frameworks that are both practical and trustworthy in real-world cryptographic settings.

Two comprehensive surveys [Backes et al., 2010, Delaune and Hirschi, 2017] provide thorough introductions to symbolic methods and the notion of computational soundness. In particular, the survey by Backes, Pfitzmann, and Waidner [Backes et al., 2010, Table 1] offers an in-depth examination of symbolic models in the context of cryptographic protocol analysis and their relationship to computational models.

For example, several key works have advanced the understanding of computational soundness under passive adversaries. Abadi and Rogaway's foundational result showed that symbolic pattern equivalence implies computational indistinguishability for symmetric encryption under strong assumptions such as the absence of key cycles [Abadi and Rogaway, 2002], while Herzog extended this to public-key encryption by introducing a graph-based key cycle detection method and relying on IND-CCA security [Herzog, 2005].

Baudet, Cortier, and Kremer generalized symbolic reasoning through static equivalence over equational theories, allowing broader application beyond specific cryptographic primitives [Baudet et al., 2009]. Building on this, Abadi, Baudet, and Warinschi demonstrated that symbolic secrecy remains computationally secure in the presence of offline guessing attacks [Abadi et al., 2006]. In secure information flow, Laud developed a programming language and static analysis technique that ensures computational security against polynomial-time adversaries [Laud, 2001], and Courant, Ene, and Lakhnech introduced a type system supporting deterministic encryption with provable non-interference under PRP assumptions [Courant et al., 2007]. Together, these works show that symbolic abstractions can yield rigorous computational guarantees with suitable assumptions.

Moreover, CPCL [Datta et al., 2005] is a Hoare-style logic extended to prove secrecy and key usability even under active adversaries [Datta et al., 2006]. It supports modular proofs for protocols using asymmetric encryption, Diffie-Hellman, and symmetric

primitives. In parallel, Gupta and Shmatikov [2005] proposed a logic for Diffie-Hellman-based key exchange protocols, showing that symbolic criteria can imply UC-style indistinguishability. While not directly related to our work, these studies illustrate how formal logic can serve as a bridge between symbolic verification and computational cryptographic models.

Chapter 3

Combining Classical and Probabilistic Reasoning

The formal verification of sophisticated probabilistic oblivious algorithms—such as Path ORAM [Stefanov et al., 2018], oblivious sampling [Sasy and Ohrimenko, 2019], and the Melbourne shuffle [Ohrimenko et al., 2014], which will be discussed in detail in Chapter 5—often requires capabilities beyond those of existing verification techniques (e.g. PSL [Barthe et al., 2019], ObliCheck [Son et al., 2021], λ_{OADT} [Ye and Delaware, 2022], and λ_{obliv} [Darais et al., 2019]). These gaps typically arise from one or more of the following requirements:

- Reasoning about probability distribution—specifically, uniform distributions—and probabilistic independence.
- Reasoning about dynamic random choices involving secrets and random variables—where selection occurs uniformly from a distribution over sets that may contain secret information.
- Reasoning about control-flow branches that depend on secret or random variables.
- Reasoning about random loops where the number of iterations is itself a random variable.
- Reasoning about negligible failure probabilities, which are intentionally designed so that the failure probability is bounded by some negligible factor (e.g. of the size of the secret data), meaning that they are secure in practice.

The pen-and-paper security proof [Stefanov et al., 2018, Ohrimenko et al., 2014] of the mentioned algorithms is often divided into two parts to address the last requirement:

1. Constructing an idealised, perfect version of the algorithm that never fails and proving that the probability of deviation between the practical and perfect versions is negligible. This will be briefly introduced in Section 3.7 but the detailed proof is out of scope of this present work. Relevant future works will be discussed in Chapter 6.

2. Demonstrating that the perfect version is secure, which involves addressing the remaining four requirements. The work in this chapter work focuses on this part.

In this chapter, we build a program logic that combines *classical* and *probabilistic* reasoning to address the aforementioned challenges, which we prove sound in Isabelle/HOL [Nipkow et al., 2002]. Our logic is situated atop the Probabilistic Separation Logic (PSL) [Barthe et al., 2019]; proving the soundness of our logic revealed several oversights in PSL, which we fixed (see Section 3.6).

The work in this chapter was published at Formal Methods 2024 [Yan et al., 2025].

3.1 Overview

This section provides an overview of the chapter, including a simple illustrative example. In the subsequent sections—Section 3.2, Section 3.3, and Section 3.4—we introduce the programming language semantics, assertion system, and inference rules, which together form the core components of our formal verification logic, analogous to those in Section 2.2.

We then present the soundness proof in Section 3.5, mechanized in Isabelle/HOL, and discuss several oversights identified in the original PSL paper [Barthe et al., 2019] in Section 3.6. Following this, Section 3.7 addresses the notion of negligible failure probability in probabilistic oblivious algorithms and discusses the corresponding proof. The chapter concludes with Section 3.8, which summarizes our contributions.

Let $\mathsf{eight}(i) = \{[x_0, x_1, \cdots, x_{i-1}] \mid \forall j.\ 0 \le x_j \le 7\}$
Let $\mathsf{pre} = \{\forall i \in \{0, 1, \cdots, n-1\}.\ S[i] \in \{0, 1\}\}$
Let $\mathsf{inv}(x) = \{\mathsf{Ct}(\mathsf{pre} \wedge i \le n) \wedge \mathbf{U}_{\mathsf{eight}(x)}[O]\}$
$\mathsf{synthetic}(S, O, n)$:
$\{\mathsf{Ct}(\mathsf{pre} \wedge O = [])\}$
1 $A[0] \leftarrow_\$ \mathbf{U}_{\{0,1,2,\cdots,7\}}$;
$\{\mathsf{Ct}(\mathsf{pre} \wedge O = []) \wedge \mathbf{U}_{\{0\cdots7\}}[A[0]]\}$
2 $A[1] \leftarrow_\$ \mathbf{U}_{\{0,1,2,\cdots,7\}}$; $i \leftarrow 0$;
$\{\mathsf{Ct}(\mathsf{pre} \wedge O = [] \wedge i = 0) \wedge \mathbf{U}_{\{0\cdots7\}}[A[0]] * \mathbf{U}_{\{0\cdots7\}}[A[1]]\}$
3 **while** $i < n$ **do** because $\mathsf{eight}(0) = \{[]\}$
$\{\mathsf{inv}(i) * \mathbf{U}_{\{0\cdots7\}}[A[S[i]]] * \mathbf{U}_{\{0\cdots7\}}[A[1 - S[i]]]\}$
4 $O \leftarrow O + A[S[i]]$; using proposition 1.8
$\{\mathsf{inv}(i+1) * \mathbf{U}_{\{0\cdots7\}}[A[1 - S[i]]]\}$
5 $m \leftarrow 8$; $j \leftarrow 0$;
$\{\mathsf{inv}(i+1) * \mathbf{U}_{\{0\cdots7\}}[A[1 - S[i]]] \wedge \mathsf{Ct}(m = 8 \wedge j = 0)\}$
6 **while** $A[S[i]] > j$ **do**
$\{\mathsf{Ct}(m > 7 \wedge m\ \%\ 8 = 0)\}$
7 $m \leftarrow m * 2$; $j \leftarrow j + 1$;
8 **if** $(j + S[i])\ \%\ 3 == 0$ **then**
9 $j \leftarrow j + 1$;
$\{\mathsf{Ct}(m > 7 \wedge m\ \%\ 8 = 0)\}$using Const rule around the loop
$\{\mathsf{inv}(i+1) * \mathbf{U}_{\{0\cdots7\}}[A[1 - S[i]]] \wedge \mathsf{Ct}(m > 7 \wedge m\ \%\ 8 = 0)\}$
10 $t \leftarrow_\$ \mathbf{U}_{\{1,2,3,\cdots,m\}}$; using RSample
$\{\mathsf{inv}(i+1) * \mathbf{U}_{\{0\cdots7\}}[A[1 - S[i]]] \wedge \mathbf{U}_{\{0\cdots7\}}[t\ \%\ 8]\}$
11 $A[S[i]] \leftarrow t\ \%\ 8$; using Rassign, Unif-Idp rule
$\{\mathsf{inv}(i+1) * \mathbf{U}_{\{0\cdots7\}}[A[1 - S[i]]] * \mathbf{U}_{\{0\cdots7\}}[A[S[i]]]\}$
12 $i \leftarrow i + 1$;
$\{\mathsf{inv}(n)\}$

FIGURE 3.1: Verification of the motivating algorithm

3.1.1 Challenges for verification

We have constructed the example algorithm in Fig. 3.1 to illustrate in a simplified form the kinds of complexities that will feature in the semantics and invariants needed to prove our case studies (Chapter 5). The teal-coloured parts show the verification and will be introduced in the next subsection. Our synthetic algorithm takes an input array S with size n containing secret elements: each either 0 or 1. The list O is the output and is empty initially but will be filled with some data later. We want to prove O will not leak any information about S. The synthetic algorithm first initialises array A with two random values sampled from the integers between 0 and 7. Its nested loop illustrates the following challenges:

1. The outer loop iterates n times where the ith iteration will append $A[S[i]]$ to O

(line 4). It simulates a simplified version of path ORAM [Stefanov et al., 2018], which maintains an invariant that virtual addresses are independent of each other and of the program's memory access patterns. The secret S can be seen as a sequence of secret virtual addresses and the output O represents the memory access pattern. We need to prove an invariant that the elements in O are independent of each other and independent of each element $A[S[i]]$ appended to O by the outer loop. Note: the assignment on line 4 breaks the independence between O and $A[S[i]]$, so Lines 4–11 update $A[S[i]]$ with a fresh random value to re-establish the independence for the next loop iteration. This ensures O is independent of S and will not leak secret information.

2. After initialising m with 8 on Line 5, we have the inner loop containing a *probabilistic and secret-dependent if-conditional*. Its secret dependence makes the control flow different over different values of the secret. The iteration count for the inner loop is *truly random*, depending on $A[S[i]]$ (where each iteration doubles m and increases j by 1 or 2 depending on whether $j + S[i]\%3 = 0$). These types of loops and conditionals are common in real-world oblivious algorithms (Chapter 5), yet they necessarily complicate reasoning.

3. On Line 10, the algorithm makes what we call a *dynamic random choice*, which is one over a truly random set (here, from 1 to the random variable m), assigning the chosen value to t. Then, (at Line 11) $A[S[i]]$ is assigned t % 8. This requires reasoning that t % 8 satisfies the uniform distribution on $\{0 \cdots 7\}$, because m is certainly a multiple of 8. Dynamic random choices are also common in the high level descriptions of real-world oblivious algorithms, as Chapter 5 demonstrates.

Lines $5 - 11$ are derived from the oblivious sampling algorithm [Sasy and Ohrimenko, 2019] to demonstrate challenges 2 and 3.

3.1.2 Mixing Probabilistic and Classical Reasoning

We show how to construct a program logic that combines classical and probabilistic (and independence) reasoning over different parts of the program so that it can verify our running example, as shown in Fig. 3.1. Namely, certain parts of the algorithm

(Lines 1, 2, 4, 10) require careful probabilistic reasoning, while others do not, but that each style of reasoning can benefit the other.

Our program logic is built upon the framework of Probabilistic Separation Logic (PSL) proposed by [Barthe et al., 2019], an existing program logic for reasoning about probabilistic programs. This choice is motivated by the fact that many probabilistic oblivious algorithms rely on probabilistic independence as a core intermediate condition to informally establish their obliviousness in pen-and-paper proofs [Stefanov et al., 2018, Ohrimenko et al., 2014, Shi, 2019, Sasy and Ohrimenko, 2019], as it provides an intuitive and simple approach. PSL naturally supports reasoning about probabilistic independence, making it a suitable and intuitive foundation for our work.

PSL employs the separating conjunction (here written $*$) familiar from separation logic [Reynolds, 2002] to capture when two probability distributions are independent. In situating our work atop PSL we extend its assertion forms with the new $\mathsf{Ct}(\cdot)$ assertion, to capture classical information. More importantly, however, we significantly extend the resulting logic with a range of novel reasoning principles for mixing classical and probabilistic reasoning embodied in a suite of new rules (Fig. 3.3), which we will present more fully in Section 3.4. These new rules show how classical reasoning (captured by $\mathsf{Ct}(\cdot)$ assertions) can be effectively harnessed, and allow reasoning about dynamic random choices, secret-dependent if-statements, and random loops, making our logic significantly more applicable than PSL; while leveraging PSL's support for intuitive reasoning about probability distributions makes our logic also more expressive than prior probabilistic program logics without probabilistic independence [den Hartog, 1999, Rand and Zdancewic, 2015]. We also harness the close interaction between classical and probabilistic reasoning to allow new ways to prove security (e.g. the Unif-Idp rule and the final proposition of Theorem 3.2, which will be introduced in Fig. 3.3 and Section 3.3.1), and new ways to reason about random sampling (embodied in the RSample rule, Fig. 3.3). Each represents a non-trivial insight, and all are necessary for reasoning about real-world oblivious algorithms (Chapter 5). The increase in expressiveness, beyond prior probabilistic program logics [Barthe et al., 2019, den Hartog, 1999, Rand and Zdancewic, 2015], within a principled and clean extension of PSL attests to the careful design of our logic.

The combination of classical and probabilistic reasoning means that our logic tracks two kinds of *atomic assertions*, as follows.

Certain Assertions. Classical reasoning is supported by certain assertions $\mathsf{Ct}(e_r)$ that state that some property e_r (which may mention random variables) is true with absolute certainty, i.e. is true in all memories supported by the current probabilistic state of the program. With certain assertions and classical reasoning, our logic can reason about **loops with random iteration numbers and randomly secret-dependent if statements**. Doing so requires distinguishing classical from distribution (such as independence) assertions, because the latter are ill-suited for reasoning about random loops and conditionals.

For example, from Line 5 to 9, although the random loop and the probabilistic- and secret-dependent if statement complicate the algorithm, we only need classical reasoning to conclude that after the loop m is certainly a multiple of 8 (using the RLoop and RCond rules in Fig. 3.3, which have the classic form). This information is sufficient to verify the remainder of the algorithm.

Distribution Assertions. On the other hand, reasoning about probability distributions is supported by distribution assertions, which we adopt and extend from PSL: for a set expression e_d (which is allowed to mention non-random program variables), $\mathbf{U}_{e_d}[e_r]$ states that expression e_r is uniformly distributed over the set denoted by e_d in the sense that when e_r is evaluated in the current probabilistic state of the program it yields a uniform distribution over the evaluation of e_d. We define these concepts formally later in Section 3.3.1 (see Theorem 3.1). With this reasoning style, we support **dynamic random choice** (e.g. Line 10, the value is sampled from a truly probabilistic set), which is not supported by previous works [Barthe et al., 2019, Son et al., 2021, Ye and Delaware, 2022, Darais et al., 2019, Barthe et al., 2014a, den Hartog, 1999, Rand and Zdancewic, 2015]. Note that we require e_d to be deterministic here because if e_d can be probabilistic, then it means a probabilistic expression satisfies a *uniform distribution on a probabilistic set*—a clear contradiction. (One could attempt to define such a notion via conditional distributions, but the resulting distribution would no longer be uniform over any set.)

For example, at Line 10, even if we do not specify the detailed distribution of m, we can conclude $t \,\%\, 8$ satisfies the uniform distribution on the set $\{0 \cdots 7\}$, as m is certainly a

multiple of 8, by an argument based on our concept of an *even partition* (Theorem 3.4). This reasoning is supported by our novel RSAMPLE rule (Fig. 3.3). Here, it requires that all the possible sets (in this case, $\{1 \cdots 8\}$ or $\{1 \cdots 16\}$ or ...) over which t was sampled, can each be evenly mapped to (and thus partitioned by) the target set (here $\{0 \cdots 7\}$) by the applied function (here %8). Thus t % 8 must satisfy the uniform distribution on $\{0 \cdots 7\}$.

Unifying classical and probabilistic independence reasoning Another important feature of our logic is that it allows independence to be *derived* by leveraging classical reasoning. For example, considering Line 10, 11, if a variable ($A[S[i]]$) always satisfies the same distribution (uniform distribution on $\{0 \cdots 7\}$) over any possible values of some other variables (O and $A[1 - S[i]]$), then the former is independent of the latter (because O and $A[1 - S[i]]$ will not influence the values of $A[S[i]]$). The new rule UNIF-IDP (Fig. 3.3) embodies this reasoning (where $*$ denotes independence and $\mathbf{D}()$ stands for an arbitrary distribution).[1]

Our logic also includes a set of useful propositions (Theorem 3.2) that aid deriving independence information from classical reasoning.

Returning to the example, with the conclusion that $A[S[i]]$ is independent of other variables, we can construct the loop invariant of the outer loop ($\mathsf{inv}(i)$) stating that the output array O always satisfies a uniform distribution following the ith iteration, which is captured by $\mathsf{eight}(i)$. We use the final proposition of Theorem 3.2 here. Intuitively, this proposition says given a reversible function (whose inputs can be decided by looking at its outputs, e.g. array appending), if its two inputs satisfy uniform distribution and are independent of each other, then the result of the function should satisfy the uniform distribution on the product (by the function) of the two inputs' distribution.

By the invariant, we can conclude finally the output array always satisfies the uniform distribution on $\mathsf{eight}(n)$, regardless of secret S, which means the output will not leak any secret information.

Next, we will introduce the details of our logic: its programming language and semantics in Section 3.2, assertions in Section 3.3.1, and rules in Section 3.4.

[1] In this case we cannot use PSL's frame rule because m is not independent of A.

3.2 Language and Semantics Modifications

We inherit all definitions from Section 2.2.1 and most from Section 2.2.2, except for the loop definition in Eq. (2.17) and the random choice definition in Eq. (2.13). The revised definitions are presented below, and the new semantics are presented in Fig. 3.2.

$$
\begin{aligned}
\mathbf{RC} \ni c ::= {} & \mathbf{skip} \mid \mathbf{RV} \leftarrow \mathbf{RE} \\
& \mid \mathbf{RV} \leftarrow_{\$} \mathbf{U_{RE}} \mid \mathbf{RC}; \mathbf{RC} \\
& \mid \mathbf{if}_D\ \mathbf{DE}\ \mathbf{then\ RC\ else\ RC} \\
& \mid \mathbf{if}_R\ \mathbf{RE}\ \mathbf{then\ RC\ else\ RC} \\
& \mid \mathbf{while}_D\ \mathbf{DE}\ \mathbf{do\ RC} \\
& \mid \mathbf{while}_R\ \mathbf{RE}\ \mathbf{do\ RC}
\end{aligned}
\qquad
\begin{aligned}
\mathbf{C} \ni c ::= {} & \mathbf{skip} \mid \mathbf{DV} \leftarrow \mathbf{DE} \\
& \mid \mathbf{RV} \leftarrow \mathbf{RE} \mid \mathbf{RV} \leftarrow_{\$} \mathbf{U_{RE}} \mid \mathbf{C}; \mathbf{C} \\
& \mid \mathbf{if}_D\ \mathbf{DE}\ \mathbf{then\ C\ else\ C} \\
& \mid \mathbf{if}_R\ \mathbf{RE}\ \mathbf{then\ RC\ else\ RC} \\
& \mid \mathbf{while}_D\ \mathbf{DE}\ \mathbf{do\ C} \\
& \mid \mathbf{while}_R\ \mathbf{RE}\ \mathbf{do\ RC}
\end{aligned}
$$

$$
[\![x_r \leftarrow_{\$} \mathbf{U}_{e_r}]\!](\sigma, \mu) = (\sigma, \mathsf{bind}(\mu, m \mapsto \mathsf{bind}(\mathsf{Unif}_{\mathsf{vset}([\![e_r]\!](\sigma,m))}, u \mapsto \mathsf{unit}(m[x_r \mapsto u]))))
\tag{3.1}
$$

$$
[\![\mathbf{while}_D\ b\ \mathbf{do}\ c]\!](\sigma, \mu) = [\![\mathbf{if}_D\ b\ \mathbf{then}\ (c;\ \mathbf{while}_D\ b\ \mathbf{do}\ c)]\!](\sigma, \mu)
\tag{3.2}
$$

$$
[\![\mathbf{while}_R\ b\ \mathbf{do}\ c]\!](\sigma, \mu) = [\![\mathbf{if}_R\ b\ \mathbf{then}\ (c;\ \mathbf{while}_R\ b\ \mathbf{do}\ c)]\!](\sigma, \mu)
\tag{3.3}
$$

FIGURE 3.2: Programming Language Semantics

To address the requirement of reasoning about random loops, we introduce random loops ($\mathbf{while}_R\ \cdots\ \mathbf{do}\ \cdots$) as a new command, where the loop condition can depend on random expressions rather than only on deterministic expressions, as in PSL. These enhancements increase the expressivity of the language, making it suitable for capturing the practical oblivious algorithms we target in Chapter 5.

As in PSL, we assume that programs always terminate. However, unlike PSL, which defines loop semantics informally by unfolding a loop into a sequence of loop body executions, our approach recursively unfolds the loop into a series of `if` statements by Eq. (3.2) and Eq. (3.3). This formulation enables direct mechanisation in Isabelle/HOL.

Additionally, to handle dynamic random choices, we modify the random choice command so that the sampled set is represented as a random expression rather than a constant value, as defined in Eq. (3.1). The function $\mathsf{vset}()$ is used to retrieve this denotation after evaluating e_r. This is required in our Isabelle/HOL formalisation: to state that an expression follows a uniform distribution over some set, the semantics must provide an

actual set at the type level. The $\mathsf{vset}()$ function converts the evaluated value into such a set, ensuring the definition type-checks. In thesis text, this function has no semantic significance and can be safely ignored.

3.3 Assertions System

3.3.1 Definitions and Semantics

Following the approach of combining classical and probabilistic reasoning, we inherit the structural assertions from PSL (Section 2.2.3, bottom half of Fig. 2.5) but redefine its atomic assertions (top half of Fig. 2.5). Specifically, we introduce the certainty assertion $\mathsf{Ct}(e_r)$ and extend the uniform distribution assertion $\mathbf{U}_{e_d}[e_r]$ by allowing the set to be specified by an deterministic expression e_d, rather than a constant as in PSL.

Definition 3.1 (Atomic assertion and semantics)**.**
Atomic assertions (**AP**) are defined as: $\mathbf{AP} \ni p ::= \mathsf{Ct}(\mathbf{RE}) \mid \mathbf{U}_{\mathbf{DE}}[\mathbf{RE}]$
Their semantics are:

$$\begin{aligned}
[\![\mathsf{Ct}(e_r)]\!] &= \{(\sigma,\mu) \mid \forall m \in \mathsf{supp}(\mu).\ \ [\![e_r]\!](\sigma,m) = \mathsf{true}\} \\
[\![\mathbf{U}_{e_d}[e_r]]\!] &= \{(\sigma,\mu) \mid \mathsf{FV}(e_r) \cup \mathsf{FV}(e_d) \subseteq \mathsf{dom}(\sigma) \cup \mathsf{dom}(\mu) \\
&\qquad\qquad \text{and } \mathsf{Unif}_{\mathsf{vset}([\![e_d]\!]\sigma)} = [\![e_r]\!](\sigma,\mu)\}
\end{aligned}$$

The free variables of an expression e are denoted $\mathsf{FV}(e)$. The domain of distribution μ over memories, written $\mathsf{dom}(\mu)$, is the set of random variables in the memories in the support of μ.

For a random variable expression e_r, $\mathsf{Ct}(e_r)$ asserts that e_r evaluates to true in every memory consistent with the current configuration, meaning it holds with absolute certainty. Notably, the set of random variable expressions e_r can encompass all standard assertions from classical Hoare logic, thereby enabling classical reasoning.

The assertion $\mathbf{U}_{e_d}[e_r]$ asserts that the evaluation of random variable expression e_r yields the uniform distribution over the set denoted by the deterministic expression e_d when evaluated in the current deterministic memory. The $\mathsf{vset}()$ function is used to retrieve that denotation after evaluating e_d. We require the expression e_d to be deterministic as

otherwise this assertion can introduce contradictions (e.g. if the set expression instead denoted a truly random set including possible sets $\{1, 2\}$ and $\{0\}$, then e_r will not be uniformly distributed on any set).

From PSL our logic inherits its other assertions and Kripke resource monoid semantics, which was introduced in Section 2.2.3.

We will write $\mathbf{D}(x)$ to abbreviate $\mathsf{Ct}(x = x)$, which asserts that the variable x is in the domain of the partial configuration. Any distribution of x satisfies this assertion.

3.3.2 Assertion Implication

Assertion implication is essential for program reasoning, as it is frequently used in the Weak rule (Fig. 2.6), also known as the Consequence rule in traditional Hoare logic (Fig. 2.3). In Hoare logic, assertion implication follows standard mathematical principles. However, in PSL and our logic, we must develop custom assertion implication rules, as direct application can be error-prone.

For example, based on our assertion definitions, $\mathsf{Ct}(P) \wedge \mathsf{Ct}(Q)$ is equivalent to $\mathsf{Ct}(P \wedge Q)$. However, $\mathsf{Ct}(a = 1) \vee \mathsf{Ct}(a = 2)$ is not equivalent to $\mathsf{Ct}(a = 1 \vee a = 2)$. The former asserts that either a is always 1 or a is always 2 (stronger); the latter asserts that always a is either 1 or 2 in each support (weaker).

As another example, the implication $(P * Q) \wedge (Q * R) \wedge (R * P) \implies P * Q * R$ always holds in standard separation logic [Reynolds, 2002], but it may be false in PSL, our logic, or other Bunched Implications [O'Hearn and Pym, 1999]. For instance, suppose $P = \mathbf{D}(a)$, $Q = \mathbf{D}(b)$, and $R = \mathbf{D}(c)$. This expression asserts that pairwise independence implies mutual independence among all three. However, consider the case where a and b are independently and uniformly sampled from $\{0, 1\}$, and $c = a \ \mathsf{xor}\ b$. Here, a and b are pairwise independent, as are b and c, and c and a, but c is fully determined by a and b, meaning they are not mutually independent.

Here, we introduce several propositions about assertions implication, whose proof are also formalised in Isabelle/HOL. They are very useful in the verification and reflect the interplay between classical and probabilistic independence reasoning, especially the last one.

Proposition 3.2.

$$\models (\phi * \psi) \wedge \eta \to (\phi \wedge \eta) * \psi \ \textit{,where} \ \models \phi \to \mathbf{D}(\mathsf{FV}(\eta) \cap \mathbf{RV})$$

$$\models (\phi * \psi) \to (\phi \wedge \psi)$$

$$\models \mathbf{U}_S[e] \wedge \mathsf{Ct}(f \textit{ is a bijection from } S \textit{ to } S') \to \mathbf{U}_{S'}[f(e)]$$

$$\models (\mathsf{Ct}(\phi \wedge \psi)) \to (\mathsf{Ct}(\phi) \wedge \mathsf{Ct}(\psi))$$

$$\models (\mathsf{Ct}(\phi) \wedge \mathsf{Ct}(\psi)) \to (\mathsf{Ct}(\phi \wedge \psi))$$

$$\models \mathbf{U}_S[e] \to \mathsf{Ct}(e \in S)$$

$$\models \mathbf{U}_S[e] \wedge \mathsf{Ct}(e = e') \to \mathbf{U}_S[e']$$

$$\models \mathsf{Ct}(x = e \wedge x \notin \mathsf{FV}(e')) \wedge \mathbf{D}(e) * \mathbf{D}(e') \implies \mathbf{D}(x) * \mathbf{D}(e')$$

$$\models \begin{array}{l} \mathsf{Ct}(\forall a, b \in S, c, d \in S'. f(a,c) = f(b,d) \to a = b \wedge c = d) \wedge \mathbf{U}_S[x] * \mathbf{U}_{S'}[e] \\ \to \mathbf{U}_{S \times_f S'}[f(x,e)], \ \textit{where } S \times_f S' = \{f(a,b) \mid a \in S \wedge b \in S'\} \end{array}$$

The first two are inherited from PSL. The third one generalises a similar proposition of PSL [Barthe et al., 2019] over possibly different sets S and S'. The fourth and fifth show the equivalence of $\wedge$ whether inside or outside the certain assertions. The sixth shows the straightforward consequence that if e is uniformly distributed over set S, then the value of e must be in S. The seventh shows two expressions satisfy the same distribution if they are certainly equal. The eighth shows if we know that e is independent of e' and we know another variable $x = e$ additionally, we can conclude that x is also independent of e' if x is not a free variable in e'.

The last one also generalises a proposition of PSL [Barthe et al., 2019] by leveraging $\mathsf{Ct}(\cdot)$ conditions: it restricts binary function f by requiring it to produce different outputs when given two different pairs of inputs. In practice, we will use this lemma letting f be the concatenation function on two arrays where S is a set of possible arrays with the same length. We conclude the concatenated array satisfies the uniform distribution on S times S' if those premises hold. We will also discuss more details of it in Section 3.5.4.

3.4 Inference Rules

The triple $\vdash \{\phi\}\ c\ \{\psi\}$ in our program logic represents a standard Hoare logic correctness statement, where c is a program command, and ϕ and ψ are the precondition and postcondition, respectively. The definition of triple validity differs slightly from that in PSL; while PSL incorporates program termination within its semantics, we assume termination within the validity of the triple. Our approach aligns more closely with conventional formal verification principles (such as Hoare Logic).

Definition 3.3 (Triple Validity)**.** Given two assertions ϕ, ψ and a program command c, the triple $\{\phi\}\ c\ \{\psi\}$ is valid if for all configuration (σ, μ) satisfying $(\sigma, \mu) \models \phi$ and $[\![c]\!](\sigma, \mu)$ exists (terminate), we have $[\![c]\!](\sigma, \mu) \models \psi$, denoted $\vdash \{\phi\}\ c\ \{\psi\}$.

$$\textsc{RAssign}\ \frac{\phi \in \mathbf{AP}}{\vdash \{\phi[e_r/x_r]\}\ x_r \leftarrow e_r\ \{\phi\}} \qquad \textsc{RSample}\ \vdash \{\mathsf{Ct}(\mathsf{EP}(f, S, S'))\}\ x_r \leftarrow_{\$} \mathbf{U}_S\ \{\mathbf{U}_{S'}[f(x_r)]\}$$

$$\textsc{RCond}\ \frac{\vdash \{\mathsf{Ct}(\phi \wedge b \neq \mathsf{false})\}\ c\ \{\mathsf{Ct}(\psi)\} \qquad \vdash \{\mathsf{Ct}(\phi \wedge b = \mathsf{false})\}\ c'\ \{\mathsf{Ct}(\psi)\}}{\vdash \{\mathsf{Ct}(\phi)\}\ \mathbf{if}_R\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \{\mathsf{Ct}(\psi)\}}$$

$$\textsc{RLoop}\ \frac{\vdash \{\phi\}\ \mathbf{if}_R\ b\ \mathbf{then}\ c\ \{\phi\}}{\vdash \{\phi\}\ \mathbf{while}_R\ b\ \mathbf{do}\ c\ \{\phi \wedge \mathsf{Ct}(b = \mathsf{false})\}}$$

$$\textsc{Unif-Idp}\ \frac{\mathsf{FV}(a) \cap \mathsf{MV}(c) = \emptyset \qquad b \notin \mathsf{FV}(a) \qquad \vdash \{\mathsf{Ct}(a \in A) * Q \wedge \mathsf{Ct}(P)\}\ c\ \{\mathbf{U}_S[b]\}}{\vdash \{\mathsf{Ct}(a \in A) * Q \wedge \mathsf{Ct}(P)\}\ c\ \{(\mathbf{D}(a) * \mathbf{U}_S[b])\}}$$

FIGURE 3.3: Rules capturing the interplay of classical and probabilistic reasoning

Our logic inherits all of PSL's original rules (Fig. 2.6) except for random assign and sampling rules, but we represent equality tests using $\mathsf{Ct}(x = y)$ instead of PSL's $x \sim y$ or $x = y$. While the meanings are equivalent, we translate PSL's $\sim$ and $=$ assertions into our Ct-style assertions.

Fig. 3.3 depicts the rules of our logic that embody its new reasoning principles, and support the requirements listed at Section 3.1.1.

Random Assignment Rule. RASSIGN has the classical Hoare logic form (Fig. 2.3). It requires the postcondition ϕ is atomic to avoid unsound derivations, for example, the

$$\mathsf{RV}(x_r \leftarrow e_r) = \mathsf{FV}(e_r), \quad \mathsf{RV}(x_r \leftarrow_{\$} \mathbf{U}_{e_r}) = \mathsf{FV}(e_r), \quad \mathsf{RV}(\mathbf{while}_D\ b\ \mathbf{do}\ c) = \mathsf{RV}(c)$$

$$\mathsf{RV}(c; c') = \mathsf{RV}(c) \cup (\mathsf{RV}(c') - \mathsf{WV}(c)), \quad \mathsf{RV}(\mathbf{if}_D\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{RV}(c) \cup \mathsf{RV}(c')$$

$$\mathsf{RV}(\mathbf{if}_R\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{RV}(c) \cup \mathsf{RV}(c') \cup \mathsf{FV}(b), \ \mathsf{RV}(\mathbf{while}_R\ b\ \mathbf{do}\ c) = \mathsf{FV}(b) \cup \mathsf{RV}(c)$$

$$\mathsf{WV}(x_r \leftarrow e_r) = \{x_r\} - \mathsf{FV}(e_r), \quad \mathsf{WV}(x_r \leftarrow_{\$} \mathbf{U}_{e_r}) = \{x_r\}, \quad \mathsf{WV}(\mathbf{while}_D\ b\ \mathbf{do}\ c) = \emptyset$$

$$\mathsf{WV}(c; c') = \mathsf{WV}(c) \cup (\mathsf{WV}(c') - \mathsf{RV}(c)), \quad \mathsf{WV}(\mathbf{if}_D\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{WV}(c) \cap \mathsf{WV}(c')$$

$$\mathsf{WV}(\mathbf{if}_R\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = (\mathsf{WV}(c) \cap \mathsf{WV}(c')) - \mathsf{FV}(b)$$

$$\mathsf{WV}(\mathbf{while}_R\ b\ \mathbf{do}\ c) = \mathsf{WV}(c) - \mathsf{FV}(b)$$

$$\mathsf{MV}(x_r \leftarrow e_r) = \{x_r\}, \quad \mathsf{MV}(x_r \leftarrow_{\$} \mathbf{U}_{e_r}) = \{x_r\}, \quad \mathsf{MV}(\mathbf{while}_D\ b\ \mathbf{do}\ c) = \mathsf{MV}(c)$$

$$\mathsf{MV}(c; c') = \mathsf{MV}(c) \cup \mathsf{MV}(c'), \quad \mathsf{MV}(\mathbf{if}_D\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{MV}(c) \cup \mathsf{MV}(c')$$

$$\mathsf{MV}(\mathbf{if}_R\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{MV}(c) \cup \mathsf{MV}(c'), \quad \mathsf{MV}(\mathbf{while}_R\ b\ \mathbf{do}\ c) = \mathsf{MV}(c)$$

FIGURE 3.4: Auxiliary Functions

invalid triple $\{0 = 0 * 0 = 0\}\ x = 0\ \{x = x * x = x\}$. Nonetheless, non-atomic assertions, such as those expressing probabilistic independence, can still be introduced through the Constant rule, Frame rule, and assertion implication propositions.

Random Sample Rule. As mentioned in Section 3.1.2, the RSAMPLE rule is another embodiment of the general principle underlying the design of our logic, of classical and probabilistic reasoning enhancing each other. Specifically, it allows us to deduce when a randomly sampled quantity $f(x_r)$ (a function f applied to a random variable x_r) is uniformly distributed over set S' when the random variable x_r was uniformly sampled over set S. It is especially useful when S is itself random. It relies on the function f *evenly partitioning* the input set S into S', as defined below.

Definition 3.4 (Even Partition)**.** Given two sets S, S' and a function f, we say that f *evenly partitions* S into S' if and only if $S' = \{f(s) \mid s \in S\}$ and there exists an integer k such that $\forall s' \in S'. |\{s \in S \mid f(s) = s'\}| = k$. In this case we write $\mathsf{EP}(f, S, S')$.

RSAMPLE allows reasoning over random choices beyond original PSL [Barthe et al., 2019], and in particular dynamic random sampling from truly random sets. For example,

at Line 10 of Fig. 3.1, we have $\mathsf{Ct}(\mathsf{EP}(f, S, S'))$ where $f = \ \%\ 8$, $S = \{0 \cdots m\}$, $S' = \{0 \cdots 7\}$. Letting $k = m/8$ with the above definition, we can prove the pre-condition implies $\mathsf{Ct}(\mathsf{EP}(f, S, S'))$. Note that if $m = 9$ then $\mathsf{Ct}(\mathsf{EP}(f, S, S'))$ will not hold because we cannot find k. The existence of k makes sure that S can be *evenly* partitioned to S' by f. Also, from our new random sample rule RSAMPLE, one can obtain PSL's original rule by letting $S' = S$ and $f = (\lambda x.\ x)$.

Random If and While Rule. In addition to PSL's random conditional rule, we also include a new RCOND rule for random conditions that operate over certainty assertions $\mathsf{Ct}(\cdot)$. It is in many cases more applicable because it does not require the branching condition to be independent of the precondition and, while it reasons only over certainty assertions, other conditions can be added by applying the CONST rule [Barthe et al., 2019]. The new random loop rule RLOOP is straightforward, requiring proof of the invariant ϕ over a random conditional.

Note that the true case of if-condition is written as $b \neq \mathsf{false}$. This is because, in our formalization, values are defined abstractly, and we adopt a C-like convention for their semantics: any value that is not false is interpreted as true.

Uniform-Independence Rule. The final new rule UNIF-IDP unifies two methods to prove the independence of an algorithm's output b from its input a: it says that if given any arbitrary distribution of a we can always prove that the result b is uniformly distributed, then a and b are independent because the distribution of a does not influence b, where $\mathsf{MV}(c)$ is the variables c may write to (same as PSL's definition). It is useful for programs that consume their secrets by random choice at runtime (e.g. Fig. 3.1 we verified in Section 3.1.2 and the Oblivious Sampling algorithm [Sasy and Ohrimenko, 2019] we verify in Chapter 5).

As an example, we used this rule between Line 10 and Line 11 in Fig. 3.1 by letting $a = (O,\ A[1 - S[i]])$ and P, Q be the other information in the assertion before Line 10. The first premise of the rule is true because these two lines of code never modify O and $A[1 - S[i]]$. The second premise is also trivially true. The third premise is proved by the RSAMPLE and RASSIGN rules. This yields the conclusion that O and $A[1 - S[i]]$ are independent of $A[S[i]]$.

Note that the pre-condition $\mathsf{Ct}(a \in A) * Q \wedge \mathsf{Ct}(P)$ appears in both premise and conclusion of the rule. Considering the WEAK rule [Barthe et al., 2019] (aka the classical consequence rule), when the precondition is in the premise, we want it be as strong as it can so that the premise is easier to be proved. When it is in the conclusion, we want it be as weak as it can so that the conclusion is more useful. These two requirements guide us to design the rule with two free assertions connected by $\wedge$ and $*$ respectively so that it is very flexible. If we change the pre-condition to $\mathbf{D}(a)$ (deleting A, P, Q), this rule is still sound (which can be proved by letting A be the universe set and P, Q be true) but much less applicable. This rule may appear similar to the Frame Rule, but it is fundamentally different: it has no frame component—both a and b are relevant to the executed program.

Since we have extended the programming language with sampling and random loop commands, it is necessary to adapt the definitions of auxiliary functions accordingly, as shown in Fig. 3.4. These definitions adhere to the same principles as those in PSL, while also addressing several oversights—specifically, missing definitions—in the original PSL paper [Barthe et al., 2019] (e.g. $\mathsf{WV}(\mathbf{if}_D\ b\ \mathbf{then}\ c\ \mathbf{else}\ c')$ and $\mathsf{MV}(\mathbf{if}_D\ b\ \mathbf{then}\ c\ \mathbf{else}\ c')$).

3.5 Soundness

Theorem 3.5. *All the rules in Fig. 3.3 and propositions 3.2 , plus the other original PSL rules Fig. 2.6 except for RAssn and RSamp rules, are sound, i.e. are valid according to Definition 3.3.*

We formalised our logic and proved it sound in Isabelle/HOL, where the code are available on Zenodo [Yan, 2024]. It constitutes 7K lines of Isabelle and required approx. 8 person-months to complete. While some Isabelle proofs closely follow the original pen-and-paper proofs from PSL, we identified several issues in PSL's definitions and proofs. In this section, we present our formal proof and in the next section we briefly discuss the errors found in PSL, highlighting the importance and benefits of machine-checked proofs in verifying the soundness of program logics.

Section 3.5.1 introduces the file `Distribution.thy`. Subsequently, Section 3.5.2 covers the files `Semantics.thy` and `SemLemma.thy`. Following this, Section 3.5.3 presents the

file `Assertion.thy` and `Rule1.thy`. Finally, Section 3.5.4 describes `Rule2.thy` which contains all main results (inference rules and assertion implications) of this work.

3.5.1 Probabilistic Distribution Formalization

We formalise probabilistic distributions as the type $'a \Rightarrow$ `real` following Section 2.2.1, where $'a$ is a type variable. Then we require the well-formedness conditions for probabilistic distributions (e.g. the total probability must equal 1). However, in Isabelle/HOL, the sum of a function μ over a set A, denoted as $\Sigma_{a \in A}\ \mu(a)$ and written as `sum` μ `A` in Isabelle, is well-defined only when the set A is finite; otherwise, the sum defaults to zero, even if μ assigns probability 1 to a particular element. Since Section 2.2.1 specifies that A may be infinite (specifically countably infinite), we cannot directly apply Isabelle's built-in sum function for our formalisation.

Thus, we introduce an alternative summation operator, `psum`, defined in terms of Isabelle's built-in `sum` function, to accurately represent probabilistic sums over potentially infinite sets:

Definition 3.6 (Probabilistic Sum)**.** `psum` $\mu\ A =$ `sum` $\mu\ (A \cap \mathsf{supp}(\mu))$

By intersecting the supports, the resulting set becomes finite (same as PSL), ensuring that the sum in the definition is well-defined. With this operator, we define the concept of a valid distribution as follows:

Definition 3.7 (Valid Distribution)**.**
`validDist` $\mu = ($`psum` μ `UNIV` $= 1 \wedge (\forall a.\ 0 \leq \mu(a) \leq 1))$

From these definitions, we can deduce that any valid distribution must have finite support. Indeed, an infinite support would result in a probabilistic sum of 0, contradicting the requirement that the total probability sum must equal 1.

Subsequently, we formalize the remaining operations introduced in Section 2.2.1 (e.g. conditioning, probabilistic times, and bind) consistently using the probabilistic sum (`psum`) defined above.

Finally, we have proved several useful lemmas concerning the definitions introduced above. Many of these lemmas reflect algebraic properties such as associativity or commutativity of addition and multiplication, adapted specifically for the probabilistic sum

context. Below, we highlight one representative lemma, notable for being frequently used and non-trivial:

Lemma 3.8 (Image of Probabilistic Sum)**.** *For a valid distribution h, we have* $\mathit{psum}\ h\ S = \mathit{psum}\ (\lambda y.\ \mathit{psum}\ h\ \{x \in S.\ g\ x = y\})\ (\mathit{image}\ g\ (S \cap \mathsf{supp}(h)))$

This lemma indicates that a probabilistic sum over a set can be equivalently expressed as nested probabilistic sums. Intuitively, this resembles summing elements of a matrix with n rows and m columns, where the sum of all elements in the matrix equals the sum over each row's individual sum. This lemma is particularly useful when proving properties related to probabilistic independence.

3.5.2 Semantics Formalization

Following Section 2.2.2 and Section 3.2, we formally define the syntax and semantics of the programming language in the Isabelle theory file `Semantic.thy`.

We start by declaring types **DV**, **RV**, and **Val**, ensuring that the type **Val** contains at least two distinct values, denoted true and false, to facilitate the proper execution of conditional statements and loops. Subsequently, deterministic memory, random memory, and configurations are defined according to the specifications provided in Section 2.2.2.

Let $\mathsf{vset}()$ be a function of type $\mathbf{Val} \to \mathcal{P}(\mathbf{Val})$, taking one value and returning a non-empty, finite set of values, for giving semantics to dynamic random choice.

Let op denote a set of operations on values, particularly including binary functions of type $(\mathbf{Val} \times \mathbf{Val}) \to \mathbf{Val}$. In practice, we assume op encompasses standard arithmetic operators, as well as list, set, and other operations as needed. Using op, we then formally redefine expressions within Isabelle, while the evaluation semantics for expressions remain standard and consistent with previous definitions:

Definition 3.9 (Expressions)**.** Expressions are either deterministic or random, defined as follows:

$$\text{Deterministic expressions: } \mathbf{DE} \ni e_d ::= \mathbf{Val} \mid \mathbf{DV} \mid \mathsf{op}\ \mathbf{DE}\ \mathbf{DE}$$
$$\text{Random expressions: } \mathbf{RE} \ni e_r ::= \mathbf{Val} \mid \mathbf{DV} \mid \mathbf{RV} \mid \mathsf{op}\ \mathbf{RE}\ \mathbf{RE}$$

We also define several auxiliary functions essential for semantic reasoning. These include `isDE`, which determines whether an expression is deterministic; `freeDV` and `freeRV`,

which return the sets of deterministic and random free variables within an expression, respectively; and `Eval`, which handles expression evaluation over memories. Notably, `REval` represents the random evaluation function, which yields a distribution of values for a given expression under specified memories distribution.

Using the above definitions and in accordance with Section 3.2, we define the program commands (`cmd`) and introduce the function `Rcmd` to differentiate between commands in **C** and **RC**. We also define the function `WFcmd`, which ensures that deterministic assignments depend exclusively on deterministic expressions. Additionally, other auxiliary functions outlined in Fig. 3.4 are formally defined based on the structure of `cmd`.

Finally, we inductively define the semantics of our programming language via the function `sem`, which captures program execution. Compared to the semantic definitions presented in Fig. 2.4 and Fig. 3.2, our formalization introduces an additional natural number n and wraps configurations with the option type. The number n decreases by one each time a loop is unfolded (i.e. each iteration). When n reaches zero or the initial configuration is `None`, the execution always yields `None`.

Introducing the natural number n greatly facilitates inductive proofs concerning the semantics (`sem`) in various related lemmas. Typically, we first perform induction on n, followed by induction on the executed command. These two inductions are completed via a primary lemma and an auxiliary lemma, respectively. The induction on n provides an essential inductive hypothesis, enabling us to handle loop cases by unfolding loops into conditional statements with a strictly smaller n. Numerous illustrative examples of this inductive approach are demonstrated in our proof.

In `SemLemma.thy`, we formally establish several essential lemmas regarding program semantics, each proven using the inductive strategy previously outlined:

- Lemma `sem_validDist` states that given a valid initial state, executing any command that yields a final state will produce a valid final state.
- Lemma `rcmd_invD` establishes that executing any random command (**RC**) does not modify deterministic memory.

- Lemma `sem_SucN` demonstrates monotonicity with respect to the execution bound n: if an execution produces a particular final state for some n, then executions with a greater n yield the same final state.

- Lemma `whileR_seq` ensures that if the execution of a loop yields a final state, there must exist a sequence of (non-recursive) If-statements leading to the same final result. This lemma is instrumental in proving the rule for random loops.

- Lemma `sem_det` states that if executing a command c transforms deterministic memory from d to d' under an initial random memory r, the same transformation holds when replacing r with another random memory r'. Thus, deterministic memory transformations are unaffected by initial random memory differences.

- Lemma `sem_supp` establishes that if the support of one initial random memory r is a subset of another random memory r', this subset relationship persists after executing any command on both memories.

- Lemma `sem_ex` complements `sem_supp` by showing that if the execution initiated by random memory r' terminates successfully (does not yield `None`), then the execution initiated by r (whose support is a subset of r') also terminates successfully.

- Finally, the critical lemma `sem_split` allows us to split an initial state into two distinct parts, each assigned the probability sum corresponding to their supports, and then execute these parts independently. The final resulting state can subsequently be recovered by recombining the outcomes of these separate executions according to their assigned probabilities. This lemma is particularly valuable when reasoning about random If-statements, and by extension, random loops, since a random loop is interpreted as a recursive random If-statement.

While some of these lemmas may appear trivial and could typically be omitted in pen-and-paper proofs, formally verifying them in Isabelle is essential for ensuring the correctness and completeness of our formalization. Other key lemmas, such as the last one (`sem_split`), require notably more detailed and lengthy proofs in Isabelle/HOL compared to their potential pen-and-paper counterparts.

3.5.3 Assertion Formalization

In `Assertion.thy`, we formally define assertion memories—`adm` (assertion deterministic memory) and `arm` (assertion random memory)—along with operations such as partial ordering, separation, and combination, following the concepts introduced in Section 2.2.3. Both `adm` and `arm` are partial functions mapping deterministic or random variables, respectively, to optional values. Each assertion memory has a domain explicitly indicating the variables that hold non-`None` values.

The function `AEval` evaluates expressions over assertion memories. It closely resembles its counterpart for standard memories, `Eval`, but may return `None` when an expression contains variables that are not within the domain of the given assertion memories.

Subsequently, an assertion configuration (`aconfig`) is defined as a record comprising an assertion deterministic memory and a distribution over assertion random memories. For the assertion configuration to be considered well-formed, we require that all random memories within the support of this distribution share the same domain.

Next, we define the combination operator for assertion configurations, denoted by `merge`, along with the associated partial ordering operator `acfLe` (assertion configuration less-or-equal).

- Intuitively, when `merge` x y $=$ z holds, the probabilistic memory distribution within the resulting assertion configuration z can be factored into two independent probabilistic distributions corresponding to x and y, both agreeing on the deterministic memory.
- Furthermore, the ordering `acfLe` x y means that the deterministic memory of x has a domain smaller than or equal to (and consistent with) that of y, and that the probabilistic memory distribution in x is a marginal distribution of that in y, restricted according to their respective domains.

Following Definition 2.3, we formalize atomic assertions as `ap`, along with their semantics as the function `semAP`, which maps each atomic assertion (`ap`) to the set of assertion configurations (`aconfig`) that satisfy it.

For `Certain` assertions, we require that the evaluated expression yields `true` in every memory within its support. For instance, the assertion $\mathsf{Ct}(x = 1)$ demands that the expression $x = 1$ evaluates to `true` across all supports. Assuming the standard definition of equality is included in `op`, this consequently implies that the value of x equals 1 in every support.

Regarding `Uniform` assertions $\mathbf{U}_{e_d}[e]$, we require that all free variables appear within the domain of the corresponding assertion configuration (`aconfig`), the expression e_d is deterministic (containing no random variables), and the probabilistic evaluation of e yields a uniform distribution over the set expressed by e_d.

Notably, uniform assertions explicitly mandate that all free variables appearing within the assertion reside within the domain of the corresponding `aconfig`. In contrast, certain assertions enforce this requirement implicitly, as they demand the expression evaluates to `true`; otherwise, the evaluation would always yield `None`.

Additionally, we introduce several auxiliary functions and lemmas, including the transitivity of `acfLe` and the commutativity of `merge`, which are straightforward to establish.

An important lemma is the monotonicity of atomic assertions with respect to the partial order `acfLe`, formalized as `apMono`. It states that if an assertion configuration m satisfies an atomic assertion a and `acfLe` m m' holds, then m' must also satisfy a. This property ensures that atomic assertions only refer to local information; hence, expanding an assertion configuration does not invalidate satisfaction. Such monotonicity is crucial for the structures (e.g. Kripke resource monoids [Galmiche et al., 2005, Pym et al., 2004]) utilized in both PSL and our logic framework.

Afterward, following Fig. 2.5, we formally define structural (non-atomic) assertions as `assertion`, along with their semantics as `semA`. Similar to atomic assertions, we must also establish monotonicity for these general assertions. The proof of monotonicity relies on several useful lemmas concerning `merge` and `acfLe`, including:

- Lemma `merge_wf` states that merging two well-formed assertion configurations either results in `None` (when their domains overlap) or yields a well-formed configuration, meaning the resulting memory distribution is `validDist` and all its supports share a consistent domain.

- Lemma `merge_ex` states that if `acfLe` $m\ m'$ and `merge` $m'\ c \neq$ `None`, then `merge` $m\ c$ must also exist (i.e. it is not `None`).
- Lemma `merge_le` extends the result of `merge_ex` by additionally establishing: `acfLe` (`merge` $m\ c$) (`merge` $m'\ c$) under the same assumptions.

At the conclusion of `Assertion.thy`, we establish lemma `assertionMono`, which formally demonstrates the monotonicity of assertions.

Assertions operate over *partial* memories, whereas the semantics of the programming language execute over *full* memories. Therefore, it is necessary to connect these two representations. In `Rule1.thy`, we define the functions `partial` and `tototal` to convert between assertion memories and full memories.

This file also contains several useful lemmas that support the proofs of inference rules. We highlight some important ones below:

- Lemmas `sem_MV` and `sem_MV2` state that if a command c does not modify any variable in a set S (i.e. $\mathsf{MV}(c) \cap S = \emptyset$), then the marginal distribution on S remains unchanged after the execution of c. These results imply the constant rule and are also useful in other proofs.
- Lemma `merge_acfLe` shows that if two configurations x and y can be merged into x, then `acfLe` $x\ z$ holds. By the commutativity of merging, it also follows that `acfLe` $y\ z$.
- Lemma `acfLe_merge` establishes the following: given two assertion configurations t_1 and t_2, suppose there exists another assertion configuration t describing the overlapping part of t_1 and t_2 (in terms of their domains), such that

$$\texttt{acfLe}\ t\ t_1 \wedge \texttt{acfLe}\ t\ t_2 \wedge (\mathsf{dom}(\mathsf{t}_1) \cap \mathsf{dom}(\mathsf{t}_2) - \mathsf{dom}(\mathsf{t}) = \emptyset)\,,$$

  then there must exist an assertion configuration t' with domain $\mathsf{dom}(\mathsf{t}_1) \cup \mathsf{dom}(\mathsf{t}_2)$ such that

$$\texttt{acfLe}\ t_1\ t' \wedge \texttt{acfLe}\ t_2\ t'.$$

  Intuitively, this lemma states that two assertion configurations can be merged into a common extension if their marginal distributions agree on the overlapping

domain. This result is instrumental in correcting one of the oversights found in the PSL paper, which will be discussed in detail in Section 3.6.

- Lemma `acfLe_sub` demonstrates that if two assertion configurations m_1 and m_2 both satisfy `acfLe` with respect to another assertion configuration m, and if $\mathsf{dom}(\mathsf{m}_1) \subset \mathsf{dom}(\mathsf{m}_2)$, then m_1 must also satisfy `acfLe` m_1 m_2.

- Lemma `merge_sub` states that if two assertion configurations t and $t2$ can be merged into $t3$ (i.e. `merge` t $t2$ $= t3$), then any marginal configurations of t and $t2$—i.e. a configuration with the same deterministic memory as t or $t2$ and a marginal distribution (with respect to the domain) of its random memory— can be merged into another configuration, which must itself be a marginal configuration of $t3$.

3.5.4 Inference Rules Formalization

Finally, we prove all the main results, assertion implications and inference rules, in `Rule2.thy`.

We begin by formally defining their meanings as predicates in Isabelle named `implies` and `valid`:

- ϕ `implies` ψ holds if and only if, for all well-formed assertion configurations m satisfying ϕ, m also satisfies ψ.

- Following Definition 3.3, the triple $\vdash \{\phi\}\ c\ \{\psi\}$, written `valid` ϕ c ψ, is true if and only if, for all initial configurations m satisfying ϕ, the execution of command c from m—if it terminates—produces a final configuration that satisfies ψ.

Similar to PSL, we establish two important and general lemmas before proving individual conclusions:

- Lemma `restriction` states that for an assertion configuration m satisfying an assertion ϕ, if we take the marginal configuration of m over the domain given by the intersection of the free variables of ϕ and the domain of m, the resulting configuration still satisfies ϕ.

- Lemma `extrusion` shows that for any assertion ϕ implying $\mathbf{D}(\mathsf{FV}(\eta))$, the assertion $(\phi * \psi) \wedge \eta$ `implies` $(\phi \wedge \eta) * \psi$. The premise ϕ implies $\mathbf{D}(\mathsf{FV}(\eta))$ means that any configuration satisfying ϕ must include all variables in $\mathsf{FV}(\eta)$ in its domain.

Unfortunately, the proofs of both lemmas in the PSL paper [Barthe et al., 2019] contain oversights. We correct these in our formalization and will explain the details in Section 3.6.

Now, we present the formalization and proofs of several particularly challenging, interesting and representative results.

3.5.4.1 Proposition `Unif_bij2`

`Unif_bij2` formalizes the final proposition described in Theorem 3.2. It establishes that if two independent distributions—one over a variable x and the other over an expression e—satisfy uniform distributions on the sets S and S', respectively, then for any bijective function f, the expression $f(x, e)$ must satisfy a uniform distribution over the projected set derived from S and S'.

For instance, consider arrays T and T'. If we have $\mathbf{U}_S[T] * \mathbf{U}_{S'}[T']$, then concatenating T and T' yields a distribution that is uniform over all possible concatenation results, provided that S and S' only contain arrays of fixed lengths, ensuring the concatenation operation is bijective on these two sets.

This example illustrates why the bijection condition is specified within the Ct assertion rather than requiring that f be globally bijective: the condition must take local information into account, as concatenation is not bijective when applied globally to arrays of varying lengths.

However, our formalization cannot literally express this proposition because the evaluation of sets within uniform assertions is defined via the abstract function $\mathsf{vset}()$. Specifically, when writing $\mathbf{U}_{S \times S'}[\cdots]$, we implicitly evaluate $\mathsf{vset}(S \times S')$. Yet, there is no guarantee that for arbitrary sets S and S', there exists another expression S'' such that $\mathsf{vset}(S'') = \mathsf{vset}(S) \times \mathsf{vset}(S')$ (i.e. an expression corresponding precisely to the Cartesian product $S \times S'$ may not exist because the type is abstractly defined and we do not assume its existence). This issue arises because we did not anticipate this requirement

during the early stages of our formalization. One possible way to fully resolve this issue would be to utilize concrete expressions instead of abstract ones, which could serve as a direction for future work.

Instead, our formalized lemma introduces an explicit premise assuming the existence of such an expression S'', and accordingly states $\mathbf{U}_{S''}[\cdots]$ as its conclusion. This formulation is semantically equivalent to the proposition described in Theorem 3.2.

3.5.4.2 Unif-Idp rule

`Rule_ind` formalizes the Unif-Idp rule in Fig. 3.3. As Section 3.4 introduced, this rule shows that if intuitively given any arbitrary distribution of input a we can always prove that the result b is uniformly distributed, and the program c does not change a, then a and b are independent in the final state, because the distribution of a does not influence b.

This rule is one of our newly introduced rules, and is the result of a number of design iterations (see the discussion about its precondition in Section 3.4). Notably, the additional premise $b \notin \mathsf{FV}(a)$ emerged directly from our formalization efforts: without this condition, although the resulting distribution remains independent, it cannot be accurately captured by our separation conjunction, which implicitly requires the domains of the separated parts to be distinct.

We prove this rule by applying the lemma `sem_split` (introduced in Section 3.5.2) to partition the execution. Specifically, for any given input distribution of variable a, we partition the execution into multiple sub-executions, each assigned an appropriate probability, ensuring that within each sub-execution, the memory region corresponding to a is deterministic. By individually executing each sub-execution, we can apply the given premise to obtain an identical resulting distribution for b. Finally, we demonstrate that recombining these individual results yields the original (unpartitioned) result, thereby establishing the required independence between a and b.

3.5.4.3 Frame rule and If rules from PSL

Although PSL [Barthe et al., 2019] provides pen-and-paper proofs for their frame rule and If rules, formalizing these rules still demands considerable effort.

As illustrated in Fig. 2.6 and Fig. 2.7, the Frame rule utilizes numerous auxiliary functions. Each auxiliary function necessitates corresponding lemmas that establish guarantees regarding command executions, and each lemma, in turn, requires a two-layer inductive proof on command structures, following the approach described in Section 3.5.2.

Moreover, combining these lemmas to reach the final conclusion of the Frame rule itself is also non-trivial. Ultimately, the formal proof for this rule requires more than 500 lines of code, most of which are not reused in the proofs of other rules.

The random If rules are notably complex, comprising over 1,000 lines of code, approximately 15% of the entire formalization. These rules also involve a key auxiliary concept, which is defined as `supported` (**SP**). We begin by outlining the challenges involved in reasoning about If-statements, and then introduce the role of this concept in addressing those challenges.

Consider our random If rule in Fig. 3.3, which resembles the Hoare Logic rule presented in Fig. 2.3. In our formulation, we require the assertions within this rule to be Ct—that is, they must not reference probabilistic information. To see why, suppose the precondition is $\mathbf{U}_{\mathsf{true},\mathsf{false}}[b]$ which is not Ct. Then, the premises of the rule would have the two branch-specific preconditions: $\mathbf{U}_{\mathsf{true},\mathsf{false}}[b] \wedge \mathsf{Ct}(b = \mathsf{true})$ and $\mathbf{U}_{\mathsf{true},\mathsf{false}}[b] \wedge \mathsf{Ct}(b = \mathsf{false})$. However, each of these is unsatisfiable: asserting that b is uniformly distributed over $\mathsf{true}, \mathsf{false}$ while also fixing b to a single value leads to a contradiction.

As shown in Fig. 2.6, PSL addresses this issue with its RCond rule, which strictly requires the precondition to specify that the distribution of the If condition b is independent from the remaining parts of the precondition. This independence allows us to safely refine the precondition with information such as $b = \mathsf{true}$ or $b = \mathsf{false}$, avoiding the contradiction seen earlier. Under this constraint, the preconditions of the two branches correctly describe the initial states corresponding to each outcome of the If-statement—that is, the conditional distributions given $b = \mathsf{true}$ or $b = \mathsf{false}$.

So far, we have discussed the challenges related to the precondition in the random If rule. However, merging the postconditions of the two branches poses additional difficulties. In both PSL's and our assertion system, even if we establish that the final states of the two branches each satisfy the same assertion ϕ, this does not guarantee that the merged state also satisfies ϕ.

For example, consider the postcondition $\mathsf{Ct}(a = 0) \vee \mathsf{Ct}(a = 1)$, which asserts that the variable a always takes the value 0 or always takes the value 1. If one branch yields a distribution satisfying $\mathsf{Ct}(a = 0)$ and the other satisfies $\mathsf{Ct}(a = 1)$, then the merged distribution allows a to be either 0 or 1—violating the original assertion that a must consistently be one or the other.

To address this issue, PSL introduces the concept of `supported` (**SP**) assertions and requires the following key property:

> If ψ is a `supported` (**SP**) assertion, then given two assertion configurations $(\sigma, \mu) \models \psi * \mathsf{Ct}(b = \mathsf{true})$ and $(\sigma', \mu') \models \psi * \mathsf{Ct}(b = \mathsf{false})$, the merged configuration satisfies $\psi * \mathbf{D}(b)$.

This property underpins the soundness of the RCond rule. In our Isabelle formalization, we establish this result through the lemma `IF_CM`.

These challenges and the conceptual role of supported assertions in reasoning about If-statements are not explicitly addressed in the PSL paper. However, we came to understand them through our formalization effort, and we believe our interpretation is consistent with the intended reasoning in PSL.

3.6 Oversights in original PSL

Our machine-checked proofs identified various oversights in the pen-and-paper formalisation of original PSL [Barthe et al., 2019]. We fixed them either by modifying specific definitions or by finding an alternative—often much more complicated, but sound—proof strategy.

PSL [Barthe et al., 2019] defines the notion of when a formula ϕ is *supported* (**SP**), requiring that for any deterministic memory σ, there exists a distribution over random variable memories μ such that if $(\sigma, \mu') \models \phi$, then $\mu \sqsubseteq \mu'$ (meaning that μ is a marginal distribution of μ' where $\mathsf{dom}(\mu) \subseteq \mathsf{dom}(\mu')$) [Barthe et al., 2019, Definition 6].

This definition aims to restrict the assertions used in PSL's original rule for random conditionals [Barthe et al., 2019, rule RCond of Figure 3], but it is not strong enough. All the assertions satisfy it because μ can always be instantiated with the unit distribution

over the empty memory $\mathsf{unit}(\emptyset \to \mathbf{Val})$, $\sqsubseteq$ (`acfLe`) all others. This means the second example in their paper [Barthe et al., 2019, Example 2] is a counterexample to their rule for random conditionals because every assertion is supported under their definition.

We fixed this by altering their definition of **SP**. Note that simply excluding the empty memory case is not enough to fix this problem. Instead, we have Definition 3.10 and Lemma 3.11 where our Isabelle proofs ensure their soundness. It does not have a big impact on adjusting the proofs strategy of relevant rules.

Definition 3.10 (Supported Assertions)**.** An assertion ϕ is Supported (**SP**) if for any deterministic memory σ, there exists a randomised memory μ such that if $(\sigma, \mu') \models \phi$, then $\mu \sqsubseteq \mu'$ and $(\sigma, \mu) \models \phi$.

Lemma 3.11 (Supported Assertions Construction)**.** *The following assertions are* **SP***:*

$$\eta ::= p_d \mid \mathsf{Ct}(x = e_d) \mid \mathbf{U}_S[x] \mid \eta * \eta$$

where p_d represents assertions that only have deterministic variables and x is any deterministic or random variable.

Additionally, key lemmas that underpin PSL's soundness argument turned out to be true, but not for the reasons stated in their proofs [Barthe et al., 2019, Lemmas 1 and 2, Appendix B].

The proof of Lemma 1 (Restriction, Lemma 3.12 in this thesis) in the PSL paper contains a flaw in the implication case, i.e. when ϕ is of the form $\phi_1 \to \phi_2$. The paper claims:

> Take any $(\sigma', \mu') \models \phi_1$ such that $(\sigma, \pi_{\mathsf{FV}(\phi_1,\phi_2)}(\mu)) \sqsubseteq (\sigma', \mu')$, there exists a distribution μ'' such that $\mathsf{dom}(\mu'') = \mathsf{dom}(\mu') \cup \mathsf{dom}(\mu)$ and $\pi_{\mathsf{dom}(\mu)}(\mu'') = \mu \wedge \pi_{\mathsf{dom}(\mu')}(\mu'') = \mu'$.

However, the existence of such a distribution μ'' is not guaranteed. The assumption ensures that μ and μ' have the same marginal distribution on the domain $\mathsf{FV}(\phi_1, \phi_2)$, but this is insufficient to guarantee the same marginal distribution on the intersection $\mathsf{dom}(\mu) \cap \mathsf{dom}(\mu')$.

In particular, the difference $(\mathsf{dom}(\mu) \cap \mathsf{dom}(\mu')) - \mathsf{FV}(\phi_1, \phi_2)$ may be nonempty, and we have no information about whether μ and μ' have the same marginal distribution on

this part. If they disagree, then no distribution μ'' can satisfy both $\pi_{\mathsf{dom}(\mu)}(\mu'') = \mu$ and $\pi_{\mathsf{dom}(\mu')}(\mu'') = \mu'$. Therefore, the construction of μ'' in this case is invalid, and the proof as presented is unsound.

Instead, our formalized proof of Lemma 3.12 constructs more carefully structured assertion configurations and relies on Lemma `acfLe_sub`, introduced in Section 3.5.3:

Lemma 3.12 (Restriction)**.** *Let (σ, μ) be any configuration and ϕ be any assertion, then: $(\sigma, \mu) \models \phi$ if and only if $(\sigma, \pi_{\mathsf{FV}(\phi)}(\mu)) \models \phi$, where $\pi_{\mathsf{FV}(\phi)}(\mu)$ gives the marginal distribution of μ on the domain $\mathsf{FV}(\phi)$.*

Proof. For the implication case $\phi = \phi_1 \to \phi_2$, we aim to show that for any assertion configuration $(\sigma', \mu') \models \phi_1$ such that

$$(\sigma, \pi_{\mathsf{FV}(\phi_1,\phi_2)}(\mu)) \sqsubseteq (\sigma', \mu'),$$

it also holds that $(\sigma', \mu') \models \phi_2$. We assume the inductive hypothesis that both ϕ_1 and ϕ_2 satisfy the restriction property and the premise $(\sigma, \mu) \models \phi_1 \to \phi_2$ (Definition 3.12).

By the inductive hypothesis applied to ϕ_1, we have $(\sigma', \pi_{\mathsf{FV}(\phi_1)}(\mu')) \models \phi_1$.

Since $\mathsf{FV}(\phi_1) \subseteq \mathsf{FV}(\phi_1, \phi_2)$ and assertions are monotonic with respect to the memory domain, it follows that $(\sigma', \pi_{\mathsf{FV}(\phi_1,\phi_2)}(\mu')) \models \phi_1$.

Next, we apply Lemma `acfLe_sub` to show $(\sigma, \pi_{\mathsf{FV}(\phi_1,\phi_2)}(\mu)) \sqsubseteq (\sigma', \pi_{\mathsf{FV}(\phi_1,\phi_2)}(\mu'))$, since both are sub-configurations of (σ', μ'), and the latter has the larger domain. Furthermore, by the definition of the marginal projection π, we have: $(\sigma, \pi_{\mathsf{FV}(\phi_1,\phi_2)}(\mu)) \sqsubseteq (\sigma, \mu)$.

Combining these, we obtain a core condition that corrects the construction used in the PSL proof: there exists an assertion configuration (σ', μ'') such that

$$(\sigma', \pi_{\mathsf{FV}(\phi_1,\phi_2)}(\mu')) \sqsubseteq (\sigma', \mu'') \quad \text{and} \quad (\sigma, \mu) \sqsubseteq (\sigma', \mu''),$$

where $\mathsf{dom}(\mu'') = \mathsf{dom}(\mu) \cup (\mathsf{FV}(\phi_1, \phi_2) \cap \mathsf{dom}(\mu'))$.

We obtain $(\sigma', \mu'') \models \phi_2$, from the assumption of implication and the construction above, and by applying the inductive hypothesis to ϕ_2 again, we get $(\sigma', \pi_{\mathsf{FV}(\phi_2)}(\mu'')) \models \phi_2$.

Finally, by transitivity of $\sqsubseteq$, we have $(\sigma', \pi_{\mathsf{FV}(\phi_2)}(\mu'')) \sqsubseteq (\sigma', \mu')$, which leads to the conclusion $(\sigma', \mu') \models \phi_2$. □

The proof of Lemma 2 (Extrusion) in the PSL paper also contains a mistake. Specifically, the third line of the proof claims that "we have $(\sigma_1, \mu_1) \models \eta$," but this is not necessarily true because σ_1 may not be equal to σ—its domain could be strictly smaller. The correct proof requires a slightly different strategy that involves adjusting the deterministic memory. Fortunately, this adjustment is relatively straightforward due to its deterministic nature. We identified and formalized a correct version of this proof in Isabelle as well.

Without mechanising the soundness of our program logic, it is unlikely we would have uncovered these issues. This shows the vital importance of mechanised soundness proofs.

3.7 Statistical Distance

So far, our focus has been on perfect security, which requires that observable information is probabilistically independent of secret data—ensuring no information leakage whatsoever. However, as introduced in Section 2.4.1, many practical oblivious algorithms allow for negligible failure probabilities by design. These failures are bounded by a negligible factor (e.g. relative to the size of the secret data), making the algorithms secure in practice. Nonetheless, such probabilistic guarantees cannot be directly verified using our program logic.

In this section, we will introduce the security definition of practical oblivious algorithms with negligible failure probabilities and its relation with the perfect security.

3.7.1 Imperfect Security Definition

We start by stating the formal security property that we target, known in this paper as *ε-Statistical Secrecy*. This property is familiar from standard cryptographic security definitions [Katz and Lindell, 2014]. It is a straightforward relaxation of the following observation. Suppose we have an algorithm that operates over a secret but whose output reveals nothing about that secret. Without loss of generality assume the secret is a

single bit: either 0 or 1. Then, assuming the secret is chosen uniformly (i.e. with equal probability) over $\{0, 1\}$, an attacker who is only able to observe the output of this algorithm can guess the value of the secret correctly with probability no more than $\frac{1}{2}$.

Statistical secrecy simply relaxes this condition to allow a small margin of advantage to the attacker, ε, permitting them to guess correctly with probability at most $\frac{1}{2} + \varepsilon$.

Definition 3.13 (ε-Statistical Secrecy)**.** Suppose a probabilistic algorithm f accepts some secret data S as its input and produces some information $f(S)$ which can be observed by some attacker. We say f satisfies *ε-statistical secrecy* if and only if for any two different secrets S_1 and S_2, if we choose S from the uniform distribution on $\{S_1, S_2\}$ and then reveal $f(S)$ to the attacker, then the attacker's probability of correctly guessing whether S was S_1 or S_2 is at most $\frac{1}{2} + \varepsilon$.

We observe that this property is classical and can be viewed as a specific instance of a quantitative information flow (QIF) guarantee [Alvim et al., 2020]. It concerns an adversary whose prior distribution over the secret is uniform and who is modeled by a gain function that assigns a value of 1 to a correct guess and 0 to an incorrect one. Under this model, the algorithm ensures that the change in the attacker's gain is at most ε.

3.7.2 Verification by Approximation

The security of practical oblivious algorithms can be expressed as a simple instance of ε-statistical security relative to an attacker that can directly observe the memory access pattern, as follows.

Definition 3.14 (Statistical Obliviousness)**.** Suppose an algorithm f takes some secret data S and produces some memory access pattern $f(S)$ which we regard as directly observable by the attacker. We say f is oblivious if and only if for any two different secrets S_1 and S_2 with the same length n, if we choose S from the uniform distribution on $\{S_1, S_2\}$ and then reveal $f(S)$ to the attacker, then that attacker has no greater than probability $1/2 + g(n)$ to guess the value of S correctly, where $g(n)$ is a negligible function of n.

Our approach to proving statistical secrecy is inspired by informal proofs of obliviousness for existing algorithms [Shi, 2019, Sasy and Ohrimenko, 2019, Stefanov et al., 2018,

Ohrimenko et al., 2014], in which reasoning proceeds by "factoring out" the sources of imperfection in the algorithm to consider an implicitly perfect, hypothetical version of the algorithm. Reasoning proceeds by arguing rigorously but informally that the hypothetical version is perfectly oblivious and, therefore, that the original algorithm is oblivious. This last step is performed by quantifying the difference between the original imperfect algorithm and the hypothetical perfect version and using this distance to bound the degree of imperfection and to argue that it is indeed negligible. The measure of difference used is *Statistical Distance* (Definition 2.6, sometimes called *Total Variation Distance*).

The following lemma enables us to decompose the verification task into the two steps introduced above.

Lemma 3.15. *Suppose there is a distribution D and an algorithm f such that for any input S, the statistical distance between $f(S)$ and D is smaller or equal to ε, then f satisfies ε-statistical secrecy.*

This lemma is a well-known result; however, its proof is not readily available in the literature. For completeness, we provide a proof here in the absence of a suitable reference.

Proof. We note firstly that for any two inputs S_1, S_2, the statistical distance between $f(S_1)$ and $f(S_2)$ is at most 2ε, since the distance between each $f(S_i)$ and D is at most ε and by the transitivity of statistical distance (easily proved by definitions).

Let function $g : E \to [0, 1]$ model the attacker's strategy of guessing the result, where E is the set of all possible observations and $g(e)$ represents the probability that the attacker guesses S_1 under observation e; otherwise the attacker guesses S_2 with the probability $1 - g(e)$. Then the overall probability of a correct guess is:

$$\tfrac{1}{2}\sum_{e\in E}(f(S_1)(e)\cdot g(e)) + \tfrac{1}{2}\sum_{e\in E}(f(S_2)(e)\cdot(1-g(e)))$$

Since both $f(S_1)(e)$ and $f(S_2)(e)$ are smaller or equal to $\max(f(S_1)(e), f(S_2)(e))$, the above expression is smaller or equal to $\frac{1}{2}\sum_{e\in E}\max(f(S_1)(e), f(S_2)(e))$.

Then since

$\sum_{e\in E}\max(f(S_1)(e), f(S_2)(e)) + \sum_{e\in E}\min(f(S_1)(e), f(S_2)(e)) =$

$\sum_{e\in E} f(S_1)(e) + f(S_2)(e) = 2$ and

$\sum_{e\in E} \max(f(S_1)(e), f(S_2)(e)) - \sum_{e\in E} \min(f(S_1)(e), f(S_2)(e)) =$
$\sum_{e\in E} |f(S_1)(e) - f(S_2)(e)| = 2\Delta(a, b),$
we have $\sum_{e\in E} \max(f(S_1)(e), f(S_2)(e)) = (2 + \Delta(f(S_1), f(S_2)))/2 \leq 1 + 2\varepsilon$.

Thus we have the correct probability is smaller or equal to

$$\tfrac{1}{2} \sum_{e\in E} \max(f(S_1)(e), f(S_2)(e)) \leq \tfrac{1}{2} + \varepsilon$$

which implies ε-statistical secrecy. □

Thus our approach to verifying an imperfect oblivious algorithm is to build a *perfectly oblivious approximation* of that algorithm, such that for all inputs the statistical distance between the two is bounded by a negligible amount. The proof of this statistical bound is typically carried out manually and lies outside the scope of this thesis.

3.8 Summary

We presented the first program logic that, to our knowledge, is able to verify the obliviousness of real-world foundational probabilistic oblivious algorithms whose implementations combine challenging features like dynamic random choice and secret- and random-variable-dependent control flow. Our logic harnesses the interplay between classical and probabilistic reasoning, is situated atop PSL [Barthe et al., 2019], and proved sound in Isabelle/HOL [Nipkow et al., 2002].

Through the formalization process, we not only gained a deeper understanding of the challenges and motivations behind the design of PSL's rules, but also identified and corrected several oversights in its definitions and proofs.

Finally, we acknowledge the gap between verifying practical oblivious algorithms with negligible failure probabilities and verifying their idealised versions. Bridging this gap remains an open direction for future work, and we discuss it in more detail in Chapter 6.

Chapter 4

Verification of Encryption in Oblivious Algorithms

In Chapter 3, we developed a program logic for verifying probabilistic independence and uniform distribution, focusing on algorithms that consistently produce uniformly distributed observable outputs. However, in addition to the negligible failure probabilities discussed in Section 3.7.2, practical encryption (see Section 2.3) introduces further deviations from ideal uniform distributions.

To illustrate the necessity of encryption in oblivious algorithms, we consider the synthetic program presented in Fig. 4.1. The memory access pattern is uniformly distributed, reflecting that of practical algorithms, though with certain simplifications. Suppose this program receives a secret integer s as input and accesses an attacker-observable array O. The program begins by randomly selecting a number from the set $0, 1$ at line 1, and then writes the encrypted form of the secret to the attacker-observable location $O[r]$ at line 2. Subsequently, it reads and decrypts the ciphertext stored in $O[r]$ at line 3, and finally writes the encrypted value of $2 \times x$ back to the same location, where x is the decrypted value equal to s.

Considering the observable memory access pattern of this program, it consistently performs a write, followed by a read, and then another write at the location $O[r]$, where r is uniformly distributed over $0, 1$. Thus, the overall memory access pattern follows a uniform distribution dictated by r.

```
    synthetic(s) :
1     r ←$ U_{0,1};
2     O[r] ← enc(s);
3     x ← dec(O[r]);
4     O[r] ← enc(2 * x);
```

FIGURE 4.1: Motivating Algorithm for Encryption

To prevent leakage of the secret, encryption is applied to s before writing it to the observable memory location. Additionally, since we subsequently write the encryption of $2 \times s$, the encryption method must be non-deterministic in its plaintext input. If the encryption algorithm enc() always produces identical ciphertexts for identical messages, an attacker could determine whether s is zero simply by checking if the two ciphertexts match. Given this encryption requirement, oblivious algorithms [Ohrimenko et al., 2014, Sasy and Ohrimenko, 2019] frequently employ CCA (Chosen-Ciphertext Attack) secure encryption, as discussed in Section 2.3.3.

Many works opt to sidestep the consideration of encryption in their obliviousness proofs by assuming that an attacker can observe only the memory access patterns, but not the actual data values. (For instance, several oblivious algorithms [Sasy and Ohrimenko, 2019, Stefanov et al., 2018, Shi, 2019] and verification frameworks [Son et al., 2021, Yan et al., 2025]) The assumption significantly simplifies their proof. Such a simplification is often justified: if the data is encrypted, then in practice, an adversary cannot extract meaningful information from it—rendering it effectively equivalent to unobserved data.

However, this approach implicitly assumes that encryption is always correctly applied. If the synthetic program does not encrypt secrets before writing them to an attacker-observable location, the verification may still succeed under the above attack model, as the plaintext remains unobservable to the attacker by assumption.

To address this gap, this chapter presents a formal verification approach with the following contributions:

- In Section 4.4 and Section 4.3, we provide a precise definition of the security guarantees for oblivious algorithms that employ encryption, grounded in either computational (Section 2.3.3) or statistical security (Section 2.3.2) notions.

- In Section 4.2, we define a systematic method for inserting *ghost code* that records observable information, based on the classification of public and private variables (locations).
- We formalize the notion of statistical obliviousness (Definition 3.14), along with its supporting proofs, in Isabelle/HOL [Yan, 2025].
- We establish computational obliviousness through pen-and-paper proofs in Section 4.4.
- As a result, our approach enables explicit and formal verification that encryption is used correctly in oblivious algorithms. We will show several case studies in Chapter 5.

4.1 Overview

As discussed earlier, numerous works avoid addressing encryption explicitly by assuming attackers cannot observe the actual data accessed. Although this simplifies the verification process, it fails to ensure that encryption is implemented correctly.

Following this simplification approach, we have developed a method that additionally verifies the correct usage of encryption. The core idea is to systematically define two methods for embedding ghost code into oblivious programs: the first method inserts ghost code that faithfully records the actual data accessed, including ciphertexts, which results in a complex distribution; the second method inserts ghost code that represents all ciphertexts using a special placeholder value, denoted as $\perp$.

For example, the two transformations of the synthetic programs after embedding ghost code are illustrated in Fig. 4.2. All blue code segments represent ghost code, which record observable information by appending entries to a list Trace without affecting the original execution of the program. We use the symbol '+' to represent list append operations. Each appended tuple begins with either Write or Read to indicate the type of access, followed by the accessed variable or array name and an optional index. The last element of each tuple represents the data accessed: in the left (original) transformation $\mathsf{synthetic}_1()$, it is the actual data, whereas on the right (simplified) transformation $\mathsf{synthetic}_2()$, it is replaced by the special placeholder value.

```
    synthetic1(s) :                                  synthetic2(s) :
0     Trace ← [];                                      Trace ← [];
1     r ←$ U{0,1};                                     r ←$ U{0,1};
2     O[r] ← enc(s);                                   O[r] ← enc(s);
3     Trace ← Trace + (Write, "O", r, O[r]);           Trace ← Trace + (Write, "O", r, ⊥);
4     x ← dec(O[r]);                                   x ← dec(O[r]);
5     Trace ← Trace + (Read, "O", r);                  Trace ← Trace + (Read, "O", r);
6     O[r] ← enc(2 ∗ x);                               O[r] ← enc(2 ∗ x);
7     Trace ← Trace + (Write, "O", r, O[r]);           Trace ← Trace + (Write, "O", r, ⊥);
```

FIGURE 4.2: Motivating Algorithms (Fig. 4.1) with Ghost Code

Note that the values read ($O[r]$, same as $\mathsf{enc}(s)$) are not explicitly recorded in Trace, as this information would be redundant—the attacker can already infer these values from the accessed addresses and previously recorded writes.

Since the left-hand transformation records actual ciphertexts in Trace, the resulting trace might not exhibit a uniform distribution, especially when statistically or computationally secure encryption schemes are employed. This complexity makes direct verification challenging and beyond the scope of existing verification frameworks.

In contrast, the simplified right-hand transformation represents all ciphertexts using the constant placeholder value $\perp$, preserving a uniform distribution in the memory access patterns. Consequently, its corresponding Trace remains uniformly distributed, enabling verification of security through our program logic introduced in Chapter 3.

Intuitively, our goal is to demonstrate that if we can verify the security of an algorithm with the simplified ghost code (right-hand transformation), the security of the algorithm with the original ghost code (left-hand transformation) naturally follows. Note that this simplification only replaces ciphertext values with the special placeholder $\perp$. Any unencrypted values will still be recorded accurately, and thus their security must be explicitly verified using the corresponding program logic. Consequently, any algorithm verified by our approach must encrypt all secret data before writing it to attacker-observable locations, although it may still operate with certain non-secret plaintext values.

However, to formally articulate and prove the intuition described above, we face the following challenges:

- How can we define the security of Trace whose distribution consists of 1. a (marginally) uniformly distributed memory access pattern and 2. unknown ciphertext distributions?
- How can we formally define the two transformations to ensure they satisfy the desired properties outlined earlier?
- How can we prove that the security of the simplified transformation implies the security of the original transformation?

Towards a Security Definition. Consider the distributions of Trace generated by the original transformation $\mathsf{synthetic}_1()$ with two different inputs, $s = 0$ and $s = 1$. In both cases, there is a 50% chance of obtaining the trace

$$[(\mathsf{Write}, \text{“}O\text{”}, 0, \mathsf{enc}(s)), (\mathsf{Read}, \text{“}O\text{”}, 0), (\mathsf{Write}, \text{“}O\text{”}, 0, \mathsf{enc}(2 * s))]$$

and the other 50% to get

$$[(\mathsf{Write}, \text{“}O\text{”}, 1, \mathsf{enc}(s)), (\mathsf{Read}, \text{“}O\text{”}, 1), (\mathsf{Write}, \text{“}O\text{”}, 1, \mathsf{enc}(2 * s))]$$

where the variation arises from the random choice of r. The corresponding encrypted plaintexts $[s, 2s]$ are either $[0, 0]$ when $s = 0$, or $[1, 2]$ when $s = 1$.

Suppose the encryption scheme enc() satisfies perfect secrecy (Section 2.3.1). Then, the distributions of enc(0), enc(1), and enc(2) are identical. In this case, different secrets result in identical distributions of Trace, implying that the observable trace leaks no information about the underlying secret.

However, recall from Section 2.3.1 that apart from one-time pads, perfect secrecy is not realistic for most practical encryption schemes. If enc() only provides statistical secrecy (see Section 2.3.2)—then the distributions of enc(0), enc(1), and enc(2) may differ. As a result, Trace may leak partial information about the secret input.

In this case, the overall security should be relaxed to a corresponding statistical security notion—specifically, we can reuse Definition 3.14 that requires the leakage from Trace to be negligible. Suppose the leakage from a single ciphertext is a negligible function of the security parameter λ and that each ciphertext is generated independently from

its corresponding set of possible values. (The independence of ciphertext generation is typically implicit in standard security definitions. For instance, in the IND-CCA game introduced in Section 2.3.3, any sequence of plaintexts must yield indistinguishable ciphertext sequences. This implicitly requires that sequential encryptions do not exhibit excessive dependence.) Then, the overall leakage from Trace remains negligible in λ as long as the maximum number of ciphertexts appearing in any support of Trace is bounded by a polynomial in λ. We provide the formalization of supporting proof in Section 4.3.

Furthermore, if the encryption scheme enc() provides only computational secrecy (see Section 2.3.3), the overall security definition must also be relaxed to a corresponding computational security notion. This notion is game-based and inherently more complex. We present the formal definition and supporting proof in Section 4.4.

Transformation. While the intuition is illustrated in Fig. 4.1 and Fig. 4.2, we also provide a formal definition of the transformation based on the programming language introduced in Section 3.2. This formalization establishes a systematic method for inserting ghost code to record observable traces.

Although there are several detailed requirements of target program and steps to transform which will be detailed in Section 4.2, intuitively the transformation guarantees that the two (original and simplified) transformations must satisfy some relation for decomposing the indistinguishability of the original trace into two components.

We will prove that the original trace distribution can be *indistinguishably simulated* by the simplified trace distribution, along with a sequence of ciphertext distributions produced by the encryption oracle enc(0). If both the simplified trace and the ciphertext distributions are indistinguishable to the attacker, then the original trace—being a composition of these—is also indistinguishable and thus secure.

Intuitively, this indistinguishable simulation is justified because the simplified trace distribution is obtained by replacing recorded ciphertexts with a special symbol $\perp$ during the execution of the transformed program. The original trace can then be simulated by reversing this replacement with enc(0). Detailed description and relevant proof of this property is provided within statistical and computational assumptions respectively.

Statistical Soundness Proof Given the definitions of security and transformation, we now aim to prove that our verification approach—combined with the transformation—implies the stated security definitions.

In Section 4.3, we formalize the proof of statistical security under the assumption that a statistically secure encryption scheme is used. This proof is mechanized in Isabelle/HOL and follows an inductive structure, analyzing each command and its corresponding transformation individually.

The main challenge lies in formulating appropriate intermediate conditions that satisfy the following criteria:

1. They must describe the relationship between the two transformed programs under the assumptions of the target program.
2. They must hold in the initial states.
3. They must imply the overall security definition, so that the final conclusion can be derived from these conditions.
4. They must remain true at every intermediate program state.
5. Assuming they hold at the beginning of a command, they must persist after executing that command.

The second and fourth criteria require that the intermediate condition must not be too strong, while the third and last criteria demand that it must not be too weak. These conflicting requirements make the formulation of a suitable intermediate condition particularly challenging. As a result, we iterated through several versions during the development process, and the conclusion is discussed in detail in Section 4.3.

Note that this proof also implies the corresponding result for perfect secrecy, because (recall from Section 2.3.2 that) perfect secrecy can be viewed as a special case of statistical security, where the statistical distance is zero.

Computational Soundness Proof Finally, we turn our attention to the computational counterpart. As described in Section 2.3.3, computational security is defined through interactive games, and unlike statistical or perfect security, it does not rely

on a precise characterization of ciphertext distributions. This fundamental difference necessitates a distinct proof strategy compared to the previous cases.

Following the approach used in standard cryptographic texts [Katz and Lindell, 2014] and the Melbourne shuffle [Ohrimenko et al., 2014], computational security can be established via a reduction. The idea of a reduction-based proof is to show that if an adversary could break the security of our transformed program, then one could use this adversary to construct a successful attack against the underlying encryption scheme.

In other words, we reduce the security of the overall system to the security of the encryption primitive: assuming the encryption scheme is secure (e.g. IND-CCA secure), it follows that no efficient adversary can distinguish between the outputs of the transformed programs. This is typically achieved through a series of hybrid game-based transitions, where the adversary's advantage is bounded and shown to be negligible under the computational assumptions.

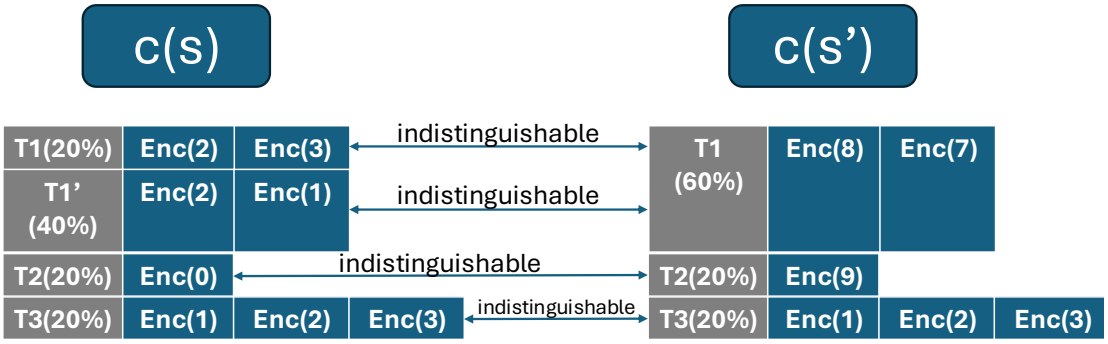

FIGURE 4.3: Indistinguishable Distributions

However, the main challenge is that our trace distribution has two random sources: memory access pattern and the data (ciphertext) accessed, which means for a final distribution of the trace, it may have several possible memory access patterns, and in each possible memory access pattern, the ciphertext sequence distribution may itself be produced by the combination of several plaintext sequence, as shown by Fig. 4.3.

The overall indistinguishable distributions in Fig. 4.3 consist of several pairs of indistinguishable sub-distributions. For example, suppose program c generates three possible memory access patterns which are not secret depending (60% of length 2, 20% of length 1 and 20% of length 3 as Fig. 4.3 shown), each of which corresponds to a plaintext-sequence distribution that produces indistinguishable ciphertexts.

However, beyond this illustrative case, the number of possible memory access patterns can grow exponentially (i.e. non-polynomially) in the size of input caused by random choices. Moreover, for each such access pattern, there may be multiple corresponding plaintext sequences (the two possible results of length two in $c(s)$), making the total number of possibilities even larger—even before accounting for the randomness introduced by encryption.

These nested distributions make direct comparison between two traces (produced by different secret inputs) in a reduction proof intractable, even when using a chain of hybrid games. The core difficulty lies in the need to match a given trace, whose ciphertexts are generated from a specific plaintext sequence, with a corresponding plaintext sequence from another secret execution such that their probabilities align. However, the number of valid plaintext sequences can be exponential, making it computationally infeasible to identify such a match.

To address this challenge, we propose an indirect proof strategy: rather than comparing the two traces produced by different secret inputs directly, we introduce an intermediate trace that shares the same memory access pattern distribution as the target traces, but where all ciphertexts are generated by $\mathsf{enc}(0)$. This construction ensures that the corresponding plaintext sequence in both cases is simply a sequence of zeros, making the comparison tractable.

Consequently, our proof strategy involves two parallel reductions—one for each secret input and the intermediate one—rather than a single reduction path. Naturally, we also require an additional reduction step to connect these two indirect comparisons. We present the details of this approach in Section 4.4.

Verification of the Motivating Algorithms Finally, we verify the motivating algorithms and their associated security guarantees. As outlined earlier, this involves verifying the algorithm produced by the simplified transformation, as depicted in Fig. 4.4. The transformation presented in this section is intended to convey the intuition behind our verification approach; the formalised transformation process, which introduces some differences in detail, will be discussed later in Section 4.2.

We begin by initializing Trace as an empty list, obtaining the corresponding assertion via the Random Assign rule. The assertion of uniformity is then established

$\mathsf{synthetic}_2(s)$:
  $\mathsf{Trace} \leftarrow []$;
  $\{\mathsf{Ct}(\mathsf{Trace} = [])\}$
  $r \leftarrow_{\$} \mathbf{U}_{\{0,1\}}$;
  $\{\mathsf{Ct}(\mathsf{Trace} = []) \wedge \mathbf{U}_{\{0,1\}}[r]\}$
  $O[r] \leftarrow \mathsf{enc}(s)$;
  $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\mathsf{Write}, \text{“}O\text{”}, r, \perp)$;
  $\{\mathsf{Ct}(\mathsf{Trace} = [(\mathsf{Write}, \text{“}O\text{”}, r, \perp)]) \wedge \mathbf{U}_{\{0,1\}}[r]\}$
  $x \leftarrow \mathsf{dec}(O[r])$;
  $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\mathsf{Read}, \text{“}O\text{”}, r)$;
  $\{\mathsf{Ct}(\mathsf{Trace} = [(\mathsf{Write}, \text{“}O\text{”}, r, \perp), (\mathsf{Read}, \text{“}O\text{”}, r)]) \wedge \mathbf{U}_{\{0,1\}}[r]\}$
  $O[r] \leftarrow \mathsf{enc}(2 * x)$;
  $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\mathsf{Write}, \text{“}O\text{”}, r, \perp)$;
  $\{\mathsf{Ct}(\mathsf{Trace} = [(\mathsf{Write}, \text{“}O\text{”}, r, \perp), (\mathsf{Read}, \text{“}O\text{”}, r), (\mathsf{Write}, \text{“}O\text{”}, r, \perp)]) \wedge \mathbf{U}_{\{0,1\}}[r]\}$
  $\{\mathbf{U}_T[\mathsf{Trace}]\}$
  where $T = \{[(\mathsf{Write}, \text{“}O\text{”}, r, \perp), (\mathsf{Read}, \text{“}O\text{”}, r), (\mathsf{Write}, \text{“}O\text{”}, r, \perp)] \mid r \in \{0, 1\}\}$

FIGURE 4.4: Verification of the Motivating Algorithm

by applying the Random Sampling rule in conjunction with the Weak (Consequence) rule. Subsequently, we continue to apply the Random Assign rule and the Weak rule throughout the algorithm. This process yields an assertion that characterizes Trace in terms of the random variable r and confirms that r is uniformly distributed. As a result, it follows that the value of Trace is also uniformly distributed over the set $\{[(\mathsf{Write}, \text{“}O\text{”}, r, \perp), (\mathsf{Read}, \text{“}O\text{”}, r), (\mathsf{Write}, \text{“}O\text{”}, r, \perp)] \mid r \in \{0, 1\}\}$ by applying the third rule in Proposition 3.2.

By combining the verification triple above with the definitions and theorems to be introduced in Section 4.3 and Section 4.4, we conclude that if the encryption scheme $\mathsf{enc}()$ satisfies statistical or computational security, then the original algorithm shown in Fig. 4.1 is statistically or computationally oblivious, respectively.

Assumptions on Context To make our assumptions explicit, we clarify the adversarial setting and cryptographic context inherited from the standard statistical and IND-CCA security models. For example, in the IND-CCA setting and in our corresponding discussion in Section 4.4, we assume that the encryption scheme's secret key is generated once at the beginning of the experiment and remains fixed throughout the game. The adversary is modelled as a probabilistic polynomial-time algorithm that may adaptively

interact with our target algorithms, exactly as in the traditional IND-CCA experiment, which abstracts away the internal details of key management and decryption.

Our framework builds directly on these assumptions: we do not redefine these mechanisms but instead rely on the guarantees provided by an underlying encryption scheme that already satisfies statistical or IND-CCA security, each with its own contextual interpretation.

4.2 Transformation

We first extend the programming language by introducing a new command, $x_r \leftarrow \mathsf{enc}(e_r)$, which invokes the encryption oracle on plaintext expression e_r and stores the resulting ciphertext in variable x_r. To maintain generality and allow for the application of different security notions, we make no assumptions about the distribution of $\mathsf{enc}()$—it may return ciphertexts according to an arbitrary distribution. Consequently, the existing random choice command, which always returns values uniformly at random, is insufficient to model this behavior.

Definition 4.1 (Encryption Semantics)**.** In addition to Fig. 3.2, we have

$$[\![x_r \leftarrow \mathsf{enc}(e_r)]\!](\sigma, \mu) = (\sigma, \mathsf{bind}(\mu, m \mapsto \mathsf{bind}(\mathsf{Enc}(e_r), u \mapsto \mathsf{unit}(m[x_r \mapsto u]))))$$

where stateless function $\mathsf{Enc}(e_r)$ returns a distribution of **Val**. We make no assumptions about what distribution $\mathsf{Enc}()$ returns, and leave it under-defined, for maximum generality.

Note that although encryption functions are often stateful in practice, they are typically modeled as stateless in theoretical settings, including both computational and statistical security definitions.

Definition 4.2 (Transformation Requirements)**.** We impose several requirements on the target programs to ensure unambiguous access recording and obliviousness. None of these requirements meaningfully impact the expressiveness of our approach, as we explain below.

- A pre-defined set of observable locations (public variables) must be specified, so that memory accesses to these variables can be properly recorded via the ghost code transformation. Alternatively, their classification can be made value-dependent, as demonstrated in [Murray et al., 2016]. However, in this thesis, we assume that the set of observable locations is fixed in advance.

- Each command must access at most one public variable. Otherwise, the order of memory accesses becomes ambiguous. For example, in the command $x \leftarrow a + b$, if both a and b are public, it is unclear which variable is read first. To avoid this ambiguity, such commands should be rewritten—for instance, as $x \leftarrow a$; $x \leftarrow x + b$, where x is a non-public variable.

- For conditional statements (e.g. `if`) and loops (e.g. `while`), the guard condition must not directly access public variables. Otherwise, it becomes difficult to immediately insert the corresponding ghost code to record observable behavior. If a condition involves a public variable, it should first be loaded into a private variable, which is then used in the control flow. This ensures that the observable access is recorded explicitly and separately from the control logic.

- Encrypted expressions must not contain public variables. If an encrypted expression does reference a public variable, it should first be rewritten so that the public value is loaded into a private variable before encryption. This restriction simplifies the definition of the transformation, as we avoid the need to handle cases where encryption directly involves reading public variables.

- Direct access to any ciphertext (i.e. values encrypted via $\mathsf{enc}()$) is prohibited, except through the decryption function $\mathsf{dec}()$. Allowing arbitrary operations on ciphertexts may result in unintended leakage. For instance, consider a program that reveals the ciphertext x of the secret value s and subsequently reveals $x\%2 + s\%2$ (with '%' denoting the modulo operation). Since an attacker knows x, they could deduce the value of $s\%2$. However, our simplified transformation records the first revealed value x as $\perp$, rather than its actual value, thereby will miss such leakage. We therefore require that ciphertexts be used only through the decryption function $\mathsf{dec}()$.

We note that the first four requirements do not affect the expressiveness of the program. Any program that does not satisfy them can be re-written to a semantically equivalent

one that does. Regarding the final requirement, it is generally satisfied by target programs, with the exception of a few cases involving direct copying of ciphertexts, which can be readily rewritten by adding re-encryption. With these requirements, we define the two transformations, which are formalized in Isabelle/HOL [Yan, 2025]:

Definition 4.3 (Transformations)**.** Given a program that satisfies the requirements stated above, we apply the following transformation steps to obtain an instrumented version that records memory access patterns along with the accessed data:

1. Introduce an unused variable to record observable information, and initialize it as an empty list. Without loss of generality, we name this variable Trace.

2. For any command (excluding encryption) that performs a write to a public variable x, append the ghost code

$$\mathsf{Trace} \leftarrow \mathsf{Trace} + (\mathsf{Write}, \texttt{`x'}, x)$$

   immediately after the command, where `'x'` denotes the name of the variable and x its current value (after the write).

3. For any command (excluding encryption) that reads from a public variable x, append the ghost code immediately after the command.

$$\mathsf{Trace} \leftarrow \mathsf{Trace} + (\mathsf{Read}, \texttt{`x'})$$

4. For any encryption command $x \leftarrow \mathsf{enc}(s)$ where x is public, append one of the following codes:

   - $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\mathsf{Write}, \texttt{`x'}, x); x \leftarrow s$; for the *original transformation*, where the actual ciphertext value is recorded;
   - $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\mathsf{Write}, \texttt{`x'}, \bot); x \leftarrow s$; for the *simplified transformation*, where $\bot$ denotes an abstract placeholder for the ciphertext, masking the actual encrypted content.
   - $x \leftarrow \mathsf{enc}(0); \mathsf{Trace} \leftarrow \mathsf{Trace} + (\mathsf{Write}, \texttt{`x'}, x); x \leftarrow s$; for the *simulated transformation*, where the actual ciphertext is replaced by $\mathsf{enc}(0)$. This transformation is only for the computational proof in Section 4.4.

Note that the plaintext is written to x after recording the ciphertext in Trace. The purpose of this is to ensure that Trace accurately captures all observable information while simultaneously simplifying the decryption process which we explain as follows.

The security standards described in Section 2.3 assume that the encryption key is randomly generated before encryption occurs, and they do not address the decryption phase. Consequently, the precise definition of decryption remains unspecified. Since our only permitted access to ciphertext data is through the act of decryption, and we avoid recording the values read during decryption in Trace as discussed in Section 4.1, we can safely write the plaintext s directly to x after the ciphertext has been logged. This approach allows us to read plaintext directly when needed, thereby circumventing the need for an explicit decryption definition without affecting the recorded observations in Trace.

5. Finally, return Trace as the observable information.

4.3 Statistical Obliviousness

Following the discussion presented in Section 4.1, Section 4.2, and recalling Definition 3.14, we establish the following theorem.

Theorem 4.4 (Statistical Obliviousness with Encryption)**.** *Let λ be a security parameter, and let c_0 be a program satisfying all conditions specified in Definition 4.2, which employs an encryption scheme that is statistically secure with advantage at most* $\mathsf{negl}(\lambda)$*. Consider the programs c and c', which represent the original and simplified transformations of c_0, respectively. If the triple $\vdash \{\phi\}\ c'\ \{\mathbf{U}_T[\mathsf{Trace}]\}$ holds such that the precondition ϕ asserts the inputs are valid for the program c' and the length of each trace $t \in T$ is bounded by a polynomial in λ, then the program c achieves statistical obliviousness according to Definition 3.14, provided its input length is also bounded by a polynomial in λ.*

Later we explain the Isabelle formalisation of this theorem and proof in Section 4.3.2.

Intuitively, the theorem asserts that if a program c_0, which employs statistically secure encryption, meets our specified requirements (see Theorem 4.2), and its simplified

transformation is proven to always yield a fixed trace distribution (by our program logic of Chapter 3), then the memory access pattern—observed under actual ciphertext execution and recorded by the variable Trace in the original transformation—satisfies statistical obliviousness (see Theorem 3.14). The structure and formalization of the proof will be presented in the following subsections.

4.3.1 Proof Structure

As discussed in Section 4.1, we need to define suitable intermediate conditions that specify the relationship between states of the original and simplified transformations.

Definition 4.5 (Intermediate Conditions)**.** For any program states (σ, μ) and (σ', μ'), we say they satisfy the intermediate conditions, denoted by $\mathsf{MC}(\sigma, \mu, \sigma', \mu')$, if and only if the following conditions hold:

- The states differ only in Trace. Formally, $\sigma = \sigma'$ and $\pi_{\mathtt{UNIV}-\mathsf{Trace}}(\mu) = \pi_{\mathtt{UNIV}-\mathsf{Trace}}(\mu')$, where $\pi_{\mathtt{UNIV}-\mathsf{Trace}}(\mu)$ denotes the marginal distribution of μ on all variables excluding Trace.
- In both μ and μ', Trace records a sequence of tuples as exemplified in Section 4.1. If each ciphertext entry in Trace from all elements in the support of μ is replaced with $\perp$, the resulting distribution is μ'.
- Assume that the encryption scheme $\mathsf{enc}()$ has ε-statistical secrecy (Definition 3.13). For each possible random memory $m' \in \mathsf{supp}(\mu')$, define μ_m as the conditional distribution of μ given the condition that replacing every ciphertext in Trace with $\perp$ will yield m'. In contrast, define μ'_m as the distribution obtained by replacing each $\perp$ in the Trace of m' with encryptions of 0 according to $\mathsf{enc}(0)$. The statistical distance between distributions μ_m and μ'_m must be bounded by $\varepsilon \cdot n$, where n is the length of Trace in m'.

We prove three key properties about the *Intermediate Conditions* for our conclusion.

- Given identical inputs for the original transformation c and its simplified counterpart c', the *Intermediate Conditions* $\mathsf{MC}(\sigma, \mu, \sigma', \mu')$ must hold for the states (σ, μ) and (σ', μ') resulting from the initialization step (step 1 in Definition 4.3)

of c and c', respectively. Because the inputs to both transformations are identical and the initial Trace is empty, it follows that $(\sigma, \mu) = (\sigma', \mu')$. Thus, the first condition of $\mathsf{MC}(\sigma, \mu, \sigma', \mu')$ is immediately satisfied. Additionally, since Trace is initially empty in both cases, the remaining two conditions hold trivially without any replacements.

- Given the *Intermediate Conditions* $\mathsf{MC}(\sigma, \mu, \sigma', \mu')$ and two programs c and c' obtained by applying Definition 4.3 to a program c_0 that satisfies Definition 4.2, suppose that execution yields $[\![c]\!](\sigma, \mu) = (\sigma_1, \mu_1)$ and $[\![c']\!](\sigma', \mu') = (\sigma'_1, \mu'_1)$. Then, it follows that $\mathsf{MC}(\sigma_1, \mu_1, \sigma'_1, \mu'_1)$ also holds. Establishing this result requires an induction on the structure of c_0, with individual cases considered separately for each type of command. We will present the details of this inductive proof in Section 4.3.2.

- Given the *Intermediate Conditions* $\mathsf{MC}(\sigma, \mu, \sigma', \mu')$ and assuming that the encryption scheme $\mathsf{enc}()$ satisfies ε-statistical secrecy, it follows that the statistical distance between the distributions of Trace in μ and μ' is bounded by $\mathsf{poly}(n) \cdot \varepsilon$, where n is the maximum length of Trace within the supports of μ and μ'. This conclusion is closely related to the last condition of the *Intermediate Conditions* (Definition 4.5); however, here we consider the overall distributions of Trace, whereas the *Intermediate Conditions* address individual conditional distributions, each conditioned on a particular simplified Trace. Since the overall distribution of Trace can be expressed as the weighted sum of these conditional distributions, the stated bound on statistical distance directly follows from the definition of statistical distance (Definition 2.6).

Given the above properties, we can now establish Theorem 4.4.

Proof of Theorem 4.4. Our goal is to prove that for any two secret inputs s and s', the traces Trace produced by executing $c(s)$ and $c(s')$ are statistically indistinguishable. By Lemma 3.15, it suffices to show that the statistical distance between these two traces is a negligible function in the security parameter λ.

We proceed by considering the executions of the simplified programs $c'(s)$ and $c'(s')$. By the first two properties stated above, the final states of $c(s)$ and $c'(s)$ satisfy the *Intermediate Conditions*, and similarly for $c(s')$ and $c'(s')$.

Then, by the third property, there exists a function f such that the statistical distance between Trace in $c(s)$ and $f(\mathsf{Trace})$ in $c'(s)$ is at most $\mathsf{poly}(n) \cdot \mathsf{negl}(\lambda)$, where n is polynomially bounded in λ, as assumed in Theorem 4.4. An analogous bound holds for $c(s')$ and $c'(s')$.

Furthermore, since c' is assumed to always produce a fixed distribution over Trace in Theorem 4.4, we have that $f(\mathsf{Trace})$ in $c'(s)$ is identically distributed to $f(\mathsf{Trace})$ in $c'(s')$. Consequently, both Trace in $c(s)$ and Trace in $c(s')$ are within statistical distance $\mathsf{poly}(\mathsf{poly}(\lambda)) \cdot \mathsf{negl}(\lambda)$ of the same fixed distribution—namely, $f(\mathsf{Trace})$ in $c'(s')$.

By the transitivity of statistical distance, the distance between Trace in $c(s)$ and Trace in $c(s')$ is bounded by $2 \cdot \mathsf{poly}(\mathsf{poly}(\lambda)) \cdot \mathsf{negl}(\lambda)$, which simplifies to $\mathsf{negl}(\lambda)$, due to the closure of negligible functions under polynomial scaling. This completes the proof. □

4.3.2 Isabelle Formalization

We formalize the transformation (Section 4.2) and the statistical obliviousness proof (Section 4.3) in Isabelle/HOL, complementing the formalization of our program logic presented in Chapter 3. The full artifact is available on Zenodo [Yan, 2025] and contains approximately 4,500 additional lines of code.

We begin by modifying the semantics of the programming language according to Theorem 4.1, implemented in `Semantics.thy`, by introducing a new command for encryption. Additionally, we introduce an axiom asserting that all ciphertext distributions form full probabilistic distributions. Relevant auxiliary functions and existing proofs are adapted accordingly to accommodate the new command.

The remaining formalization is contained in `Transform.thy`. We first assume the existence of functions that distinguish between public and private variables, as well as a designated variable Trace used to record observable information, which is initialized to an `empty` value.

Next, we assume the existence of a distinguished `special` value (denoted $\perp$ in this thesis), along with an abstract function `Cval`, which maps a command to a value for the purpose of recording memory access patterns. In practice, `Cval` is intended to capture both the access type and the accessed memory location. For our formalization,

we additionally require that `Cval` is capable of distinguishing encryption operations from other commands, in order to determine whether the accessed data corresponds to ciphertext.

Finally, we introduce a set of axioms concerning the `append` operation on lists. These are necessary due to our use of an abstract definition of values, which prevents us from relying on the built-in list append properties in Isabelle. The asserted axioms, such as the injectivity of list append, are consistent with the standard definitions provided in Isabelle/HOL.

Transformations We formalize the requirements for program transformations using the `primrec` construct in Isabelle/HOL to define the primitive recursive function `WellFormed` in accordance with Definition 4.2. The original transformation procedure is formalized as `tsfm`, and the simplified transformation as `tsfmS`. The simulated transformation is not formalized in Isabelle as it is only for computational proof.

As described in Section 4.1, we omit the recording of actual read values in Trace, since such information is redundant.

Termination To support the proof in Section 4.3.1, we show that when the program resulting from the original transformation terminates, then this implies that the program resulting from the simplified transformation also terminates, when given the same inputs. This guarantees the existence of corresponding pairs of terminating executions.

This property holds because the only difference between the two transformations lies in the handling of Trace, which is a ghost variable and does not affect control flow. To establish this result in Isabelle/HOL, we apply the two-layer induction principle introduced in Section 3.5.2. The proof relies on an intermediate condition, `eq_but_trace`, which asserts that two program states are identical except for Trace. This condition also constitutes the first part of our main *Intermediate Conditions* (Definition 4.5).

Intermediate Conditions We define our main definition, *Intermediate Conditions*, as `mid_condition`, in accordance with Definition 4.5. The first component reuses the predicate `eq_but_trace`, while the second employs the function `simplify`, which replaces all ciphertexts in Trace with the `special` symbol. This operation reduces the support of

the Trace distribution, since ciphertext distributions are collapsed into a single symbolic value.

The remaining conditions assert that the statistical secrecy of encryption implies the statistical distance property described in the final part of Definition 4.5. To formalize this, we introduce the auxiliary function `simulate`, which replaces each occurrence of the `special` symbol in Trace with enc(0). This operation expands the Trace distribution by `bind`ing the distribution of enc(0) to each `special` symbol, thereby effectively reversing the effect of `simplify`.

Suppose a trace Trace is first transformed by `simplify` and then by `simulate`; the resulting trace will have a negligible statistical distance from its original distribution. This is because the original ciphertexts may have been produced by encrypting values other than 0, and statistical secrecy ensures that such ciphertexts are indistinguishable from enc(0).

Many useful lemmas follow from these definitions, providing foundational results that support the soundness of our transformation and statistical obliviousness proofs.

Implications of Intermediate Conditions We prove a lemma `mid_init`, which states that the *Intermediate Conditions* are satisfied by the initial states of the two transformations. In addition, we establish another lemma, `mid_final`, which shows that the *Intermediate Conditions* imply our desired conclusion: namely, that the statistical distance between the two traces Trace, produced under different secret inputs, is negligible.

The lemma `mid_final` encapsulates the reasoning introduced in the third part of the *Intermediate Conditions*, as well as the proof of Theorem 4.4 in Section 4.3.1. The overall proof strategy follows the structure described in Section 4.3.1, supported by several auxiliary lemmas, all of which are grouped in the `mid_final` section (as indicated by comments in the formalization).

Inductive Proof on Commands Finally, we demonstrate that the `Intermediate Conditions` remain invariant throughout the transformations applied to each command. This inductive argument constitutes the core of our formalization, accounting for approximately 70% of the total lines of code.

The primary inductive structure is encapsulated in lemma `st_mid_ind`, which comprises the following cases:

- **Case skip:** No changes occur, and thus the `Intermediate Conditions` are trivially preserved.
- **Case Deterministic Assignment:** Since random memory remains unaffected, the `Intermediate Conditions` are trivially preserved in this scenario as well.
- **Case Random Assignment:** This command involves adding a ghost command to record memory accesses whenever public variables are present. Since both transformations incorporate identical ghost commands, preservation of the intermediate conditions might seem straightforward. However, the actual proof is nontrivial because assignment commands may merge certain probabilistic outcomes (supports). For example, consider a variable x initially following a uniform distribution over the set $\{0, 1\}$. Executing the command $x \leftarrow 0$ merges two distinct outcomes into one. Given that the `Intermediate Conditions` reference supports for all variables except Trace, reconstruction of them is required after potential merging caused by assignments. Lemma `SD_psum_eq` establishes that the statistical distance remains negligible upon merging distributions, and lemma `mid_mid_Rassign` (the conclusion of this case) leverages this result to handle the random assignment case.
- **Case Random Sampling:** This command inherits the complexity discussed above, including the necessity of lemma `SD_psum_eq`. Additionally, random sampling may also expand supports due to the introduction of uniform distributions. The formalized semantics of this command involves a two-layer bind structure. Since the `Intermediate Conditions` incorporate conditional probabilities within their final and central statistical distance condition, this case's proof necessitates carefully unfolding these layers and constructing precise mathematical equations, which is captured by lemma `mid_mid_Sample`.
- **Case Encrypt:** The ghost commands added by this transformation can differ, making this case particularly critical. Each transformation introduces two assignment commands, resulting in a multi-layered bind structure similar to the previous cases but with distinct requirements for detailed unfolding. The proof for this case is divided into two parts:

Lemma `mid_mid_Core_half` addresses all `Intermediate Conditions` except their final statistical distance condition, leveraging the earlier random assignment proof.

Lemma `mid_mid_Core` establishes the final statistical distance condition based on previously proven conditions (by `mid_mid_Core_half`). It demonstrates that the tuple added to Trace in the original transformation records ciphertext distributions freshly generated by enc(), which can be `simulated` by enc(0) with negligible statistical distance, as assumed in the encryption scheme. Moreover, this only adds a negligible amount to the initial statistical distance defined in the `Intermediate Conditions`.

However, this result only holds true after the final added command $x \leftarrow s$. Prior to this command, x also stores the ciphertext, causing conditional probabilities on all variables except Trace to yield only a single ciphertext of the last tuple rather than the intended full distribution.

Lemma `SD_increase` plays a crucial role in this detailed proof, stating that merging two pairs of distributions—each pair having statistical distances n and n', respectively—by a reversible and injective function g results in a combined statistical distance of $n + n'$. This lemma is central to handling the addition of ciphertext to Trace. Additionally, auxiliary lemmas such as `mid_mid_simp` significantly aid in mathematical simplifications within this proof.

- **Case Sequence, Case Deterministic If:** Trivial by unfolding the definitions and applying the induction hypothesis.
- **Case Random If:** Executing a random If-statement involves splitting an initial distribution into two conditional distributions corresponding to each branch. After executing both branches separately, the resulting distributions must be merged. The critical step is demonstrating that these splitting and merging operations preserve the `Intermediate Conditions`. This preservation is established via mathematical simplifications and the application of lemma `mid_cond_inv` stating that the conditional distributions preserve the `Intermediate Conditions`.
- **Case Deterministic and Random Loop:** Trivial due to the two-layer induction approach described in Section 3.5.2.

Conclusion Finally, the conclusion in Theorem 4.4 is summarized by `Statistical`, consolidating all prior results. Given a well-formed program c_0, a security parameter s, a precondition ϕ, and two initial states satisfying

$$\phi \models (\sigma, \mu) \wedge \phi \models (\sigma', \mu'),$$

let c and c' denote the original and simplified transformations of c_0, respectively, such that:

$$[\![c]\!](\sigma, \mu) = (\sigma_1, \mu_1) \quad \text{and} \quad [\![c]\!](\sigma', \mu') = (\sigma_2, \mu_2).$$

Suppose we have established the correctness triple:

$$\vdash \{\phi\}\ c\ \{\mathbf{U}_T[\mathsf{Trace}]\},$$

where T is a fixed set of traces, each of polynomial length in s. Under the assumption of statistically secure encryption, we then conclude:

> the statistical distance between $[\![\mathsf{Trace}]\!](\mu_1)$ and $[\![\mathsf{Trace}]\!](\mu_2)$ is negligible in s.

4.4 Computational Obliviousness

Based on the conceptual framework presented in Section 4.1, we define computational obliviousness as follows, inspired by Section 2.3.3. The proof in this section is not formalized in Isabelle/HOL.

To avoid repeatedly defining the same game structure, we introduce the following definition, which generalises IND-CCA–style games. This is a higher-order definition that will later be instantiated in the subsequent definitions and proofs.It reveals the common structure of IND-style games and simplifies the presentation of proofs.

Although this style of definition may be uncommon in cryptography proof, it is quite natural in programming languages and formal verification.

Definition 4.6 (General-IND Game)**.** Given two target algorithms f_0, f_1, a set of additional oracles F, two integers k and n in $\mathsf{poly}(k)$, a polynomial-time bounded (PPT) adversary $\mathcal{A}$, we define $\mathsf{game}(f_0, f_1, F, k, n, \mathcal{A})$ as the following game:

Set up algorithms $\{f_0, f_1\} \cup F$ with security parameter k.

Stage 1:

for $j \in \{1, \ldots, m_1\}$, where m_1 is $\mathsf{poly}(k)$:

$\mathcal{A}$ picks an input for any algorithm in $\{f_0, f_1\} \cup F$ and then gets the result.

Stage 2:

$\mathcal{A}$ picks two valid inputs I_0 and I_1 of the same length n for f_0 and f_1 respectively. A uniform and secret bit $b \in \{0, 1\}$ is chosen, then $f_b(I_b)$ is executed. The results are stored in C and given to $\mathcal{A}$.

Stage 3:

for $j \in \{m_1 + 1, \ldots, m_1 + m_2\}$, where m_2 is $\mathsf{poly}(k)$:

$\mathcal{A}$ picks an input for any algorithm in $\{f_0, f_1\} \cup F$ and then gets the result.

Eventually, $\mathcal{A}$ outputs a bit b'. $\mathcal{A}$ wins the game if and only if $b' = b$.

Definition 4.7 (General Computational Indistinguishability)**.** Let k be a security parameter, n an integer polynomial in k, and F a set of oracles. Algorithms f_0 and f_1 are said to be computationally indistinguishable, denoted $\mathsf{IND}(f_0, f_1, F, k, n)$, if for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}()$ such that the probability that $\mathcal{A}$ wins the game $\mathsf{game}(f_0, f_1, F, k, n, \mathcal{A})$ is bounded by:

$$\mathbb{P}\left[\mathcal{A} \text{ wins}\right] \leq \frac{1}{2} + \mathsf{negl}(k).$$

The definitions have the same structure of the IND-CCA game (Definition 2.9) and the IND-CCA security notion (Definition 2.10), by introducing additional parameters beneficial in subsequent proofs (Section 4.4.1):

- The *General-IND Game* extends traditional indistinguishability definitions by comparing two potentially distinct algorithms with different inputs, rather than a single algorithm with varying inputs. This extension facilitates the connection between encryptions of unknown plaintexts depending on inputs and fixed plaintexts, as illustrated in Section 4.1. It also streamlines the framework by eliminating the

need for multiple similar game definitions, while clearly highlighting the essential distinctions among them. For example, we will show several instance of games in the following subsections with different parameters.

- It includes an optional oracle set F to support auxiliary functionalities, such as decryption oracles, if required.
- The integer parameter n explicitly constrains the input length, enabling the definition to accommodate requirements of oblivious algorithms.

We now specialize the notion of computational indistinguishability to define computational obliviousness. Intuitively, computational obliviousness implies that for any pair of inputs s, s' of equal length n, the outputs produced by the algorithm are computationally indistinguishable under security parameter k.

Definition 4.8 (Computational Obliviousness)**.** Given a security parameter k and integer n polynomial in k, an algorithm c, which is obtained via the original transformation described earlier, is said to be *computationally oblivious* if $\mathsf{IND}(c, c, \emptyset, k, n)$.

In this setting, the indistinguishability game involves two identical algorithms c and asserts that the resulting memory access traces are computationally indistinguishable under different inputs. In Section 4.4.1 and Section 4.4.2, we will consider games involving distinct algorithms, which serve as intermediate steps in the game-based reduction proof.

Finally, we state our main theorem:

Theorem 4.9 (Computational Obliviousness with Encryption)**.** *Let λ be a security parameter, and let c_0 be a PPT program satisfying all conditions specified in Definition 4.2, which employs an encryption scheme that is computationally secure with advantage at most $\mathsf{negl}(\lambda)$. Moreover, we require that the secret information does not affect decryptions of ciphertexts not resulting from previously executed encryptions in the algorithm execution (we explain this requirement below). Consider the programs c and c', which represent the original and simplified transformations of c_0, respectively. If the triple $\vdash \{\phi\}\ c'\ \{\mathbf{U}_T[\mathsf{Trace}]\}$ holds such that the precondition ϕ asserts that the inputs are valid for the program c' and the length of each trace $t \in T$ is bounded by a polynomial in λ, then the program c achieves computational obliviousness, provided its input length is also bounded by a polynomial in λ.*

We require that secret information must not influence the decryption of ciphertexts unless those ciphertexts are the output of previously executed encryption operations. For instance, the program $c(s) = a \leftarrow \mathsf{dec}(s)$ violates this condition when s is secret, as it attempts to decrypt a secret value directly. In contrast, the program $c(s) = a \leftarrow \mathsf{dec}(\mathsf{enc}(s))$ adheres to this requirement, since the ciphertext being decrypted originates from an earlier encryption performed within the same execution of the program.

This restriction is essential because, in the third stage of the IND-CCA game, adversaries are prohibited from querying the decryption oracle on the challenge ciphertext. Without this requirement, the transformed program could directly decrypt ciphertexts dependent on secret information, thereby violating this critical restriction and undermining the integrity of the security definition. In practice, ciphertexts appearing in the initial state generally do not contain secret inputs. Therefore, this additional restriction is typically satisfied in practical scenarios and does not compromise the applicability of our construction.

4.4.1 Proof Structure

We prove Theorem 4.9 by contradiction. Suppose that the program c is *not* computationally oblivious; that is, there exists a PPT adversary $\mathcal{A}$ winning the game $\mathsf{game}(c, c, \emptyset, k, n, \mathcal{A})$ with non-negligible advantage. Under this assumption, our goal is to construct another adversary capable of breaking the IND-CCA security of the underlying encryption scheme. This leads to a contradiction with the security assumption stated in Theorem 4.9. To facilitate this construction, we first introduce the following auxiliary lemma:

Lemma 4.10. *Let k be a security parameter, n an integer polynomially bounded in k, and F a set of auxiliary programs. Consider two programs f_0 and f_1, where f_1 returns outputs following a fixed distribution for any input of length n. If there exists a PPT adversary $\mathcal{A}$ that successfully wins the game $\mathsf{game}(f_0, f_0, F, k, n, \mathcal{A})$ with non-negligible advantage, then there must also exist PPT adversaries $\mathcal{A}_0$ and $\mathcal{A}_1$ for the games $\mathsf{game}(f_0, f_1, F, k, n, \mathcal{A}_0)$ and $\mathsf{game}(f_1, f_0, F, k, n, \mathcal{A}_1)$ respectively, such that at least one adversary wins its respective game with non-negligible advantage.*

We defer the detailed proof of this lemma to Section 4.4.2. Intuitively, this lemma asserts that if a program f_0 is computationally *distinguishable* from itself based on two different inputs, then it must also be computationally *distinguishable* from any program f_1 that returns a fixed output distribution. This property can be viewed as a computational indistinguishability analog of the triangle inequality: if two distributions A and B are computationally *distinguishable* (i.e. have non-negligible computational "distance"), then for any distribution C, the inequality $\mathsf{distance}(A, C) + \mathsf{distance}(B, C) \geq \mathsf{distance}(A, B)$ implies that at least one of $\mathsf{distance}(A, C)$ or $\mathsf{distance}(B, C)$ is non-negligible, and therefore C must be computationally *distinguishable* from either A or B.

Since the definition of the General-IND Game is symmetric, Lemma 4.10 could equivalently be restated with a single game in its conclusion. However, we retain the explicit formulation involving two distinct games to clearly match the subsequent proof steps and explicitly convey the intuitive analogy to the triangle inequality.

We now apply Lemma 4.10 with our transformation (Definition 4.3). Suppose we have a target program c_0 to verify. Its original transformation, simplified transformation and simulated transformation are c, c', c'' respectively. We aim to demonstrate that c (recording the exact trace) and c'' (recording all ciphertext as $\mathsf{enc}(0)$) are computationally *distinguishable* by letting $f_0 = c$ and $f_1 = c''$. Specifically, we argue the existence of a PPT adversary $\mathcal{A}$ who succeeds in the indistinguishability game $\mathsf{game}(c, c'', \emptyset, k, n, \mathcal{A})$ with non-negligible advantage.

We can let $f_1 = c''$ because program c'' consistently outputs a fixed distribution since the sole distinction between c' (which is assumed to be proved that it consistently outputs a fixed distribution in Theorem 4.9) and c'' lies in how ciphertexts are recorded—as either fixed ciphertext $\perp$ or fixed ciphertext $\mathsf{enc}(0)$—both of which yield constant output distributions.

Finally, we complete our proof by reducing the indistinguishability $\mathsf{game}(c, c'', \emptyset, k, n, \mathcal{A})$ to the IND-CCA security game for multiple messages (Definition 2.9):

Lemma 4.11. *Let k be a security parameter and n an integer polynomially bounded in k. Consider two programs c and c'', corresponding respectively to the original and simulated transformations of the program c_0. If there exists a PPT adversary $\mathcal{A}$ capable of winning the indistinguishability game $\mathsf{game}(c, c'', \emptyset, k, n, \mathcal{A})$ with non-negligible*

advantage, then we can construct another PPT adversary $\mathcal{A}'$ that successfully breaks the IND-CCA security for multiple messages as stated in Definition 2.9.

We defer the detailed proof of Lemma 4.11 to Section 4.4.2. Intuitively, the essential distinction between the programs c and c'' lies solely in the recorded ciphertexts: in c, the ciphertexts correspond to encryptions of unknown plaintexts distribution, while in c'', they consistently arise from encrypting a fixed plaintext (namely, $\mathsf{enc}(0)$). This mirrors the differing ciphertext sequences considered in the IND-CCA security game. By establishing this connection, Lemma 4.11 allows us to conclude our proof by contradiction.

4.4.2 Proof Details

Proof of Lemma 4.10. Suppose $\mathcal{A}$ can win $\mathsf{game}(f_0, f_0, F, k, n, \mathcal{A})$ with probability $1/2+g(k)$ where g is not a negligible function, we construct $\mathcal{A}_0$ and $\mathcal{A}_1$ based on $\mathcal{A}$. Without loss of generality, we assume $\mathcal{A}$ will always give output for any C (the challenge message) received in its game even if C is not a possible output of f_0. (If the original $\mathcal{A}$ may fail to output results for some C, we can construct another $\mathcal{A}$ satisfying our assumption by letting it always output 0 when the original $\mathcal{A}$ fails to output.)

For $\mathcal{A}_0$ and its $\mathsf{game}(f_0, f_1, F, k, n, \mathcal{A}_0)$:

At the stage 1 of game (Definition 4.6), $\mathcal{A}_0$ calls $\mathcal{A}$ who may require the executions of $\{f_0\} \cup F$. $\mathcal{A}_0$ follows the requirements from $\mathcal{A}$ and provides outputs.

Afterwards (stage 2), $\mathcal{A}$ outputs I_0 and I_1. $\mathcal{A}_0$ uses I_0 and I_1 for its own game.

A uniform bit $b \in \{0, 1\}$ is chosen, then $f_b(I_b)$ is executed. The results are stored in C and given to $\mathcal{A}_0$. $\mathcal{A}_0$ gives it to $\mathcal{A}$.

At stage 3, $\mathcal{A}$ may continue to require the executions of $\{f_0\} \cup F$. $\mathcal{A}_0$ follows the requirements from $\mathcal{A}$ and provides outputs.

Eventually, $\mathcal{A}$ outputs a bit b'. $\mathcal{A}_0$ uses it as its own answer.

For $\mathcal{A}_1$ and its $\mathsf{game}(f_1, f_0, F, k, n, \mathcal{A}_1)$:

At the stage 1 of game (Definition 4.6), $\mathcal{A}_1$ calls $\mathcal{A}$ who may require the executions of $\{f_0\} \cup F$. $\mathcal{A}_1$ follows the requirements from $\mathcal{A}$ and provides outputs.

Afterwards (stage 2), $\mathcal{A}$ outputs I_0 and I_1. $\mathcal{A}_1$ uses I_0 and I_1 for its own game. In this game, I_0 is for f_1 and I_1 is for f_0, which is reversed from the previous case.

A uniform bit $b \in \{0, 1\}$ is chosen, then $f_{1-b}(I_b)$ is executed. The results are stored in C and given to $\mathcal{A}_1$. $\mathcal{A}_1$ gives it to $\mathcal{A}$.

At stage 3, $\mathcal{A}$ may continue to require the executions of $\{f_0\} \cup F$. $\mathcal{A}_1$ follows the requirements from $\mathcal{A}$ and provides outputs.

Eventually, $\mathcal{A}$ outputs a bit b'. $\mathcal{A}_1$ uses it as its own answer.

Now we analyse their winning probabilities. Let us denote the set of all possible outputs of f_0 and f_1 as S, the probability of $f_0(I_0)$ outputting v as $p(f_0(I_0) = v)$.

Suppose given v, a possible output of f_0, $\mathcal{A}$ has the probability $\mathcal{A}(v)$ for outputting 0 and thus $1 - \mathcal{A}(v)$ for outputting 1, its winning probability can be also written as

$$p_{\mathcal{A}} = 1/2 * \sum_{v \in S} p(f_0(I_0) = v) * \mathcal{A}(v) + p(f_0(I_1) = v) * (1 - \mathcal{A}(v))$$

$\mathcal{A}_0$ will win its game with the probability

$$p_0 = 1/2 * \sum_{v \in S} p(f_0(I_0) = v) * \mathcal{A}(v) + p(f_1(I_1) = v) * (1 - \mathcal{A}(v))$$

$\mathcal{A}_1$ will win its game with the probability

$$p_1 = 1/2 * \sum_{v \in S} p(f_1(I_0) = v) * \mathcal{A}(v) + p(f_0(I_1) = v) * (1 - \mathcal{A}(v))$$

Then we have

$$p_0 + p_1 = p_{\mathcal{A}} + 1/2 * \sum_{v \in S} p(f_1(I_0) = v) * \mathcal{A}(v) + p(f_1(I_1) = v) * (1 - \mathcal{A}(v))$$

By the assumption that f_1 always outputs the same distribution, we have

$$\forall v.\ p(f_1(I_0) = v) = p(f_1(I_1) = v)$$

Thus

$$\begin{aligned} p_0 + p_1 =& p_{\mathcal{A}} + 1/2 * \sum_{v \in S} p(f_1(I_0) = v) * \mathcal{A}(v) + p(f_1(I_0) = v) * (1 - \mathcal{A}(v)) \\ =& p_{\mathcal{A}} + 1/2 * \sum_{v \in S} p(f_1(I_0) = v) \\ =& p_{\mathcal{A}} + 1/2 \end{aligned}$$

Finally, we have $p_0 + p_1 = 1 + g(k)$ by assumptions that $p_{\mathcal{A}}$ is equal to $1/2$ plus some non-negligible function, which implies that at least one of p_0 and p_1 is greater than $1/2 + g(k)/2$. The desired conclusion follows. □

Proof of Lemma 4.11. Assume the adversary $\mathcal{A}$ wins $\mathsf{game}(c, c'', \emptyset, k, n, \mathcal{A})$ with probability $1/2 + g(k)$, where $g(k)$ is a non-negligible function. We construct an adversary $\mathcal{A}'$ against the IND-CCA security game that leverages $\mathcal{A}$ as a subroutine. The adversary $\mathcal{A}'$ has access to encryption and decryption oracles $\mathsf{enc}()$ and $\mathsf{dec}()$.

Initially, $\mathcal{A}'$ runs the program c once (with any valid input), recording the inputs and outputs of all calls to the decryption oracle $\mathsf{dec}()$ for later reference.

At stage 1, whenever $\mathcal{A}$ requests to execute either c or c'' on chosen inputs, $\mathcal{A}'$ simulates these executions using the provided encryption and decryption oracles.

Eventually, at stage 2, $\mathcal{A}$ outputs two message sets S_0 and S_1. At this point, $\mathcal{A}'$ executes $c(S_0)$ and collects all plaintexts whose ciphertexts appear in the Trace into a list L_1. Then, $\mathcal{A}'$ creates another list L_0 of equal length, filled entirely with zeros. These two lists, L_0 and L_1, serve as the message sequences submitted in the multi-message IND-CCA challenge.

A uniform random bit $b \in \{0, 1\}$ is chosen by the challenger, and $\mathsf{enc}(L_b)$ is executed. The ciphertext results are stored in list L and provided to $\mathcal{A}'$. Next, $\mathcal{A}'$ substitutes the ciphertexts from L into the original execution result E (obtained from $c(S_0)$), forming a new execution trace E' that is passed to the adversary $\mathcal{A}$.

At stage 3, whenever $\mathcal{A}$ again requests execution of c or c'', $\mathcal{A}'$ simulates these executions using only the encryption oracle $\mathsf{enc}()$. Since all ciphertext-plaintext relationships have been recorded or can be inferred from prior encryptions in the current execution, $\mathcal{A}'$ can simulate these executions without querying the decryption oracle, assuming the secret information does not affect decryptions of ciphertexts not resulting from previously executed encryptions.

Finally, $\mathcal{A}$ outputs a bit b', which $\mathcal{A}'$ forwards as its own guess.

If $b = 0$, the distribution of E' matches exactly the distribution of E (albeit with re-executed encryptions). If $b = 1$, the distribution of E' matches the distribution of $c''(S_1)$ due to the definition of the transformation (Definition 4.3), where all ciphertexts recorded in the trace originate from encryptions of zeros.

Thus, the distribution of E' in this constructed game precisely corresponds to the challenge ciphertext distribution in $\mathsf{game}(c, c'', \emptyset, k, n, \mathcal{A})$. Consequently, the success probability of $\mathcal{A}'$ in the constructed game equals the success probability of $\mathcal{A}$ in the original indistinguishability game, completing the proof. □

4.5 Summary

This chapter introduces a formal verification framework that precisely defines the security guarantees of oblivious algorithms incorporating encryption. The verification process is systematic: it involves transforming the program and applying the program logic developed in Chapter 3. We establish the soundness of this approach, providing a proof for statistical security mechanized in Isabelle/HOL and a proof for computational security carried out manually.

We illustrated the method using a toy example in Section 4.1, and in the following chapter, we apply it to several practical probabilistic oblivious algorithms that utilize encryption.

Chapter 5

Case Studies

In Section 4.1, we presented a motivating algorithm along with its verification process, illustrating the use of our transformation and program logic as introduced in Chapter 4 and Chapter 3. To demonstrate its practical utility on real-world examples, we further applied this framework to verify the obliviousness of four non-trivial algorithms: Oblivious Sampling [Sasy and Ohrimenko, 2019], the Melbourne Shuffle [Ohrimenko et al., 2014], Path ORAM [Stefanov et al., 2018], and Path Oblivious Heap [Shi, 2019]. These case studies were performed at the pen-and-paper level. Extending them to a mechanized formalization would be valuable future work, but it is nontrivial, as our framework currently employs abstract data types that are still far from the level of detail required for implementing the full algorithms.

Each of these case studies incorporates encryption and is transformed using our framework from Chapter 4. Moreover, the algorithms exhibit distinct structural characteristics and require different verification strategies, as detailed below.

To our knowledge, Oblivious Sampling, the Melbourne Shuffle, and Path Oblivious Heap have never been formally verified as each requires the combination of features that our approach uniquely supports. Path ORAM has received some formal verification [Sahai et al., 2020, Leung et al., 2023] (see Section 2.4.2.2) and also comes with an informal but rigorous proof of security [Stefanov et al., 2018]. We verified it following their overall strategy to show that our logic can indeed encode existing rigorous security arguments (see Section 5.2 for details).

Sampling serves as a fundamental building block in various domains, including differential privacy, oblivious data analysis, and machine learning [Abadi et al., 2016, McMahan et al., 2018, Yu et al., 2019]. One of our case studies is the oblivious sampling algorithm (Section 5.1) introduced by Sasy and Ohrimenko [2019], which performs sampling over a dataset while ensuring that the resulting memory access pattern is uniformly distributed. The algorithm involves randomised and secret-dependent control flows, such as loops and conditionals, as well as dynamic random operations like shuffling a truly probabilistic array. Consequently, the combination of classical and probabilistic reasoning in our program logic is crucial for verifying the security guarantees of this algorithm.

Path ORAM [Stefanov et al., 2018] is a seminal oblivious RAM algorithm with practical efficiency, providing general-purpose oblivious storage. Path oblivious heap is inspired by Path ORAM and the two share the same idea: using a binary tree with a random and virtual location table to store secret data, where the mappings between each physical and virtual location are always independent of each other and of the memory access pattern. Thus probabilistic independence is crucial to express and prove these algorithms' key invariants. We present the detailed verification and the distinct challenges involved in Section 5.2 and Section 5.3, respectively.

The Melbourne Shuffle [Ohrimenko et al., 2014] is an effective [Ohrimenko et al., 2014, Table 2] oblivious shuffling algorithm used in cloud storage and also as a basic building block for other higher-level algorithms (e.g. oblivious sampling [Sasy and Ohrimenko, 2019]). Its operation is non-trivial, including rearranging array elements with dummy values and other complexities. Its verification (Section 5.4) employs much classical reasoning because, while it is probabilistic, its memory access pattern is deterministic (absent failure).

Overall, the verification process typically follows these steps:

- **Program Adaptation:** Modify the target algorithm while preserving its original semantics to conform to the syntax and semantics of our programming language introduced in Chapter 3 and to the transformation requirements (Definition 4.2).
- **Transformation Application:** Apply the transformation (Definition 4.3) to obtain a simplified transformation of the program (i.e. choose the 2nd option for

step 4), which can then be verified using our program logic. A concrete example of this procedure was provided in Section 4.1.

- **Trace Specification:** Define functions that characterize the memory access traces recorded in Trace. These traces generally depend on one or more random variables and can be abstractly represented as functions of those variables. For example, the trace in Section 4.1 depends on a random variable r, and is therefore described by a function of r. In more complex cases presented in this chapter, the trace is typically modeled as the concatenation of several such functions over multiple random variables.

- **Logical Verification:** Use our program logic (Chapter 3) to show that the algorithm's actual trace Trace indeed conforms to the trace specification defined over the random variables via appropriate assertions $\mathsf{Ct}()$. Furthermore, demonstrate that these random variables are mutually independent so that the concatenated trace achieves uniform distribution, as established by Proposition 3.2. This step may involve a variety of techniques and nontrivial reasoning strategies, which will be elaborated on later in this chapter.

- **Obliviousness Conclusion:** Finally, establish the obliviousness of the target algorithm by invoking Theorems 4.9 and 4.4.

An earlier version of this work appeared in Formal Methods 2024 [Yan et al., 2025]. At that time, it applied the program logic of Chapter 3 to prove the properties assuming perfect encryption. In this chapter, we extend our previous results by incorporating encryption, along with its corresponding transformations and verification, based on the framework introduced in Chapter 4. These extensions enabled our previous verification using Chapter 3's program logic to be used to prove statistical and computational obliviousness of the target algorithm relative to the strength of the actually used encryption scheme, without altering the overall structure or the core ideas of the original verification.

```
1   D ← oblshuffle(D);
2   SWO ←$ U_{X(n,m)} ;
3   S ← [] ; j ← 1 ; l ← 1 ;
4   e ← D[1];
5   e_next ← D[1];
6   while_R l < n + 1 do
7     i ← 1;
8     while_R i < k + 1 do
9       if_R SWO[i][j] then
10        S.append(enc((e, i)));
11        l ← l + 1;
12        e_next ← D[l];
13      i ← i + 1;
14    e ← e_next;
15    j ← j + 1;
16  S ← oblshuffle(S);
17  s ← [[], [], · · · , []]; //k empty arrays
18  p ← 1;
19  while_R p < m + 1 do
20    (e, i) ← dec(S[p]);
21    s[i].append(enc(e));
22    p ← p + 1;
```

FIGURE 5.1: Sampling Algorithm

5.1 Oblivious Sampling

Oblivious random sampling [Sasy and Ohrimenko, 2019] plays a crucial role in oblivious training algorithms for machine learning, where it is commonly used to randomly select mini-batches.

The algorithm takes a secret database (an array) D of size n as input, and will output several arrays $(s[0], s[1], \ldots, s[k])$ where each contains m pieces of independently sampled data from D. Moreover, $n = m \cdot k$. The memory access pattern of the database D, temporary array S and the returned arrays $s[\ldots]$ are observable to attackers.

We present the rewritten algorithm in Fig. 5.1, modified to comply with the transformation requirements outlined in Section 4.2. The primary change is the replacement of the direct copy operation at lines 20,21 with an additional re-encryption step (dec() and enc() in these two lines).

Additional changes include translating the `for` loop into a `while` loop and separating

memory accesses to public locations that were originally combined within a single command. These adjustments, as required by Definition 4.2, preserve the original algorithm's semantics.

Following the original algorithm paper [Sasy and Ohrimenko, 2019], the array SWO is a two-dimensional array of booleans of size $k \times n$, which is randomly chosen from the set $X(n, m)$ which contains all such arrays such that $\forall i \in \{1 \cdots k\}$, the number of true in $\mathsf{SWO}[i][1 \cdots n]$ is m. Unlike other case studies (where arrays are indexed from 0), the arrays in Fig. 5.2 are indexed from 1 [Sasy and Ohrimenko, 2019].

This algorithm consists of three main stages:

- Lines 1–5 perform the initialization of several variables. The input database D, represented as an array, is then obliviously shuffled and implicitly re-encrypted. This process is modeled as a uniform random choice over the set of all permutations of the array, denoted $\mathsf{Perm}(D)$. This re-encryption is essential—without it, an adversary could infer which elements were sampled by observing unchanged ciphertexts. The variable SWO (as the secret) is initialized by randomly selecting one of its possible values. The intermediate array S is prepared to store sampled data, while the remaining variables are set up to facilitate the construction of S in the second stage.

- Lines 6–15 construct the intermediate array S using a nested loop. Each entry in S consists of a sampled data element e paired with a target index i, indicating that e should appear in the ith sample $s[i]$. The inner loop is randomised, with the number of iterations depending on the secret inputs.

- Lines 16–22 perform an oblivious shuffle of the intermediate array S to ensure a random memory access pattern. Afterwards, each data element e in S is decrypted and written to the target location $s[i]$ according to its associated index i, producing the final sampling results. The memory access pattern of this stage is uniformly distributed.

```
1   Trace ← [];
2   D ←$ U_{Perm(D)}; Trace ← Trace + oblSF("D", n);
    SWO ←$ U_{X(n,m)} ;
    S ← [] ; j ← 1 ; l ← 1 ;
3   e ← D[1]; Trace ← Trace + ("Read", "D", 1);
4   e_next ← D[1]; Trace ← Trace + ("Read", "D", 1);
    while_R l < n + 1 do
      i ← 1;
      while_R i < k + 1 do
        if_R SWO[i][j] then
          S.append(enc((e, i)));
5         Trace ← Trace + ("Write", "S", size(S), ⊥);
          l ← l + 1;
          e_next ← D[l];
6         Trace ← Trace + ("Read", "D", l);
        i ← i + 1;
      e ← e_next;
      j ← j + 1;
7   S ←$ U_{Perm(S)}; Trace ← Trace + oblSF("S", size(S));
    s ← [[], [], ··· , []]; //k empty arrays
    p ← 1;
    while_R p < m + 1 do
8     (e, i) ← dec(S[p]); Trace ← Trace + ("Read", "S", p);
      s[i].append(enc(e));
9     Trace ← Trace + ("Write", "s", i, size(s[i]), ⊥);
      p ← p + 1;
```

FIGURE 5.2: Transformed Sampling Algorithm

5.1.1 Transformation

We then apply the transformation defined in Definition 4.3 to get the simplified transformation, shown in Fig. 5.2. However, the following adjustments are made to accommodate specific language features and modeling considerations:

- For the function call oblshuffle(), which performs an oblivious shuffle of the input array, we model its behavior as a uniform random choice over all possible permutations. The corresponding memory access pattern is specified as oblSF(), which depends only on the array name and its length. If oblshuffle() is probabilistic, our verification remains sound as long as the selected permutation is independent of secret values (i.e., the initial and final permutations), which it is for oblivious shuffling algorithms.

- For array append operations, we model each append as a write to the last position in the array. The index of this write corresponds to the new size of the array.
- For array accesses, we record both the array name and the accessed index, which together determine the memory access address. In the case of nested arrays, multiple indices may be recorded, as illustrated at line 9.

5.1.2 Verification

The verification of the sampling algorithm has three main parts following the corresponding three stages introduced in Section 5.1.

- The first stage initialize several variables, which is staightforward to verify.
- For the second stage, its memory access trace is deterministic, and so we prove that as a certainty via $\mathsf{Ct}(\cdot)$ reasoning. At this point, we have that the trace is a deterministic value (captured by the predicate $\mathsf{inv}(\mathsf{Trace})$), plus some certain information about $\mathsf{snd}(S)$, which is essential for the verification that follows. Suppose S is an array of tuples, then we let $\mathsf{snd}(S)$ represent the array of all the second elements of the tuples in the array S.
- The last part of the verification covers the remaining code, which shuffles S and then produces a memory access pattern which is a deterministic function of (the shuffled) $\mathsf{snd}(S)$. We thus prove obliviousness by proving that the overall memory access trace is uniformly distributed (and thus independent of secrets).

Then we introduce the detailed reasoning of the proof, sketched in Fig. 5.3. The reasoning is as follows.

Let $A[i..j]$ denote the sub-array from $A[i]$ to $A[j]$ and $\mathsf{Count}(v, A)$ represent the number of occurrences of v in the array A.

Let $\mathsf{inv}(\mathsf{Trace})$ be a predicate on traces that holds if and only if:

$$\begin{aligned}&\mathsf{Trace}[1..3] = [\mathsf{oblSF}(\text{“}D\text{”}, n), (\text{“}\mathsf{Read}\text{”}, \text{“}D\text{”}, 1), (\text{“}\mathsf{Read}\text{”}, \text{“}D\text{”}, 1)] \wedge \\ &\quad (\forall x,\ 3 < x \leq \mathsf{size}(\mathsf{Trace}) \implies (x\%2 = 0 \implies \mathsf{Trace}[x] = (\text{“}\mathsf{Write}\text{”}, \text{“}S\text{”}, (x-2)/2), \bot) \\ &\qquad\qquad \wedge (x\%2 = 1 \implies \mathsf{Trace}[x] = (\text{“}\mathsf{Read}\text{”}, \text{“}D\text{”}, (x-1)/2)))\end{aligned}$$

$\{\mathsf{Ct}(n = m \times k)\}$
$\mathsf{Trace} \leftarrow []$;
$D \leftarrow_{\$} \mathbf{U}_{\mathsf{Perm}(D)}$; $\mathsf{Trace} \leftarrow \mathsf{Trace} + \mathsf{oblSF}(\text{“}D\text{”}, n)$;
$\mathsf{SWO} \leftarrow_{\$} \mathbf{U}_{X(n,m)}$;
$S \leftarrow []$; $j \leftarrow 1$; $l \leftarrow 1$;
$e \leftarrow D[1]$; $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Read”}, \text{“}D\text{”}, 1)$;
$e_{\mathsf{next}} \leftarrow D[1]$; $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Read”}, \text{“}D\text{”}, 1)$;
$\{\mathsf{Ct}(\mathsf{Trace} = [\mathsf{oblSF}(\text{“}D\text{”}, n), (\text{“Read, “}D\text{”}, 1), (\text{“Read”}, \text{“}D\text{”}, 1)] \wedge n = m \times k \wedge S = [] \wedge j = l = 1 \wedge$
$e = e_{\mathsf{next}} = D[1] \wedge D \in \mathsf{Perm}(D) \wedge \mathsf{SWO} \in X(n, m))\}$
Start Unif-Idp rule on SWO and Trace
while$_R$ $l < n + 1$ **do**
Loop Invariant:
$\{\mathsf{Ct}(n = m \times k \wedge D \in \mathsf{Perm}(D) \wedge \mathsf{SWO} \in X(n, m) \wedge l = \mathsf{size}(S) + 1 = (\mathsf{size}(\mathsf{Trace}) - 1)/2 =$
$\mathsf{Count}(\mathsf{true}, \mathsf{SWO}[1..k][1..j-1]) + 1 \wedge \mathsf{inv}(\mathsf{Trace}) \wedge \mathsf{size}(\mathsf{Trace})\%2 = 1 \wedge l \leq n + 1 \wedge$
$(\forall x.\ 0 < x \leq k \implies \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[x][1..j-1]) = \mathsf{Count}(x, \mathsf{snd}(S))))\}$
$i \leftarrow 1$;
while$_R$ $i < k + 1$ **do**
Loop Invariant:
$\{\mathsf{Ct}(\mathsf{inv}(\mathsf{Trace}) \wedge n = m \times k \wedge D \in \mathsf{Perm}(D) \wedge \mathsf{SWO} \in X(n, m) \wedge i \leq k + 1 \wedge l = \mathsf{size}(S) + 1 =$
$(\mathsf{size}(\mathsf{Trace}) - 1)/2 = \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[1..k][1..j-1]) + \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[1..i-1][j]) + 1 \wedge$
$(\forall x.\ 0 < x < i \implies \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[x][1..j]) = \mathsf{Count}(x, \mathsf{snd}(S))) \wedge$
$(\forall x.\ i \leq x \leq k \implies \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[x][1..j-1]) = \mathsf{Count}(x, \mathsf{snd}(S))))\}$
if$_R$ $\mathsf{SWO}[i][j]$ **then**
S.append(enc((e, i)));
$\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Write”}, \text{“}S\text{”}, \mathsf{size}(S), \perp)$;
$l \leftarrow l + 1$;
$e_{\mathsf{next}} \leftarrow D[l]$;
$\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Read”}, \text{“}D\text{”}, l)$;
$i \leftarrow i + 1$;
$e \leftarrow e_{\mathsf{next}}$;
$j \leftarrow j + 1$;
$\{\mathsf{Ct}(\mathsf{size}(\mathsf{Trace}) = 2n + 3 \wedge \mathsf{inv}(\mathsf{Trace}) \wedge \mathsf{size}(S) = n \wedge (\forall x.\ 0 < x \leq k \implies \mathsf{Count}(x, \mathsf{snd}(S)) = m))\}$
$S \leftarrow_{\$} \mathbf{U}_{\mathsf{Perm}(S)}$;
$\{\mathsf{Ct}(\mathsf{size}(\mathsf{Trace}) = 2n + 3 \wedge \mathsf{inv}(\mathsf{Trace}) \wedge \mathsf{size}(S) = n \wedge \mathbf{U}_{\mathsf{Perm}(Se)}[\mathsf{snd}(S)] \wedge$
$(\forall x.\ 0 < x \leq k \implies \mathsf{Count}(x, \mathsf{snd}(S)) = m)\}$
$\mathsf{Trace} \leftarrow \mathsf{Trace} + \mathsf{oblSF}(\text{“}S\text{”}, \mathsf{size}(S))$;
$s \leftarrow [[], [], \cdots, []]$; //k of empty arrays
$p \leftarrow 1$;
$\{\mathsf{Ct}(\mathsf{inv2}(\mathsf{Trace}, S) \wedge \mathsf{size}(S) = n \wedge p = \mathsf{size}(\mathsf{Trace})/2 - n - 1 \wedge$
$(\forall x.\ 0 < x \leq k \implies \mathsf{Count}(x, \mathsf{snd}(S)) = m)) \wedge \mathbf{U}_{\mathsf{Perm}(Se)}[\mathsf{snd}(S)]\}$
Start Const rule with $\mathbf{U}_{\mathsf{Perm}(Se)}[\mathsf{snd}(S)]$
$\{\mathsf{Ct}(\mathsf{inv2}(\mathsf{Trace}, S) \wedge \mathsf{size}(S) = n \wedge p = \mathsf{size}(\mathsf{Trace})/2 - n - 1 \wedge$
$(\forall x.\ 0 < x \leq k \implies \mathsf{Count}(x, \mathsf{snd}(S)) = m) \wedge$
$(\forall y.\ 0 < y \leq k \implies \mathsf{size}(s[i]) = \mathsf{Count}(i, \mathsf{snd}(S)[1...p-1])))\}$ (Invariant for the last loop)
while$_R$ $p < m + 1$ **do**
$(e, i) \leftarrow \mathsf{dec}(S[p])$; $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Read”}, \text{“}S\text{”}, p)$;
$s[i]$.append(enc(e));
$\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Write”}, \text{“}s\text{”}, i, \mathsf{size}(s[i]), \perp)$;
$p \leftarrow p + 1$;
$\{\mathsf{Ct}(\mathsf{inv2}(\mathsf{Trace}, S) \wedge \mathsf{size}(S) = n \wedge p = \mathsf{size}(\mathsf{Trace})/2 - n - 1 \wedge$
$(\forall x.\ 0 < x \leq k \implies \mathsf{Count}(x, \mathsf{snd}(S)) = m) \wedge p > \mathsf{size}(S))\}$
$\{\mathsf{Ct}(\mathsf{Trace} = f(\mathsf{snd}(S)) \wedge (f \text{ is bijective}))\}$
End Const rule
$\{\mathsf{Ct}(\mathsf{Trace} = f(\mathsf{snd}(S)) \wedge (f \text{ is bijective})) \wedge \mathbf{U}_{\mathsf{Perm}(Se)}[\mathsf{snd}(S)]\}$
$\{\mathbf{U}_{f(Se)}[\mathsf{Trace}]\}$
End Unif-Idp rule
$\{\mathbf{U}_{f(Se)}[\mathsf{Trace}] * \mathbf{D}(\mathsf{SWO})\}$

FIGURE 5.3: Verification of Sampling Algorithm

This invariant characterizes the values in Trace at the beginning of each iteration of the nested loop. The first line specifies the initial three elements of Trace, which are determined by the code executed prior to entering the loops. The remaining part of the invariant describes the memory access pattern induced by the loop structure: each iteration consistently performs a write to the array S, followed by a read from the database array D. (Note that the first read from D occurs before the loop begins.)

Let $\mathsf{inv2}(\mathsf{Trace}, S)$ be a predicate that holds if and only if

$$\begin{aligned}&\mathsf{inv}(\mathsf{Trace}[1..2n+3], S) \wedge \mathsf{Trace}[2n+4] = \mathsf{oblSF}(\text{“}S\text{”}, \mathsf{size}(S)) \wedge \\ &\quad (\forall x, j.\ (2n+4 < x \leq \mathsf{size}(\mathsf{Trace}) \wedge j = (x-(2n+3))/2) \implies \\ &\qquad (x\%2 = 0 \implies \mathsf{Trace}[x] = (\text{“}\mathsf{Write}\text{”}, \text{“}s\text{”}, \mathsf{snd}(S)[j], \mathsf{Count}(\mathsf{snd}(S)[j], \mathsf{snd}(S)[1..j]), \bot)) \\ &\qquad\qquad \wedge (x\%2 = 1 \implies \mathsf{Trace}[x] = (\text{“}\mathsf{Read}\text{”}, \text{“}S\text{”}, j))\end{aligned}$$

The invariant $\mathsf{inv2}()$ specifies the value of Trace within the loop invariant of the final loop. It includes $\mathsf{inv}()$ as a subcomponent, since the nested loop executes prior to this one. In addition, it contains two lines of classical logical assertions that describe the memory access pattern induced by the final loop.

At the beginning, we have $\{\mathsf{Ct}(n = m \times k)\}$, required by the algorithm specification.

Then, just before the first loop, we can apply RSample rule, RAssign rule (Fig. 3.3), Const rule, and Weak rule (Fig. 2.6) to get:

$$\begin{aligned}&\{\mathsf{Ct}(\mathsf{Trace} = [\mathsf{oblSF}(\text{“}D\text{”}, n), (\text{“}\mathsf{Read}, \text{“}D\text{”}, 1), (\text{“}\mathsf{Read}\text{”}, \text{“}D\text{”}, 1)] \wedge n = m \times k \wedge S = [] \wedge \\ &\quad j = l = 1 \wedge e = e_{\mathsf{next}} = D[1] \wedge D \in \mathsf{Perm}(D) \wedge \mathsf{SWO} \in X(n, m))\}\end{aligned}$$

Then, for the first loop, we use the loop invariant:

$$\begin{aligned}&\{\mathsf{Ct}(n = m \times k \wedge D \in \mathsf{Perm}(D) \wedge \mathsf{SWO} \in X(n, m) \wedge l = \mathsf{size}(S) + 1 = (\mathsf{size}(\mathsf{Trace}) - 1)/2 = \\ &\quad \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[1..k][1..j-1]) + 1 \wedge \mathsf{inv}(\mathsf{Trace}) \wedge \mathsf{size}(\mathsf{Trace})\%2 = 1) \wedge \\ &\quad (\forall x.\ 0 < x \leq k \implies \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[x][1..j-1]) = \mathsf{Count}(x, \mathsf{snd}(S)))\}\end{aligned}$$

For the inner loop, we use another loop invariant:

$$\begin{aligned}&\{\mathsf{Ct}(n = m \times k \wedge D \in \mathsf{Perm}(D) \wedge \mathsf{SWO} \in X(n, m) \wedge l = \mathsf{size}(S) + 1 = (\mathsf{size}(\mathsf{Trace}) - 1)/2 = \\ &\quad \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[1..k][1..j-1]) + \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[1..i-1][j]) + 1 \wedge \mathsf{inv}(\mathsf{Trace}) \wedge \\ &\quad (\forall x.\ 0 < x < i \implies \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[x][1..j]) = \mathsf{Count}(x, \mathsf{snd}(S))) \wedge \\ &\quad (\forall x.\ i \leq x \leq k \implies \mathsf{Count}(\mathsf{true}, \mathsf{SWO}[x][1..j-1]) = \mathsf{Count}(x, \mathsf{snd}(S))))\}\end{aligned}$$

Both loop invariants can be proved by applying the random assignment rule, weak rule, and random if rule. After the nested loop, we know the first loop invariant holds and $l > n$. By the weak rule, we obtain:

$\{\mathsf{Ct}(\mathsf{size}(\mathsf{Trace}) = 2n + 3 \wedge \mathsf{inv}(\mathsf{Trace}) \wedge \mathsf{size}(S) = n \wedge$

$(\forall x.\ 0 < x \leq k \implies \mathsf{Count}(x, \mathsf{snd}(S)) = m))\}$

Note that $\mathsf{inv}(\mathsf{Trace})$ is satisfied by at most one Trace of a particular length. So we know that Trace has a deterministic value at this point.

After the random choice following the loop, let Se be an array containing m occurrences of every number between 1 and k. Then we have that $\mathbf{U}_{\mathsf{Perm}(Se)}[\mathsf{snd}(S)]$ in addition to the previous assertion, by the random sample rule. Then we apply the random assignment rule and weak rule several times to obtain the following assertion before the last loop:

$\{\mathsf{Ct}(\mathsf{inv2}(\mathsf{Trace}, S) \wedge \mathsf{size}(S) = n \wedge p = \mathsf{size}(\mathsf{Trace})/2 - n - 1 \wedge$

$(\forall x.\ 0 < x \leq k \implies \mathsf{Count}(x, \mathsf{snd}(S)) = m) \wedge$

$(\forall y.\ 0 < y \leq k \implies \mathsf{size}(s[i]) = \mathsf{Count}(i, \mathsf{snd}(S)[1...p-1])))\}$

Then we use the Const rule to add the Uniform assertion between the loop, and use the $\mathsf{Ct}(\ldots)$ part of the above assertion as the last loop's invariant, which can be proved by applying the random loop, random if, random assignment, and weak rules. Finally, we obtain the above assertion and $p > \mathsf{size}(S)$. This information implies that the value of Trace is a bijective function of $\mathsf{snd}(S)$, which means Trace also satisfies a uniform distribution. We can also apply the Unif-Ind rule to show Trace is independent of SWO. Note that since Se is independent of the original database contents D, we have trivially that Trace is independent of D.

5.1.3 Conclusion

Following the verification framework introduced in Chapter 4 and the program logic formalized in Chapter 3, we have verified the obliviousness of the sampling algorithm presented in this section.

We began by adapting the original algorithm to conform to our programming language and transformation requirements (Fig. 5.1). We then applied the transformation procedure to obtain a simplified version of the program (Section 5.1.1, Fig. 5.2), which was subsequently verified using our program logic framework (Section 5.1.2, Fig. 5.3).

Finally, we conclude that the algorithm satisfies statistical or computational obliviousness, depending on the underlying encryption scheme, as established in Theorem 4.4 and Theorem 4.9.

All case studies in this chapter follow a similar verification process and yield analogous security guarantees.

5.2 Path ORAM

Path ORAM [Stefanov et al., 2018] is a (probabilistic) oblivious RAM algorithm. It allows a client to conceal its access pattern to some remote storage. When the client wants to read/write something from/to the remote storage, it calls the function Access$(op, a, data*)$ where op is the type of access being performed (read or write), a is the *virtual* (i.e. as seen by the client) storage location being accessed, and $data^*$ is either None (in the case of a read access) or is a value to be written (in the case of a write access). The *physical* location of a in the remote server is stored in the global array Q (originally called *position* in [Stefanov et al., 2018]). This location changes following the execution of the algorithm, to hide the (subsequent) access patterns of future executions of the algorithm.

The original presentation of the algorithm included a perfectly oblivious approximation [Stefanov et al., 2018, Figure 1], and bounded the statistical distance between it and the practical version of the algorithm [Stefanov et al., 2018, Section 5]. Thus our focus is to prove the perfectly oblivious version is indeed perfectly oblivious.

Our goal is to prove that any two sequences of operations of the same length produce indistinguishable memory access patterns, in the sense that both are uniformly distributed over the same set of possibilities. A sequence of operations corresponds to a series of calls to the Access function, e.g. a sequence could be [(write, a, 1), (read, a, None)], and would correspond to two calls to the access function: namely $\mathsf{Access}(\text{write}, a, 1)$ followed by $\mathsf{Access}(\text{read}, a, \text{None})$. The distribution of the memory access pattern produced by this sequence should be identical to any other sequence of operations of length 2.

The simplified transformation of the path ORAM algorithm appears in Fig. 5.4 where the ghost codes are added by transformation to capture the observable memory access

pattern. This figure follows the original [Stefanov et al., 2018, Figure 1], renaming the original *position* array to Q for brevity. We also add an additional ghost variable T' to record the initial value of Trace, which will be useful in the verification, because the initial value of it is unknown as this function will be called many times to execute many operations. The only two observable commands are the function calls of WriteBucket() and ReadBucket(), where their parameters x and l determine the physical locations that are accessed.

As described in [Stefanov et al., 2018, Section 3.5], the function WriteBucket() writes data to a specified bucket with re-encryption, while ReadBucket() reads data from a specified bucket with decryption. Both operations induce deterministic memory access patterns (excluding the encryption producing random ciphertext) based on the parameters x and l, which determine the bucket location. While the original paper outlines the high-level behavior of these functions, it does not provide their concrete implementations. We present our own implementation and formal verification of these subroutines in Section 5.2.2. In the verification of the main function Access(), the memory access patterns induced by these operations will be abstractly represented as ("WriteBucket()", x, l) and "ReadBucket()", x, l), respectively. We further establish a mapping from these abstract representations to the concrete traces produced by our implementations (see Section 5.2.2), thereby showing that the abstractions faithfully capture the essential behavior of the subroutines.

5.2.1 Verification

We prove perfect obliviousness by proving that ACCESS maintains the following security invariant on the observable memory access trace Trace: given any initial memory access pattern Trace satisfying a fixed uniform distribution, the resulting access pattern after calling ACCESS (an extension of Trace) still satisfies a fixed uniform distribution. This security definition depends on the main invariant introduced in Section 5.2.

We cannot assume that the initial memory access pattern is empty, because assuming so will only prove the first call does not leak information. It is possible that an algorithm does not leak information in the first call but does leak information in the following calls.

function ACCESS$(op, a, data^*, \mathsf{Trace})$
  $T' \leftarrow \mathsf{Trace}$
  $x \leftarrow Q[a]$
  $Q[a] \leftarrow_{\$} \mathbf{U}_{\{0..(2^L-1)\}}$
  $l \leftarrow 0$
  while$_R$ $l \leq L$ **do**
    $S \leftarrow S \cup \mathsf{ReadBucket}(P(x,l))$
    $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“}\mathsf{ReadBucket}()\text{”}, x, l)$
    $l \leftarrow l + 1$
  $data \leftarrow \mathsf{find}(a, S)$
  if$_D$ $op = \mathsf{Write}$ **then**
    $S \leftarrow (S - \{(a, data)\}) \cup \{(a, data^*)\}$
  $l \leftarrow L$
  while$_R$ $l \geq 0$ **do**
    $S' \leftarrow \{(a', data') \in S : P(x,l) = P(Q[a'], l)\}$
    $S' \leftarrow \mathsf{select}(Z, S')$
    $S \leftarrow S - S'$
    $\mathsf{WriteBucket}(P(x,l), S')$
    $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“}\mathsf{WriteBucket}()\text{”}, x, l)$
    $l \leftarrow l - 1$
end function

FIGURE 5.4: Transformed path ORAM

We prove maintenance of the security invariant by proving that the algorithm maintains its key implementation invariant [Stefanov et al., 2018, Section 3], which we refer to in this paper as the *Main Invariant*: each block is mapped to a uniformly random leaf bucket in the tree (whose height is L), i.e. every value in the array Q satisfies the uniform distribution on $\{0..(2^L - 1)\}$ and is independent of the others. The Main Invariant is encoded as an assertion in our logic using the $\mathbf{U}.[\cdot]$ and $*$ operators (see below).

Then the verification proceeds as follows. Let $W = \{0...(2^L - 1)\}$ and $n = \mathsf{size}(Q)$ which also means the number of (virtual) locations that can be accessed. As shown in Fig. 5.5, we start with our desired security invariant. It includes the Main Invariant which is $\{(\mathbf{U}_W[Q[0]] * \mathbf{U}_W[Q[1]] * ... * \mathbf{U}_W[Q[n]])\}$, plus the fact that there exists a fixed set Y where Trace satisfies the uniform distribution on it (denoted $\mathbf{U}_Y[\mathsf{Trace}]$), independently to the Main Invariant.

The verification of the first three lines of code is performed by unfolding the Main Invariant, and using the RASSIGN rule (Fig. 3.3), FRAME rule and WEAK rule (Fig. 2.6), then refolding the Main Invariant. Then we use the CONST rule (Fig. 2.6) to carry all the information except $\mathsf{Ct}(\mathsf{Trace} = T')$ to the end of this function as these facts

function ACCESS$(op, a, data^*, \mathsf{Trace})$
$\{\text{Main Invariant} * \mathbf{U}_Y[\mathsf{Trace}]\}$
  $T' \leftarrow \mathsf{Trace}$
$\{\text{Main Invariant} * \mathbf{U}_Y[\mathsf{Trace}] \wedge \mathsf{Ct}(\mathsf{Trace} = T')\}$
  $x \leftarrow Q[a]$
$\{\mathbf{U}_W[Q[0]] * ... \mathbf{U}_W[x] ... * \mathbf{U}_W[Q[n]] * \mathbf{U}_Y[\mathsf{Trace}] \wedge \mathsf{Ct}(\mathsf{Trace} = T')\}$
  $Q[a] \leftarrow_\$ \mathbf{U}_{\{0..(2^L-1)\}}$
$\{(\text{Main Invariant} * \mathbf{U}_Y[T'] * \mathbf{U}_W[x]) \wedge \mathsf{Ct}(\mathsf{Trace} = T')\}$
(Start CONST rule) $\{\mathsf{Ct}(\mathsf{Trace} = T')\}$
  $l \leftarrow 0$
  while$_D$ $l \leq L$ **do**
    $S \leftarrow S \cup \mathsf{ReadBucket}(P(x, l))$
    $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“ReadBucket()”}, x, l)$
    $l \leftarrow l + 1$
  $data \leftarrow \mathsf{find}(a, S)$
  if$_D$ $op = \mathsf{Write}$ **then**
    $S \leftarrow (S - \{(a, data)\}) \cup \{(a, data^*)\}$
  $l \leftarrow L$
  while$_D$ $l \geq 0$ **do**
    $S' \leftarrow \{(a', data') \in S : P(x, l) = P(Q[a'], l)\}$
    $S' \leftarrow \mathsf{select}(Z, S')$
    $S \leftarrow S - S'$
    $\mathsf{WriteBucket}(P(x, l), S')$
    $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“WriteBucket()”}, x, l)$
    $l \leftarrow l - 1$
$\{\mathsf{Ct}(\mathsf{Trace} = T' +\!\!+ f(x))\}$ (End CONST rule)
$\{(\text{Main Invariant} * \mathbf{U}_Y[T'] * \mathbf{U}_W[x]) \wedge \mathsf{Ct}(\mathsf{Trace} = T' +\!\!+ f(x))\}$
$\{\text{Main Invariant} * \mathbf{U}_{Y \times f(W)}[\mathsf{Trace}]\}$
end function

FIGURE 5.5: Verification of path ORAM

are never modified. Thanks to using the CONST rule, reasoning proceeds via classical ($\mathsf{Ct}(\cdot)$) reasoning. It is easy to prove that at the end of the function we have $\mathsf{Ct}(\mathsf{Trace} = T' +\!\!+ f(x))$, where $f(x) = [(\mathsf{ReadBucket}(), x, 0), \ldots, (\mathsf{ReadBucket}(), x, L), (\mathsf{WriteBucket}(), x, L), \ldots, (\mathsf{WriteBucket}(), x, 0)]$.

Finally, we convert this assertion using the third proposition in Theorem 3.2 and the proposition introduced above to prove the desired invariant.

5.2.2 Omitted Subroutines

In this subsection, we present our concrete implementation and formal verification of two subroutines—$\mathsf{WriteBucket}()$ and $\mathsf{ReadBucket}()$—which were described but not fully specified in the original Path ORAM paper [Stefanov et al., 2018, Section 3.5]. As we

Let $\mathsf{mapping}(\text{“ReadBucket()”}, x, l) =$
$[(\text{“Read”}, P(x,l), 0), (\text{“Read”}, P(x,l), 1), \cdots, (\text{“Read”}, P(x,l), N-1)]$
Let $\mathsf{mapping}(\text{“WriteBucket()”}, x, l) =$
$[(\text{“Write”}, P(x,l), 0, \perp), (\text{“Write”}, P(x,l), 1, \perp), \cdots, (\text{“Write”}, P(x,l), N-1, \perp)]$,
where $P()$ is a deterministic function and $P(x,l)$ represents the l-level bucket at the path from root to the xth leaf node [Stefanov et al., 2018, Section 3.2].

function ReadBucket(B)
  $\{\mathsf{Ct}(\mathsf{Trace} = T \wedge B = P(x,l))\}$
  $S \leftarrow \{\}$
  $i \leftarrow 0$
  while$_D$ $i < Z$ **do** ▷ where Z is the constant size of each Bucket
    $d \leftarrow \mathsf{dec}(B[i])$
    $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Read”}, B, i)$
    $S \leftarrow S \cup d$
    $i \leftarrow i + 1$
  return S
  $\{\mathsf{Ct}(\mathsf{Trace} = T + (\text{“Read”}, B, 0) + (\text{“Read”}, B, 1) + \cdots + (\text{“Read”}, B, N-1))\}$
  $\{\mathsf{Ct}(\mathsf{Trace} = T +\!\!+ \mathsf{mapping}(\text{“ReadBucket()”}, x, l))\}$
end function

function WriteBucket(B, S)
  $\{\mathsf{Ct}(\mathsf{Trace} = T \wedge B = P(x,l))\}$
  $i \leftarrow 0$
  while$_D$ $i < Z$ **do** ▷ where Z is the constant size of each Bucket
    if$_R$ $i < \mathsf{size}(S)$ **then** ▷ The size of S is random
      $d \leftarrow S[i]$
    else
      $d \leftarrow \mathsf{Dummy}$ ▷ dummy value, such as 0 or random value
    $B[i] \leftarrow \mathsf{enc}(d)$
    $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Write”}, B, i, \perp)$
    $i \leftarrow i + 1$
  return S
  $\{\mathsf{Ct}(\mathsf{Trace} = T + (\text{“Write”}, B, 0, \perp) + \cdots + (\text{“Write”}, B, N-1, \perp))\}$
  $\{\mathsf{Ct}(\mathsf{Trace} = T +\!\!+ \mathsf{mapping}(\text{“WriteBucket()”}, x, l))\}$
end function

FIGURE 5.6: Implementation and verification of ReadBucket() and WriteBucket()

specified in Section 5.2, we will also establish a mapping from the abstract representations to the concrete traces produced by our implementations, thereby showing that the abstractions faithfully capture the essential behavior of the subroutines.

As illustrated in Fig. 5.6, the mapping function are defined at the beginning and the blue ghost codes are added by our transformation (Chapter 4). Both ReadBucket() and WriteBucket() operate on buckets of size Z, where Z is typically set to 4 or 5 in practice [Stefanov et al., 2018, Section 3.3]. The function ReadBucket(B) decrypts all

entries in the bucket B and returns the resulting plaintexts as an array S. Conversely, $\mathsf{WriteBucket}(B, S)$ writes the contents of S into the bucket B. If the size of S is smaller than Z, dummy values are written into the remaining slots to ensure a fixed memory access pattern.

Our verification establishes that $\mathsf{ReadBucket}(B)$ always performs exactly Z sequential reads from B, while $\mathsf{WriteBucket}(B, S)$ always performs exactly Z sequential writes to B. Thus, their behavior is fully determined by the choice of bucket B. Since in the main function (see Fig. 5.4) the bucket is given by $B = P(x, l)$, these operations are ultimately determined by the two parameters x and l. The detailed relation are shown by the mapping function defined at the beginning of Fig. 5.6.

5.3 Path Oblivious Heap

The Path Oblivious Heap [Shi, 2019] provides the standard interfaces of a heap including element insertion, deletion, finding the minimum, and extracting the minimum (this last being a combination of finding and deletion). The Path Oblivious Heap has been used to implement oblivious sorting [Shi, 2019], among other applications. The Path Oblivious Heap is inspired by Path ORAM [Stefanov et al., 2018] and shares the same data tree structure and several sub-functions.

However, unlike Path ORAM which provides only one interface, the Path Oblivious Heap provides multiple such interfaces which increases somewhat the complexity of its verification. The Insert function takes two parameters: the key k and value v to be inserted; it returns two values: *pos* and τ, the *position* and *timestamp* of the inserted element that together uniquely identify the inserted element in the heap. Timestamps are allocated deterministically: the first item inserted into the heap has timestamp 0, the second 1, and so on. The Delete function takes a position *pos* and a timestamp τ and removes the corresponding element that they uniquely identify.

This algorithm is designed to hide the contents of the key-value parameters passed to Insert (i.e., to hide the heap data), but not the kind of heap operations performed (i.e., whether Insert or Delete was called), to an adversary who can observe the algorithm's memory access pattern. Thus the adversary is assumed to know what heap operations will be performed and in what order; although not the parameters passed to

function INSERT(k, v)
  $pos \leftarrow_{\$} \mathbf{U}_{\{0,1,2,\ldots,N-1\}}$
  Add$(B_{root}, (k, v, (pos, \tau)))$
  $P \leftarrow_{\$} \mathbf{U}_{\{0,1,2,\ldots,N/2-1\}}$
  $P' \leftarrow_{\$} \mathbf{U}_{\{N/2,N/2+1,\ldots,N-1\}}$
  Evict(P)
  UpdateMin(P)
  Evict(P')
  UpdateMin(P')
  return (pos, τ)
end function

function DELETE(pos, τ)
  ReadNRm(pos, τ)
  Evict(pos)
  UpdateMin(pos)
end function

FIGURE 5.7: Main interfaces of the Path Oblivious Heap

those operations. As such, since timestamps are deterministically allocated, they reveal no information and so, following [Shi, 2019]'s original presentation, for simplicity we largely ignore them henceforth.

The INSERT function uniformly chooses a new position *pos* from the set of all such positions, $\{0, \ldots, N-1\}$, and returns it, where N is the maximum number of items the heap can store. When a DELETE happens, the location of the deleted element is revealed to the attacker in the memory-access pattern. However, each inserted element's location was chosen independently and uniformly from the same fixed set, so the location reveals nothing about the deleted element.

A notable feature—and a key verification challenge—of this algorithm is the delayed information revelation: positions are selected during INSERT operations but are only revealed during the corresponding DELETE operations. The number and order of INSERT and DELETE calls may vary arbitrarily (although they are not considered as secret), provided that each DELETE must happen after its corresponding INSERT.

At a high level, we address this challenge by constructing invariants and specifying contracts for the INSERT and DELETE functions. In particular, we show that if the sequence of function types (i.e., whether each call is an INSERT or a DELETE) is fixed, then the resulting memory access pattern is uniformly distributed over a fixed set, regardless of the specific parameters of the function calls. As a result, the memory access pattern reveals no information about those parameters.

5.3.1 Verification

Our verification applies to a perfectly oblivious approximation that never fails by assuming the private storage is infinite; in practice the failure probability (and, thus, the statistical distance between the real implementation and the perfectly oblivious approximation that we verify) is bounded by a negligible quantity [Shi, 2019, Corollary 2], thereby allowing Theorem 3.15 to conclude obliviousness for the imperfect algorithm.

Our goal is to prove that any two sequences of operations of the same length and that perform the same types of operations in the same order produce indistinguishable memory access patterns, in the sense that both are uniformly distributed over the same set of possibilities. A sequence of operations corresponds to a series of calls to the interfaces, e.g. a sequence could be [INSERT($\cdots$), INSERT($\cdots$), DELETE($\cdots$)]. The distribution of the memory access pattern produced by two executions of this sequence should be identical regardless of the parameters passed in each.

Fig. 5.7 depicts the oblivious heap algorithm [Shi, 2019, Section 3.3]. The various sub-functions (e.g. Evict, UpdateMin) also update Trace to record their memory-access patterns, which will be introduced in Section 5.3.2.

In Fig. 5.8, we prove obliviousness by verifying that each interface (1) maintains an invariant $\mathsf{inv}(E)$, where E is the sequence of existing elements' position in the heap, and (2) ensures that the resulting memory-access pattern is uniformly distributed over a fixed set independent of the input parameters. The invariant states that each position is independently uniformly distributed over the fixed set of possible positions:

$$\mathsf{inv}([pos_1, pos_2, \cdots, pos_n]) = \mathbf{U}_{\{0..N-1\}}[pos_1] * \mathbf{U}_{\{0..N-1\}}[pos_2] * .. * \mathbf{U}_{\{0..N-1\}}[pos_n]$$

The preconditions in Fig. 5.8 should be read as quantifying over T, the set over which the historical memory-access pattern is uniformly distributed. At the beginning, T is empty and we will verify the existence of T as an global invariant between each call of the interfaces.

We put the classical, Hoare logic verification of the sub-functions in the next section (5.3.2, they are totally deterministic). The classical specification for each sub-function

Let $\mathsf{inv}([pos_1, pos_2, \cdots, pos_n]) = \mathbf{U}_{\{0..N-1\}}[pos_1] * \mathbf{U}_{\{0..N-1\}}[pos_2] * .. * \mathbf{U}_{\{0..N-1\}}[pos_n]$.

function INSERT(k, v)
$\{\mathbf{U}_T[\mathsf{Trace}] * \mathsf{inv}(E)\}$
$pos \leftarrow_\$ \mathbf{U}_{\{0,1,2,\ldots,N-1\}}$
$\{\mathbf{U}_T[\mathsf{Trace}] * \mathsf{inv}(E +\!\!+ [pos])\}$
$\mathsf{Trace}' \leftarrow \mathsf{Trace}$
$\{\mathbf{U}_T[\mathsf{Trace}'] * \mathsf{inv}(E +\!\!+ [pos]) \wedge \mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}')\}$
$\mathsf{Add}(B_{root}, (k, v, (pos, \tau)))$
$\{\mathbf{U}_T[\mathsf{Trace}'] * \mathsf{inv}(E +\!\!+ [pos]) \wedge \mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}' +\!\!+ TA(\text{root}))\}$
$\{\mathbf{U}_{T\times\{TA(\text{root})\}}[\mathsf{Trace}] * \mathsf{inv}(E +\!\!+ [pos])\}$
$P \leftarrow_\$ \mathbf{U}_{\{0,1,2,\ldots,N/2-1\}}$
$P' \leftarrow_\$ \mathbf{U}_{\{N/2,N/2+1,\ldots,N-1\}}$
$\{\mathbf{U}_{T\times\{TA(\text{root})\}}[\mathsf{Trace}] * \mathsf{inv}(E +\!\!+ [pos]) * \mathbf{U}_{\{0..N/2-1\}}[P] * \mathbf{U}_{\{N/2..N-1\}}[P']\}$
$\mathsf{Trace}' \leftarrow \mathsf{Trace}$
$\{\mathbf{U}_{T\times\{TA(\text{root})\}}[\mathsf{Trace}'] * \mathsf{inv}(E +\!\!+ [pos]) * \mathbf{U}_{\{0..N/2-1\}}[P] * \mathbf{U}_{\{N/2..N-1\}}[P'] \wedge$
$\mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}')\}$
$\mathsf{Evict}(P)$
$\mathsf{UpdateMin}(P)$
$\{\mathbf{U}_{T\times\{TA(\text{root})\}}[\mathsf{Trace}'] * \mathsf{inv}(E +\!\!+ [pos]) * \mathbf{U}_{\{0..N/2-1\}}[P] * \mathbf{U}_{\{N/2..N-1\}}[P'] \wedge$
$\mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}' +\!\!+ TE(P) +\!\!+ TU(P))\}$
$\{\mathbf{U}_{T\times\{TA(\text{root})\}\times\{TE(p)+\!\!+TU(p) \mid 0\le p<N/2\}}[\mathsf{Trace}] * \mathsf{inv}(E +\!\!+ [pos]) * \mathbf{U}_{\{N/2..N-1\}}[P']\}$
$\mathsf{Trace}' \leftarrow \mathsf{Trace}$
$\{\mathbf{U}_{T\times\{TA(\text{root})\}\times\{TE(p)+\!\!+TU(p) \mid 0\le p<N/2\}}[\mathsf{Trace}'] * \mathsf{inv}(E +\!\!+ [pos]) * \mathbf{U}_{\{N/2..N-1\}}[P'] \wedge$
$\mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}')\}$
$\mathsf{Evict}(P')$
$\mathsf{UpdateMin}(P')$
$\{\mathbf{U}_{T\times\{TA(\text{root})\}\times\{TE(p)+\!\!+TU(p) \mid 0\le p<N/2\}}[\mathsf{Trace}'] * \mathsf{inv}(E +\!\!+ [pos]) * \mathbf{U}_{\{N/2..N-1\}}[P'] \wedge$
$\mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}' +\!\!+ TE(P') +\!\!+ TU(P'))\}$
$\{\mathbf{U}_{T\times\{TA(\text{root})\}\times\{TE(p)+\!\!+TU(p) \mid 0\le p<N/2\}\times\{TE(p)+\!\!+TU(p) \mid N/2\le p<N\}}[\mathsf{Trace}'] *$
$\mathsf{inv}(E +\!\!+ [pos])\}$
return (pos, τ)
end function

function DELETE(pos, τ)
$\{\mathbf{U}_T[\mathsf{Trace}] * \mathsf{inv}(E) \wedge \mathsf{Ct}(pos \in E)\}$
$\mathsf{Trace}' \leftarrow \mathsf{Trace}$
$\{\mathbf{U}_T[\mathsf{Trace}'] * \mathsf{inv}(E) \wedge \mathsf{Ct}(pos \in E) \wedge \mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}')\}$
$\mathsf{ReadNRm}(pos, \tau)$
$\{\mathbf{U}_T[\mathsf{Trace}'] * \mathsf{inv}(E) \wedge \mathsf{Ct}(pos \in E) \wedge \mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}' +\!\!+ TR(pos))\}$
$\mathsf{Evict}(pos)$
$\{\mathbf{U}_T[\mathsf{Trace}'] * \mathsf{inv}(E) \wedge \mathsf{Ct}(pos \in E) \wedge \mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}' +\!\!+ TR(pos) +\!\!+ TE(pos))\}$
$\mathsf{UpdateMin}(pos)$
$\{\mathbf{U}_T[\mathsf{Trace}'] * \mathsf{inv}(E) \wedge \mathsf{Ct}(pos \in E) \wedge$
$\mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}' +\!\!+ TR(pos) +\!\!+ TE(pos) +\!\!+ TU(pos))\}$
(taking pos_i out from invariant)
$\{\mathbf{U}_T[\mathsf{Trace}'] * \mathsf{inv}(E \setminus [pos]) * \mathbf{U}_{\{0..N-1\}}[pos] \wedge \mathsf{Ct}(pos \in E) \wedge$
$\mathsf{Ct}(\mathsf{Trace} = \mathsf{Trace}' +\!\!+ TR(pos) +\!\!+ TE(pos) +\!\!+ TU(pos))\}$
(using proposition 1.8)
$\{\mathbf{U}_{T\times\{TR(p)+\!\!+TE(p)+\!\!+TU(p) \mid 0\le p<N\}}[\mathsf{Trace}] * \mathsf{inv}(E \setminus [pos])\}$
end function

FIGURE 5.8: Path Oblivious Heap (Fig. 5.7) Verification

states that it adds to the Trace a fixed access pattern that depends only on the input parameter. For instance, Evict's classical Hoare logic specification is:

$\vdash \{\mathsf{Trace} = \mathsf{Trace}'\}\ \mathsf{Evict}(x)\ \{\mathsf{Trace} = \mathsf{Trace}' +\!\!+ TE(x)\}$, where $TE(x)$ ("Trace of Evict") abbreviates the access pattern for Evict for input parameter x.

We follow this same naming convention throughout: e.g. $TU(x)$ is the memory-access pattern of $\mathsf{UpdateMin}(x)$. For a set X, we write $TE(X)$ to mean $\{TE(x) \mid x \in X\}$ and so on.

Since the preconditions of these triples specify the initial value of Trace using an auxiliary variable, we insert a line of ghost code $\mathsf{Trace}' \leftarrow \mathsf{Trace}$ prior to each invocation of these deterministic subfunctions (e.g. Evict) to ensure that their respective preconditions are satisfied. The ghost variable Trace' is not read anywhere in the program and has no effect on the program's execution.

DELETE's precondition also makes sure its input is valid with $\mathsf{Ct}(pos \in E)$, i.e. we only delete existing elements. We first add a ghost variable Trace' to record the initial value of Trace and then use $\mathsf{Ct}(\cdot)$ (certain) reasoning for the following three function calls. Finally we convert the assertion to the desired one by Theorem 3.2. To do so, we use the fact that TR, TE, TU are bijective and always produce the sequence with the same length on their valid inputs (from 0 to $N-1$) respectively, to satisfy the assumption of the 8th proposition of Theorem 3.2.

Finally, the verification of INSERT shares the same idea as DELETE, repeated several times. Note, $TA(\text{root})$ is a fixed sequence because sub-function ADD's memory-access pattern is independent of ADD's arguments (see Section 5.3.2). *pos* is the position of the inserted element and is never used or leaked in this function. It is recorded in the invariant and will be released (and leaked) when it is deleted; however, as explained in Section 5.3, doing so reveals nothing since *pos* was chosen uniformly and independently of secrets.

5.3.2 Path Oblivious Heap Deterministic Sub-functions

In this section we verify the memory access pattern of several deterministic sub-functions satisfy the specification used in Section 5.3.1.

$$\mathsf{TA}(x) = \mathsf{TD}(x) =$$
$$\begin{cases} [(\text{“Read”}, 1, 0), (\text{“Write”}, 1, 0, \bot), ..., (\text{“Read”}, 1, n_{\text{root}}), (\text{“Write”}, 1, n_{\text{root}}), \bot] & \text{, if } x = 1 \\ [(\text{“Read”}, x, 0), (\text{“Write”}, x, 0, \bot), ..., (\text{“Read”}, x, 3), (\text{“Write”}, x, 3, \bot)] & \text{, otherwise} \end{cases}$$
$$\mathsf{TR}(x) = \mathsf{TD}(\mathsf{path}(x)[0]) +\!\!+ \mathsf{TD}(\mathsf{path}(x)[1]) +\!\!+ \cdots +\!\!+ \mathsf{TD}(\mathsf{path}(x)[\log(N)])$$

$$\mathsf{TU}_{\mathsf{ith}}(x, i) = [(\text{“readAll”}, \mathsf{pathR}(x)[i]), (\text{“readMin”}, 2 * \mathsf{pathR}(x)[i]), (\text{“readMin”}, 2 * \mathsf{pathR}(x)[i] + 1), (\text{“writeMin”}, \mathsf{pathR}(x)[i], \bot)]$$

$$\mathsf{TU}(x) = [(\text{“readAll”}, \mathsf{pathR}(x)[0]), (\text{“writeMin”}, \mathsf{pathR}(x)[0])] +\!\!+ \mathsf{TU}_{\mathsf{ith}}(x, 1) +\!\!+ \mathsf{TU}_{\mathsf{ith}}(x, 2) +\!\!+ \cdots +\!\!+ \mathsf{TU}_{\mathsf{ith}}(x, \log(2N) - 1)$$

$$\mathsf{TE}(x) = [(\text{“readAll”}, \mathsf{path}(x)[0]), (\text{“readAll”}, \mathsf{path}(x)[1]), \cdots, (\text{“readAll”}, \mathsf{path}(x)[\log(N)])] +\!\!+ [(\text{“writeAll”}, \mathsf{path}(x)[0], \bot), \cdots, (\text{“writeAll”}, \mathsf{path}(x)[\log(N)], \bot)]$$

FIGURE 5.9: Path Oblivious Heap's trace functions

The path oblivious heap is implemented internally by two arrays: there is an array *tree* of tree nodes that represents the tree, and an array *min* where $min[i]$ stores the minimum element of the sub-tree rooted at node $tree[i]$. The length of both arrays is $2N$; however index 0 is unused, as is standard for using an array to represent a heap.

The reads and writes to both arrays are observable to the attacker.

Each node in the tree has a fixed number of positions for storing elements, where an element has the form (k, v, τ) for key k, value v and timestamp τ. The number (denoted n_{root}) of positions for the root node is decided by the algorithm's security parameter [Shi, 2019], whereas the number of positions for the other nodes is 4. All of them store a dummy element DUMMY initially.

We use ghost code to record the accesses to the *tree* and *min* arrays. When we access a position $tree[i][j]$, we add $(\text{“Read”}, i, j)$ or $(\text{“Write”}, i, j, \bot)$ into the memory-access trace depending on which type of access was performed. Similarly, when we access a position $min[i]$, we add $(\text{“readMin”}, i)$ or $(\text{“writeMin”}, i, \bot)$ into the trace. When we read an entire node, we use $(\text{“readAll”}, i)$ to denote $[(\text{“Read”}, i, 0), (\text{“Read”}, i, 1), ...]$ up to the number of positions for the node; we use corresponding notation for write accesses.

Given an integer p such that $0 \leq p < N$, we define $\mathsf{path}(p)$ as the sequence of indexes of the path from the root to the pth leaf of the tree. For example, $\mathsf{path}(0) = [1, 2, 4, ..., N]$. We also define $\mathsf{pathR}(p)$ as the reversed sequence of $\mathsf{path}(p)$.

```
function ADD(i, (k, v, τ))
    {Trace = Trace'}
    s ← false
    j ← 0
    while_D j < length(tree[i]) do
        w ← dec(tree[i][j])
        Trace ← Trace + ("Read", i, j)
        if_D w = DUMMY ∧ (!s) then
            w ← (k, v, τ)
            s ← true
        tree[i][j] ← enc(w)
        Trace ← Trace + ("Write", i, j, ⊥)
        j ← j + 1
    {Trace = Trace' + TA(i)}
end function

function EVICT(p)
    {Trace = Trace'}
    i ← 0
    A ← []
    while_D i < log(2N) do
        A ← A + dec(tree[path(p)[i]])
        Trace ← Trace + ("readAll", path(p)[i])
        i ← i + 1
    A ← Evict_Locally(A)
    while_D i < log(2N) do
        tree[path(p)[i]] ← enc(A[i])
        Trace ← Trace + ("writeAll", path(p)[i], ⊥)
        i ← i + 1
    {Trace = Trace' + TE(p)}
end function

function READNRM(p, τ)
    {Trace = Trace'}
    j ← 0
    while_D j < log(2N) do
        DEL(path(p)[j], τ, Trace)
        j ← j + 1
    {Trace = Trace' + TR(p)}
end function

function DEL(i, τ)
    {Trace = Trace'}
    j ← 0
    while_D j < length(tree[i]) do
        w ← dec(tree[i][j])
        Trace ← Trace + ("Read", i, j)
        if_D τ = w[2] then
            w ← DUMMY
        tree[i][j] ← enc(w)
        Trace ← Trace + ("Write", i, j, ⊥)
        j ← j + 1
    {Trace = Trace' + TD(i)}
end function

function UPDATEMIN(p)
    {Trace = Trace'}
    i ← 0
    while_D i < log(2N) do
        j ← pathR(p)[i]
        A ← dec(tree[j])
        Trace ← Trace + ("readAll", j)
        if_D i > 0 then
            A ← A + dec(min[2j])
            Trace ← Trace + ("readMin", 2j)
            A ← A + dec(min[2j + 1])
            Trace ← Trace + ("readMin", 2j + 1)
        min[j] ← enc(min(A))
        Trace ← Trace + ("writeMin", j, ⊥)
        i ← i + 1
    {Trace = Trace' + TU(p)}
end function
```

FIGURE 5.10: Path Oblivious Heap's deterministic sub-functions

Then the sub-functions and corresponding specifications are in Fig. 5.10. The corresponding functions that define their memory-access patterns are in Fig. 5.9. Note that Evict_Locally(), called by Evict(), moves elements from the root towards the leaf on a given path, as described in [Shi, 2019, Section 3.2]. It operates over the private memory A and so its memory accesses are unobservable to the attacker.

5.4 The Melbourne Shuffle

The Melbourne Shuffle [Ohrimenko et al., 2014] is a probabilistic oblivious algorithm that takes an array (database) I as the public input and a target permutation π as its secret input. Its job is to shuffle I according to the permutation π, placing the shuffled data into the output array O. To do so, it makes use of a (larger) temporary array T. The algorithm is designed to be oblivious despite the access patterns of I, T and O being observable to attackers.

In practice, this algorithm is designed to improve e.g. storage solutions for network-based outsourcing of data [Goodrich et al., 2012, Williams et al., 2008]: the arrays I and O are held locally by a client; shuffling is performed by a server whose memory is T. All communication between the client and server is encrypted; however, there might be spies who can nonetheless observe the access patterns to the three arrays (e.g. via cache side channels). Moreover, the cloud provider might be malicious and so might directly observe the contents of array T to learn the secret data or the desired permutation. For this reason, the algorithm keeps the contents of T encrypted.

function SHUFFLE(I, π, O)
  $\pi_1 \leftarrow_\$ \mathbf{U}_{F(I)}$;
  $T \leftarrow []$;
  shuffle_pass(I, T, π_1, O);
  copy(O, I);
  shuffle_pass(I, T, π, O);
end function

FIGURE 5.11: The Melbourne Shuffle

The algorithm is presented in Fig. 5.11. Its primary function, shuffle(), takes an input array I and rearranges it according to a given permutation π, placing the resulting order into an output array O. The algorithm first uniformly selects an intermediate permutation π_1 from the set of all possible permutations of I, denoted as $F(I)$. It then

function SHUFFLE'(I, π, O)
  Trace $\leftarrow$ []
  $\pi_1 \leftarrow_\$ \mathbf{U}_{S_p(\pi) \cap S_p(\pi_I)}$
  $T \leftarrow$ []
  shuffle_pass$(I, T, \pi_1, O,$ Trace$)$
  copy(O, I); Trace $\leftarrow$ Trace $+$ ("copy()", "O", "I")
  shuffle_pass$(I, T, \pi, O,$ Trace$)$
end function

FIGURE 5.12: Transformation of the perfect Melbourne shuffle

initializes a temporary array T and invokes the internal function shuffle_pass() twice. The first invocation applies the intermediate permutation π_1 to the original array, and the second invocation applies the desired permutation π (from π_1) to complete the shuffle.

The algorithm does not directly apply the target permutation π to the input array due to a negligible chance of failure within the internal function shuffle_pass(). Specifically, this internal function may fail for particular input permutations. Directly shuffling the input according to the final permutation would cause consistent failure for certain permutations, thus leaking information about the permutation. By introducing an intermediate permutation, the Melbourne Shuffle algorithm ensures that the probability of failure is bounded by a negligible function with respect to the size of I [Ohrimenko et al., 2014] given any possible input.

Following the conventions established at the algorithm's inception, permutations π and π_1 are expressed as functions mapping each data element to its intended index position [Ohrimenko et al., 2014]. For instance, given an input array $I = [x, y, z]$ and permutation π defined as $\pi(x) = 1$, $\pi(y) = 0$, and $\pi(z) = 2$, the resulting array after a shuffle pass would be $[y, x, z]$. Without loss of generality, we assume the input array I contains distinct elements.

5.4.1 Approximation and Transformation

To apply our verification approach, we first construct a perfectly oblivious approximation of Fig. 5.11 by modifying the random selection of π_1. Rather than choosing from all permutations $F(I)$, it instead selects from the set of permutations under which the algorithm doesn't fail. Letting π_I denote the unique permutation that describes the contents of the input array I, this set of permutations under which the algorithm does not fail is $S_p(\pi_I) \cap S_p(\pi)$, where S_p is defined following Ohrimenko et al. [2014].

Let $V(\pi, i, j) = \{v \mid i \leq \pi(v) < j\}$ denote ith to jth values in the permutation π. Then S_p is a function that takes a permutation whose length is n, and returns the set of permutations for which the input array uniquely described by π can be shuffled without failure. S_p is defined as the set of all π' such that:

$\mathsf{size}(\pi) = \mathsf{size}(\pi') = n$ and $\forall i \in \{0, 1, ..., \sqrt{n} - 1\}$,

$$\mathsf{size}(V(\pi, \sqrt{n} \cdot i, \sqrt{n} \cdot (i+1)) \cap V(\pi', \sqrt{n} \cdot i, \sqrt{n} * (i+1))) \leq p \cdot \log(n).$$

This definition of S_p is symmetric about π and π', so we have that $\forall \pi, \pi'.\ \pi \in S_p(\pi') \iff \pi' \in S_p(\pi)$. Thus, the Melbourne shuffle succeeds if and only if $\pi_1 \in S_p(\pi_I) \cap S_p(\pi)$ [Ohrimenko et al., 2014, Lemma 4.4].

The statistical distance between our approximation and the original algorithm is at most a negligible function of n [Ohrimenko et al., 2014, Lemma 4.3]. We can thus verify Fig. 5.11 by proving perfect obliviousness of the approximation following Lemma 3.15.

The function $\mathsf{copy}(O, I)$ is re-written by us according to the requirement of transformation (Definition 4.2). It copies the entire contents of O to I, performing decryption followed by re-encryption, thereby ensuring that the second shuffle pass operates on the output of the first. Its corresponding memory access pattern in the simplifed transformation is deterministic and is abstractly represented as $(\text{“}\mathsf{copy}()\text{”}, \text{“}O\text{”}, \text{“}I\text{”}, n)$. This is shorthand for assuming there is a memory access pattern function for copy() that is deterministic in the starting addresses of O and I, and their lengths n - we capture here instead the arguments such a deterministic function would depend on

5.4.2 Verification

We verify the perfectly oblivious approximation by applying our logic to prove that it produces a fixed, deterministic memory access pattern.

Let $\mathsf{TS}(\text{“}I\text{”}, \text{“}T\text{”}, \text{“}O\text{”}, n)$ denote the memory access pattern determined by n and produced by calling $\mathsf{shuffle\_pass}(I, T, \pi, O)$ for any π, we prove the memory access pattern of Fig. 5.13 is $\mathsf{TS}(\text{“}I\text{”}, \text{“}T\text{”}, \text{“}O\text{”}, n) + (\text{“}\mathsf{copy}()\text{”}, \text{“}O\text{”}, \text{“}I\text{”}, n) \mathbin{+\!\!+} \mathsf{TS}(\text{“}I\text{”}, \text{“}T\text{”}, \text{“}O\text{”}, n)$, as shown in the final postcondition.

The precondition of Fig. 5.13 states that the initial array I contains n unique elements, and that π_I does indeed describe its contents correctly.

function SHUFFLE'(I, π, O)
$\{\mathsf{Ct}(\mathsf{size}(I) = n = \mathsf{size}(\mathsf{Set}(I)) \wedge \forall v \in I.\ \ I[\pi_I(v)] = v)\}$
  $\mathsf{Trace} \leftarrow []$
$\{\mathsf{Ct}(\mathsf{Trace} = [] \wedge \mathsf{size}(I) = n = \mathsf{size}(\mathsf{Set}(I)) \wedge (\forall v \in I.\ \ I[\pi_I(v)] = v))\}$
  $\pi_1 \leftarrow_\$ \mathbf{U}_{S_p(\pi) \cap S_p(\pi_I)}$
$\{\mathsf{Ct}(\mathsf{Trace} = [] \wedge \mathsf{size}(I) = n = \mathsf{size}(\mathsf{Set}(I)) \wedge (\forall v \in I.\ \ I[\pi_I(v)] = v))$
    $* \mathbf{U}_{\pi_1}[S_p(\pi) \cap S_p(\pi_I)]\}$
$\{\mathsf{Ct}(\mathsf{Trace} = [] \wedge \mathsf{size}(I) = n = \mathsf{size}(\mathsf{Set}(I)) \wedge$
    $(\forall v \in I.\ \ I[\pi_I(v)] = v) \wedge \pi_1 \in S_p(\pi) \cap S_p(\pi_I))\}$
  $T \leftarrow []$
$\{\mathsf{Ct}(\mathsf{Trace} = [] \wedge \mathsf{size}(I) = n = \mathsf{size}(\mathsf{Set}(I)) \wedge (\forall v \in I.\ \ I[\pi_I(v)] = v) \wedge$
    $T = [] \wedge \pi_1 \in S_p(\pi) \cap S_p(\pi_I)\}$
  $\mathsf{shuffle\_pass}(I, T, \pi_1, O, \mathsf{Trace})$
$\{\mathsf{Ct}(\mathsf{Trace} = \mathsf{TS}(\text{“}I\text{”}, \text{“}T\text{”}, \text{“}O\text{”}, n) \wedge \mathsf{size}(O) = n = \mathsf{size}(\mathsf{Set}(O)) \wedge$
    $(\forall v \in O.\ \ O[\pi_1(v)] = v) \wedge \pi_1 \in S_p(\pi) \cap S_p(\pi_I))\}$
  $\mathsf{copy}(O, I); \mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“}\mathsf{copy}()\text{”}, \text{“}O\text{”}, \text{“}I\text{”}, n)$
$\{\mathsf{Ct}(\mathsf{Trace} = \mathsf{TS}(\text{“}I\text{”}, \text{“}T\text{”}, \text{“}O\text{”}, n) + (\text{“}\mathsf{copy}()\text{”}, \text{“}O\text{”}, \text{“}I\text{”}, n) \wedge$
    $\mathsf{size}(I) = n = \mathsf{size}(\mathsf{Set}(I)) \wedge (\forall v \in I.\ \ I[\pi_1(v)] = v) \wedge \pi \in S_p(\pi_1))\}$
  $\mathsf{shuffle\_pass}(I, T, \pi, O, \mathsf{Trace})$
$\{\mathsf{Ct}(\mathsf{Trace} = \mathsf{TS}(\text{“}I\text{”}, \text{“}T\text{”}, \text{“}O\text{”}, n) + (\text{“}\mathsf{copy}()\text{”}, \text{“}O\text{”}, \text{“}I\text{”}, n) +\!\!+ \mathsf{TS}(\text{“}I\text{”}, \text{“}T\text{”}, \text{“}O\text{”}, n)$
    $\wedge\ \mathsf{size}(O) = n = \mathsf{size}(\mathsf{Set}(O)) \wedge (\forall v.\ \ O[\pi(v)] = v))\}$
end function

FIGURE 5.13: Verification of the Melbourne shuffle. $\mathsf{Set}(A)$ is the set of values in array A.

This proof relies on the inner function shuffle_pass adhering to the following specification.

$$\{\mathsf{Ct}(\mathsf{Trace} = X \wedge \mathsf{size}(I) = n = \mathsf{size}(\mathsf{Set}(I)) \wedge (\forall v \in I.\ \ I[\pi_1(v)] = v) \wedge \pi \in S_p(\pi_1))\}$$
$$\mathsf{shuffle\_pass}(I, T, \pi, O, \mathsf{Trace})$$
$$\{\mathsf{Ct}(\mathsf{Trace} = X +\!\!+ \mathsf{TS}(\text{“}I\text{”}, \text{“}T\text{”}, \text{“}O\text{”}, n) \wedge \mathsf{size}(O) = n = \mathsf{size}(\mathsf{Set}(O)) \wedge$$
$$(\forall v \in O.\ \ O[\pi(v)] = v))\}$$

Note that this is a classical Hoare logic specification, and that shuffle_pass is a deterministic algorithm. Therefore this specification can be proved in classical Hoare logic [Hoare, 1969] and so is introduced in Appendix A. Importantly, the postcondition of this specification states also that the shuffle is correctly performed: $\forall v \in O.\ \ O[\pi(v)] = v$. This is crucial as otherwise the two calls to shuffle_pass cannot be chained together (as the precondition for the second call relies on this assumption). Thus the obliviousness of this algorithm depends on its correctness. This explains the vital importance of being

able to mix classical correctness reasoning with probabilistic reasoning, as supported by our logic via the $\mathsf{Ct}(\cdot)$ assertions and associated rules (Fig. 3.3).

5.5 Summary

This chapter demonstrated the practical applicability and expressiveness of our verification framework by applying it to four diverse, non-trivial probabilistic oblivious algorithms: Oblivious Sampling [Sasy and Ohrimenko, 2019], the Melbourne Shuffle [Ohrimenko et al., 2014], Path ORAM [Stefanov et al., 2018], and the Path Oblivious Heap [Shi, 2019]. Each of these algorithms was formally verified—at the pen-and-paper level—using the logic introduced in Chapter 3 and the transformation framework of Chapter 4.

All four case studies involved the use of encryption and required our transformation. The transformed versions were then verified using our logic, proving either statistical or computational obliviousness depending on the assumed security properties of the encryption scheme. Our verification shows that the framework is expressive and modular enough to support complex reasoning involving probabilistic independence, uniform distributions, encryption soundness, and control-flow branching.

These results demonstrate that our framework is not only theoretically sound (as established in previous chapters), but also practically applicable to real-world oblivious algorithms with significant cryptographic and oblivious requirements.

Chapter 6

Conclusion and Future Directions

This thesis introduced a comprehensive framework for the formal verification of probabilistic oblivious algorithms. These algorithms, central to secure and privacy-preserving computation, pose significant verification challenges due to their combination of probabilistic behavior, cryptographic operations, and subtle side-channel resistance properties.

6.1 Summary of Contributions

Our work addressed these challenges through a sequence of novel, systematically developed components:

- **A Program Logic for Obliviousness:** In Chapter 3, we developed a new program logic that combines classical reasoning with probabilistic and independence-based reasoning. Situated atop Probabilistic Separation Logic (PSL), it supports verification of properties such as uniform distribution, probabilistic independence, and their interaction with classical assertions. Key constructs include the certainty assertion $\mathsf{Ct}(\cdot)$, and several inference rules allowing the interaction of the two reasoning styles.

- **A Framework for Verifying Encryption Usage:** In Chapter 4, we introduced a systematic transformation and verification method to check for the correct application of encryption in oblivious algorithms. This framework handles both statistical and computational security definitions and distinguishes encrypted from unencrypted trace data to ensure secret values are properly protected.

- **Formal Verification of Case Studies:** In Chapter 5, we applied our framework to verify four real-world oblivious algorithms:

  – *Oblivious Sampling*, which requires reasoning about dynamic random choices and secret-dependent control flow.

  – *Path ORAM*, involving invariants over probabilistic independence.

  – *The Melbourne Shuffle*, combining deterministic access patterns with cryptographic assumptions.

  – *Path Oblivious Heap*, with delayed information release and complex update behavior.

  To the best of our knowledge, these case studies are formally verified here for the first time.

- **Artifact:** The soundness of our logic, transformation and the statistical security implication theorem were fully formalised and proved in Isabelle/HOL, uncovering and correcting several oversights in the original PSL. This provides a robust foundation for trustworthy formal reasoning in probabilistic settings.

6.2 Future Work

This work opens several promising directions for future research:

Automation and Tactic Support A natural extension of this work lies in improving the degree of proof automation and tactic support available to users of the logic. Currently, a significant amount of manual effort is required to construct and manage invariants, as well as to select suitable reasoning rules for different proof obligations. By integrating invariant inference techniques and tactic-guided rule application, verification could become substantially more scalable and accessible to non-experts. One

particularly challenging aspect is the automatic determination of the most appropriate reasoning style—whether purely classical or probabilistic—for each proof step. Addressing this problem may require the development of heuristics or meta-logical rules capable of adapting to the structure of the algorithm being verified. Successful automation in this direction would not only reduce the cognitive burden on verifiers but also pave the way for applying these techniques to larger systems.

Direct Reasoning about Negligible Failures Another compelling line of research is to extend our logic or introduce the new logic to reason directly about bounded failure probabilities, as discussed in Section 3.7.2. The current framework is designed around idealized assumptions, but practical algorithms often tolerate negligible failure probabilities for better space complexity. Developing methods to capture and verify such imperfect behaviors within the logic would significantly broaden its practical applicability. However, reasoning about bounded failures is technically difficult, as the structure and nature of the failure arguments vary widely across different algorithms. Building a systematic logical framework to accommodate these diverse techniques is therefore an ambitious but impactful research goal.

Moreover, logics that support conditional probability reasoning (see Section 2.4.2.3) may enable a direct formalization of imperfect obliviousness for practical algorithms, rather than reasoning only about idealized versions without failure probability. In particular, one could capture the guarantee as: if the program does not fail, then it is oblivious.

Broader Applicability Although the central motivation of this work has been the verification of probabilistic oblivious algorithms, the underlying techniques have the potential to extend much further. The integration of classical reasoning about deterministic state and probabilistic reasoning about distributional properties provides a flexible foundation that may benefit other domains of security verification. For instance, many cryptographic primitives and randomised algorithms require reasoning about independence or probabilistic indistinguishability, which closely parallels the arguments developed in this thesis. By generalizing the framework beyond obliviousness, future work may uncover applications in the formal verification of cryptographic protocols, randomised data structures, and probabilistic concurrent systems. In doing so, the logic could contribute to a broader class of verification tasks where probabilistic reasoning plays a central role.

Support for Non-Uniform Distributions The current assertion language is restricted to uniform distributions. While this is sufficient for the case studies considered in this thesis, it limits the broader applicability of the framework. In practice, many randomised algorithms rely on biased coins, weighted sampling, or other non-uniform distributions that cannot be naturally expressed within the present design. Extending the logic to accommodate such distributions would therefore represent a substantial advancement. The key objective of this extension would be to increase expressiveness while preserving the soundness of existing proof rules. Technically, this requires enriching the assertion language with symbolic representations of arbitrary distributions and introducing additional inference rules. Since the current rules do not depend on uniformity assumptions, they would remain valid under this extension. While the additional complexity introduces new proof challenges, the outcome would greatly expand the range of algorithms and applications that can be formally verified within this framework.

Systematic Comparison of Verification Approaches Path ORAM (one of our case studies in Chapter 5) has also been the subject of prior verification efforts [Sahai et al., 2020, Leung et al., 2023]. In particular, [Sahai et al., 2020] analyze the algorithm in a non-probabilistic setting by modeling it as a nondeterministic transition system and applying model counting to establish a security property. Their property states that, for any observable output, there exists a sufficiently large set of inputs such that the adversary cannot determine which specific input produced that output. This specification is arguably the strongest guarantee attainable in a nondeterministic model. However, it is also satisfied by implementations that employ biased randomness, which may in fact leak excessive information about the input. By contrast, our specification requires that, for each input, the resulting outputs are identically distributed with respect to the adversary's observational capabilities—a condition that would not hold under such biased implementations. A systematic comparison between these complementary approaches would provide valuable insight into their respective strengths and limitations.

Verification under Alternative Obliviousness Standards Some oblivious algorithms adopt security definitions that are not based on probabilistic independence, placing them beyond the direct scope of our current framework. A notable example is the

class of *Differentially Oblivious Algorithms* [Chan et al., 2022], which extend ideas from differential privacy to the setting of obliviousness.

Differential obliviousness ensures that the memory access patterns of an algorithm do not reveal sensitive information about the input, except up to a small statistical leakage. More precisely, it requires that for any two adjacent inputs (e.g. differing in one record), the distribution over the resulting access patterns is close in a probabilistic sense, typically bounded by parameters (ε, δ), analogous to differential privacy [Dwork et al., 2006, Dwork and Roth, 2014].

Verifying differential obliviousness involves reasoning about *approximate probabilistic couplings* between program executions, rather than exact independence. This approach allows one to prove that two probabilistic programs behave similarly with high probability, even if their outputs are not exactly the same. Techniques such as the approximate relational Hoare logic (apRHL) [Barthe et al., 2016a, 2013] and other relevant works [Zhang and Kifer, 2017, Farina et al., 2021, Barthe et al., 2014b, Bichsel et al., 2018] have been developed to support formal reasoning about such probabilistic closeness.

Integrating differential obliviousness into a verification framework poses several challenges. First, it requires a logic that can express relational properties over distributions with approximate bounds. Second, the verification often depends on fine-grained reasoning about the interaction between control flow, randomness, and memory access patterns. These aspects are fundamentally different from the independence-based reasoning used in our logic, suggesting the need for orthogonal techniques.

In future work, it would be valuable to investigate whether these verification methods could be combined with our framework. In particular, it is worth exploring how approximate couplings—central to differential privacy and differential obliviousness—might interact with our transformation-based reasoning framework.

Appendix A

Deterministic Subfunctions of the Melbourne Shuffle

In this appendix, we present the deterministic subfunctions of the Melbourne Shuffle. As shown in Fig. A.1, the primary subfunction is Shuffle_Pass(), which internally calls getRange(), putRange(), and several other auxiliary routines. Since getRange() and putRange() perform reads and writes to public locations (I, T, and O), their memory access patterns are observable. We implement these functions in accordance with the descriptions provided in [Ohrimenko et al., 2014, Section 2.2]. Additional subfunctions, such as clean() and dummy_pad(), are also defined in the Melbourne Shuffle paper but are omitted here, as they are entirely unobservable to an adversary and do not affect the algorithm's obliviousness.

The function Shuffle_Pass() operates in two phases [Ohrimenko et al., 2014, Section 4.2]. The first phase, implemented as the outer loop with two nested inner loops, is the distribution phase, which moves elements from I (the input array) into T (a temporary array). The second phase, also implemented as an outer loop, is the clean-up phase, which clears T and writes the results to O (the output array). In practice, the distribution phase may fail; however, for our reasoning, we consider an idealized version that never fails, assuming that the target permutation π is suitably chosen before Shuffle_Pass(I, T, π, O) is called.

function GETRANGE(S, loc, l)
  $a \leftarrow []$
  $i \leftarrow 0$
  while$_D$ $i < l$ **do**
    $a \leftarrow a + S[\mathsf{loc} + i]$
  return a
end function

function PUTRANGE(S, loc, a)
  $i \leftarrow 0$
  while$_D$ $i < \mathsf{size}(a)$ **do**
    $S[\mathsf{loc} + i] \leftarrow \mathsf{enc}(a[i])$
end function

function SHUFFLE_PASS(I, T, π, O)
  $\mathsf{max\_elems} \leftarrow p * \log(n)$
  $\mathsf{num\_buckets} \leftarrow \sqrt{n}$
  $\mathsf{id}_I \leftarrow 0$
  while$_D$ $\mathsf{id}_I < \mathsf{num\_buckets}$ **do**
    $\mathsf{bucket}_M \leftarrow \mathsf{getRange}(I, \mathsf{id}_I * \sqrt{n}, \sqrt{n})$
    $\mathsf{rev\_bucket}_M \leftarrow \mathsf{empty\_map}()$
    $\mathsf{id}_e \leftarrow 0$
    while$_D$ $\mathsf{id}_e < \sqrt{n}$ **do**
      $\mathsf{xv} \leftarrow \mathsf{dec}(\mathsf{bucket}_M[\mathsf{id}_e])$
      $\mathsf{id}_T \leftarrow \lfloor \pi(\mathsf{xv}[0])) / \sqrt{n} \rfloor$
      $\mathsf{rev\_bucket}_M[\mathsf{id}_T].\mathsf{add}(\mathsf{xv})$
      $\mathsf{id}_e \leftarrow \mathsf{id}_e + 1$
    $\mathsf{id}_T \leftarrow 0$
    while$_D$ $\mathsf{id}_T < \mathsf{num\_buckets}$ **do**
      if$_D$ $\mathsf{size}(\mathsf{rev\_bucket}_M[\mathsf{id}_T] > \mathsf{max\_elems})$ **then** $\mathsf{fail}()$
      $\mathsf{rev\_bucket}_M[\mathsf{id}_T] \leftarrow \mathsf{dummy\_pad}(\mathsf{rev\_bucket}_M[\mathsf{id}_T, \mathsf{max\_elems}])$
      $\mathsf{putRange}(T, \mathsf{id}_T * \sqrt{n} * \mathsf{max\_elems} + \mathsf{max\_elems} * \mathsf{id}_I, \mathsf{rev\_bucket}_M[\mathsf{id}_T])$
      $\mathsf{id}_T \leftarrow \mathsf{id}_T + 1$
    $\mathsf{id}_I \leftarrow \mathsf{id}_I + 1$
  $\mathsf{id}_T \leftarrow 0$
  while$_D$ $\mathsf{id}_T < \mathsf{num\_buckets}$ **do**
    $\mathsf{bucket}_M \leftarrow \mathsf{getRange}(T, \mathsf{id}_T * \sqrt{n} * \mathsf{max\_elems}, \sqrt{n} * \mathsf{max\_elems})$
    $\mathsf{bucket}_M \leftarrow \mathsf{clean}(\mathsf{bucket}_M)$
    $\mathsf{putRange}(O, \mathsf{id}_T * \sqrt{n}, \mathsf{bucket}_M)$
    $\mathsf{id}_T \leftarrow \mathsf{id}_T + 1$
end function

FIGURE A.1: The Melbourne Shuffle Subfunctions

Let $\mathsf{get}(S, \mathsf{loc}, i) = [(\text{“Read”}, S, \mathsf{loc}+0), (\text{“Read”}, S, \mathsf{loc}+1), \cdots, (\text{“Read”}, S, \mathsf{loc}+i-1)]$
Let $\mathsf{put}(S, \mathsf{loc}, i) = [(\text{“Write”}, S, \mathsf{loc}+0, \bot), \cdots, (\text{“Write”}, S, \mathsf{loc}+i-1, \bot)]$

function GETRANGE'(S, loc, l)
$\{\mathsf{Ct}(\mathsf{Trace} = T)\}$
  $a \leftarrow []$
  $i \leftarrow 0$
$\{\mathsf{Ct}(\mathsf{Trace} = T \wedge a = [] \wedge i = 0)\}$
  while$_D$ $i < l$ **do**
  $\{\mathsf{Ct}(\mathsf{Trace} = T +\!\!+ \mathsf{get}(S, \mathsf{loc}, i) \wedge i \leq l)\}$
    $a \leftarrow a + S[\mathsf{loc}+i]$
    $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Read”}, S, \mathsf{loc}+i)$
  return a
$\{\mathsf{Ct}(\mathsf{Trace} = T +\!\!+ \mathsf{get}(S, \mathsf{loc}, l) \wedge i = l)\}$
end function

function PUTRANGE'(S, loc, a)
$\{\mathsf{Ct}(\mathsf{Trace} = T)\}$
  $i \leftarrow 0$
  while$_D$ $i < \mathsf{size}(a)$ **do**
  $\{\mathsf{Ct}(\mathsf{Trace} = T +\!\!+ \mathsf{put}(S, \mathsf{loc}, i) \wedge i \leq \mathsf{size}(a))\}$
    $S[\mathsf{loc}+i] \leftarrow \mathsf{enc}(a[i])$
    $\mathsf{Trace} \leftarrow \mathsf{Trace} + (\text{“Write”}, S, \mathsf{loc}+i, \bot)$
$\{\mathsf{Ct}(\mathsf{Trace} = T +\!\!+ \mathsf{put}(S, \mathsf{loc}, \mathsf{size}(a)) \wedge i = \mathsf{size}(a))\}$
end function

FIGURE A.2: Transformations and Specifications of two low-level functions

Transformation Since only getRange() and putRange() directly access observable locations, we introduce ghost code exclusively within these two subfunctions, as illustrated in Fig. A.2. To meet the transformation requirements specified in Definition 4.2, we defer the encryption of secret data until immediately before it is written to the public locations in putRange() so that the transformation can identify the ciphertext and record them as $\bot$ in the simplified transformation.

Obliviousness Verification The verification of getRange() and putRange() is presented in Fig. A.2. These functions perform sequential reads and writes, respectively, with their memory access patterns determined by get() and put(). The correctness of these patterns is established by introducing appropriate loop invariants.

The obliviousness verification of Shuffle_Pass() is presented in Fig. A.3. This function contains one large nested loop followed by a single-layer loop. To facilitate verification,

we define several auxiliary functions at the beginning of the figure to specify the loop invariants for each loop that may access public locations. Since the loops are nested, the auxiliary functions are likewise nested to mirror this structure. Finally, the function TS() captures the overall memory access pattern produced by Shuffle_Pass().

The precondition of this function consists of three components:

- Trace $= X$ specifies that the initial state may already include a memory access history, stored in X.
- size$(I) = n =$ size(Set(I)) asserts that the size of the input array I is n, and that all elements of I are unique. This uniqueness requirement ensures the validity of the permutation functions π and π_1. If duplicates are present, they can be padded with their indices to enforce uniqueness.
- The remaining part asserts that the permutation function π_1 correctly describes the input I, and that the target permutation π will not cause a failure (i.e. $\pi \in S_p(\pi_1)$, as defined in Section 5.4).

As Section 5.4 specified, the comprehensive verification of this function encompasses both obliviousness—with respect to Trace, as illustrated in Fig. A.3—and correctness, which establishes that π correctly describes the output and that the output elements are unique. In this section, we present only the obliviousness verification; the correctness proof can be found in [Ohrimenko et al., 2014, Section 4.2].

Let $\mathsf{inner}(T, i, t, n, m) = \mathsf{put}(T, (0 * \sqrt{n} + i) * m, m) \mathbin{+\!\!+} \mathsf{put}(T, (1 * \sqrt{n} + i) * m, m) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathsf{put}(T, ((t-1) * \sqrt{n} + i) * m, m)$

Let $\mathsf{dist}(I, T, i, n, m) = \mathsf{get}(I, 0 * \sqrt{n}, \sqrt{n}) \mathbin{+\!\!+} \mathsf{inner}(T, 0, \mathsf{num\_buckets}, n, m) \mathbin{+\!\!+}$
$\mathsf{get}(I, 1 * \sqrt{n}, \sqrt{n}) \mathbin{+\!\!+} \mathsf{inner}(T, 1, \mathsf{num\_buckets}, n, m) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+}$
$\mathsf{get}(I, (i-1) * \sqrt{n}, \sqrt{n}) \mathbin{+\!\!+} \mathsf{inner}(T, i-1, \mathsf{num\_buckets}, n, m)$

Let $\mathsf{cln}(T, O, i, n, m) = \mathsf{get}(T, 0 * \sqrt{n} * m, \sqrt{n} * m) \mathbin{+\!\!+} \mathsf{put}(O, 0 * \sqrt{n}, \sqrt{n}) \mathbin{+\!\!+}$
$\mathsf{get}(T, 1 * \sqrt{n} * m, \sqrt{n} * m) \mathbin{+\!\!+} \mathsf{put}(O, 1 * \sqrt{n}, \sqrt{n}) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+}$
$\mathsf{get}(T, (i-1) * \sqrt{n} * m, \sqrt{n} * m) \mathbin{+\!\!+} \mathsf{put}(O, (i-1) * \sqrt{n}, \sqrt{n})$

Let $\mathsf{TS}(I, T, O, n) = \mathsf{dist}(I, T, \sqrt{n}, n, p * \log(n)) \mathbin{+\!\!+} \mathsf{cln}(T, O, \sqrt{n}, n, p * \log(n))$

function SHUFFLE_PASS'(I, T, π, O)
$\{\mathsf{Ct}(\mathsf{Trace} = X \wedge \mathsf{size}(I) = n = \mathsf{size}(\mathsf{Set}(\mathsf{dec}(I))) \wedge (\forall v \in I.\ \ I[\pi_1(v)] = v) \wedge \pi \in S_p(\pi_1))\}$
  $\mathsf{max\_elems} \leftarrow p * \log(n)$
  $\mathsf{num\_buckets} \leftarrow \sqrt{n}$
  $\mathsf{id}_I \leftarrow 0$
  while$_D$ $\mathsf{id}_I < \mathsf{num\_buckets}$ **do**
  $\{\mathsf{Ct}(\mathsf{Trace} = X \mathbin{+\!\!+} \mathsf{dist}(I, T, \mathsf{id}_I, n, \mathsf{max\_elems}))\}$
    $\mathsf{bucket}_M \leftarrow \mathsf{getRange}'(I, \mathsf{id}_I * \sqrt{n}, \sqrt{n})$
  $\{\mathsf{Ct}(\mathsf{Trace} = X \mathbin{+\!\!+} \mathsf{dist}(I, T, \mathsf{id}_I, n, \mathsf{max\_elems}) \mathbin{+\!\!+} \mathsf{get}(I, \mathsf{id}_I * \sqrt{n}, \sqrt{n}))\}$
    $\mathsf{rev\_bucket}_M \leftarrow \mathsf{empty\_map}()$
    $\mathsf{id}_e \leftarrow 0$
    while$_D$ $\mathsf{id}_e < \sqrt{n}$ **do**
      $\mathsf{xv} \leftarrow \mathsf{dec}(\mathsf{bucket}_M[\mathsf{id}_e])$
      $\mathsf{id}_T \leftarrow \lfloor \pi(\mathsf{xv}[0])) / \sqrt{n} \rfloor$
      $\mathsf{rev\_bucket}_M[\mathsf{id}_T].\mathsf{add}(\mathsf{xv})$
      $\mathsf{id}_e \leftarrow \mathsf{id}_e + 1$
    $\mathsf{id}_T \leftarrow 0$
    while$_D$ $\mathsf{id}_T < \mathsf{num\_buckets}$ **do**
    $\{\mathsf{Ct}(\mathsf{Trace} = X \mathbin{+\!\!+} \mathsf{dist}(I, T, \mathsf{id}_I, n, \mathsf{max\_elems}) \mathbin{+\!\!+} \mathsf{get}(I, \mathsf{id}_I * \sqrt{n}, \sqrt{n})$
    $\mathbin{+\!\!+} \mathsf{inner}(T, \mathsf{id}_I, \mathsf{id}_T, n, \mathsf{max\_elems}))\}$
      if$_D$ $\mathsf{size}(\mathsf{rev\_bucket}_M[\mathsf{id}_T] > \mathsf{max\_elems})$ **then** $\mathsf{fail}()$
      $\mathsf{rev\_bucket}_M[\mathsf{id}_T] \leftarrow \mathsf{dummy\_pad}(\mathsf{rev\_bucket}_M[\mathsf{id}_T, \mathsf{max\_elems}])$
      $\mathsf{putRange}'(T, \mathsf{id}_T * \sqrt{n} * \mathsf{max\_elems} + \mathsf{max\_elems} * \mathsf{id}_I, \mathsf{rev\_bucket}_M[\mathsf{id}_T])$
      $\mathsf{id}_T \leftarrow \mathsf{id}_T + 1$
    $\mathsf{id}_I \leftarrow \mathsf{id}_I + 1$
$\{\mathsf{Ct}(\mathsf{Trace} = X \mathbin{+\!\!+} \mathsf{dist}(I, T, \mathsf{num\_buckets}, n, \mathsf{max\_elems}))\}$
  $\mathsf{id}_T \leftarrow 0$
  while$_D$ $\mathsf{id}_T < \mathsf{num\_buckets}$ **do**
  $\{\mathsf{Ct}(\mathsf{Trace} = X \mathbin{+\!\!+} \mathsf{dist}(I, T, \mathsf{num\_buckets}, n, \mathsf{max\_elems}) \mathbin{+\!\!+} \mathsf{cln}(T, O, \mathsf{id}_T, n, \mathsf{max\_elems}))\}$
    $\mathsf{bucket}_M \leftarrow \mathsf{getRange}'(T, \mathsf{id}_T * \sqrt{n} * \mathsf{max\_elems}, \sqrt{n} * \mathsf{max\_elems})$
    $\mathsf{bucket}_M \leftarrow \mathsf{clean}(\mathsf{bucket}_M)$
    $\mathsf{putRange}'(O, \mathsf{id}_T * \sqrt{n}, \mathsf{bucket}_M)$
    $\mathsf{id}_T \leftarrow \mathsf{id}_T + 1$
$\{\mathsf{Ct}(\mathsf{Trace} = X \mathbin{+\!\!+} \mathsf{TS}(\text{“}I\text{”}, \text{“}T\text{”}, \text{“}O\text{”}, n))$
end function

FIGURE A.3: Transformation and Verification of Shuffle Pass

Bibliography

Pengbo Yan. Formally verifying the security of probabilistic oblivious algorithms, September 2025. URL https://doi.org/10.5281/zenodo.17034603.

Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, February 2012. The Internet Society. Full text available from NDSS Symposium (open access).

Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, August 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho.

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015a. doi: 10.1109/SP.2015.43.

Sajin Sasy and Olga Ohrimenko. Oblivious sampling algorithms for private data analysis. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2019. Curran Associates Inc.

Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics

platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng.

Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptol. ePrint Arch.*, page 431, 2014. URL http://eprint.iacr.org/2014/431.

Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 311–324, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324779. doi: 10.1145/2508859.2516692. URL https://doi.org/10.1145/2508859.2516692.

Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376, 2015b. doi: 10.1109/SP.2015.29.

Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *J. ACM*, 65(4), apr 2018. ISSN 0004-5411. doi: 10.1145/3177872. URL https://doi.org/10.1145/3177872.

Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming*, pages 556–567, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-43951-7.

David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology-ICISC 2005: 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers 8*, pages 156–168. Springer, 2006.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, volume 16, pages 53–70, 2016.

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–189, 2019.

Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156, USA, 2012. Society for Industrial and Applied Mathematics.

Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Automata, Languages and Programming*, pages 576–587, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22012-8.

Jeongseok Son, Griffin Prechter, Rishabh Poddar, Raluca Ada Popa, and Koushik Sen. ObliCheck: Efficient verification of oblivious algorithms with unobservable state. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2219–2236. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/son.

Gilles Barthe, Justin Hsu, and Kevin Liao. A probabilistic separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, December 2019. ISSN 2475-1421. doi: 10.1145/3371123. URL https://doi.org/10.48550/arXiv.1907.10708.

Qianchuan Ye and Benjamin Delaware. Oblivious algebraic data types. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi: 10.1145/3498713. URL https://doi.org/10.1145/3498713.

David Darais, Ian Sweet, Chang Liu, and Michael Hicks. A language for probabilistically oblivious computation. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371118. URL https://doi.org/10.1145/3371118.

Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. Cryptology ePrint Archive, Paper 2019/274, 2019. URL https://eprint.iacr.org/2019/274. https://eprint.iacr.org/2019/274.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12 (10):576–580, oct 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817.

Nick Benton. Simple relational correctness proofs for static analyses and program transformations. *SIGPLAN Not.*, 39(1):14–25, jan 2004. ISSN 0362-1340. doi: 10.1145/982962.964003. URL https://doi.org/10.1145/982962.964003.

Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for c11 concurrency. *SIGPLAN Not.*, 48(10):867–884, October 2013. ISSN 0362-1340. doi: 10.1145/2544173.2509532. URL https://doi.org/10.1145/2544173.2509532.

Krzysztof R. Apt. Ten years of hoare's logic: A survey—part ii: Nondeterminism. *Theoretical Computer Science*, 43(2-3):177–210, 1986. doi: 10.1016/0304-3975(86)90068-0.

Peter W O'Hearn and David J Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: the semantics of bi. *Theoretical Computer Science*, 315(1):257–305, 2004. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2003.11.020. URL https://www.sciencedirect.com/science/article/pii/S0304397503006248. Mathematical Foundations of Programming Semantics.

D. Galmiche, D. Méry, and D. Pym. The semantics of BI and resource tableaux. *Mathematical Structures in Computer Science*, 15(6):1033–1088, 2005. doi: 10.1017/S0960129505004858.

Claude E Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949.

Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography Second Edition*. Chapman, Hall/CRC, 2nd edition, 2014. ISBN 1466570261.

Salil Vadhan. Lecture 3: Perfect secrecy. https://people.seas.harvard.edu/~salil/cs127/fall13/lec3.pdf, 2013.

Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984. doi: 10.1016/0022-0000(84)90070-9.

Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer, 1998. doi: 10.1007/BFb0055718.

Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000. doi: 10.1007/3-540-44448-3_1.

David A. McGrew and John Viega. The galois/counter mode of operation (gcm). IETF RFC 5116, 2007. URL https://datatracker.ietf.org/doc/html/rfc5116.

Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. In *STOC*, pages 408–416, 1987.

Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference (AFIPS)*, pages 307–314. ACM, 1968. doi: 10.1145/1468075.1468121.

Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. In *Journal of the ACM*, volume 43, pages 431–473, 1996.

Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 502–519, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14623-7.

Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1262–1277, 2010.

Tiancheng Wang, Xiao Shaun Wang, Jonathan Katz, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *CCS*, pages 850–861, 2015.

Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. *J. ACM*, 70(1), December 2022. ISSN 0004-5411. doi: 10.1145/3566049. URL https://doi.org/10.1145/3566049.

Saba Eskandarian and Matei Zaharia. Oblidb: oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2):169–183, October 2019. ISSN 2150-8097. doi: 10.14778/3364324.3364331. URL https://doi.org/10.14778/3364324.3364331.

Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 215–226, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329576. doi: 10.1145/2660267.2660314. URL https://doi.org/10.1145/2660267.2660314.

Michael T. Goodrich and Michael Mitzenmacher. Cache-oblivious dictionaries and multimaps with negligible failure probability. In *Algorithms and Data Structures (WADS)*, pages 271–282, 2012.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Probabilistic relational hoare logics for computer-aided security proofs. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, volume 7358 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2012. doi: 10.1007/978-3-642-31424-7_9.

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, pages 146–166, 2014a.

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3), November 2013. ISSN 0164-0925. doi: 10.1145/2492061. URL https://doi.org/10.1145/2492061.

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving differential privacy via probabilistic couplings. In *Proceedings of the 31st*

Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, page 749–758, New York, NY, USA, 2016a. Association for Computing Machinery. ISBN 9781450343916. doi: 10.1145/2933575.2934554. URL https://doi.org/10.1145/2933575.2934554.

Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Advanced probabilistic couplings for differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 55–67, New York, NY, USA, 2016b. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978391. URL https://doi.org/10.1145/2976749.2978391.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

Shubham Sahai, Pramod Subramanyan, and Rohit Sinha. Verification of quantitative hyperproperties using trace enumeration relations. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*, pages 201–224. Springer, 2020.

Hannah Leung, Talia Ringer, and Christopher W Fletcher. Towards formally verified path oram in coq. 2023. Available online: https://dependenttyp.es/pdf/oramproposal.pdf.

Janez Ignacij Jereb and Alex Simpson. Safety, relative tightness and the probabilistic frame rule, 2025. URL https://arxiv.org/abs/2506.01626.

John M. Li, Amal Ahmed, and Steven Holtzen. Lilac: A modal separation logic for conditional probability. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi: 10.1145/3591226. URL https://doi.org/10.1145/3591226.

John M. Li, Jon Aytac, Philip Johnson-Freyd, Amal Ahmed, and Steven Holtzen. A nominal approach to probabilistic separation logic. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706608. doi: 10.1145/3661814.3662135. URL https://doi.org/10.1145/3661814.3662135.

Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. A bunched logic for conditional independence. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '21, New York, NY, USA, 2021a. Association for Computing Machinery. ISBN 9781665448956. doi: 10.1109/LICS52264.2021.9470712. URL https://doi.org/10.1109/LICS52264.2021.9470712.

Ugo Dal Lago, Davide Davoli, and Bruce M. Kapron. On separation logic, computational independence, and pseudorandomness (extended version), 2024. URL https://arxiv.org/abs/2405.11987.

Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. A separation logic for negative dependence. *CoRR*, abs/2111.14917, 2021b. URL https://arxiv.org/abs/2111.14917.

Jialu Bao, Emanuele D'Osualdo, and Azadeh Farzan. Bluebell: An alliance of relational lifting and independence for probabilistic reasoning. *Proc. ACM Program. Lang.*, 9 (POPL):Article 58, 2025. doi: 10.1145/3704894.

Joseph Tassarotti and Robert Harper. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.*, 3(POPL):64:1–64:30, 2019. doi: 10.1145/3290377.

Shing Hin Ho, Nicolas Wu, and Azalea Raad. Bayesian separation logic: A logical foundation and axiomatic semantics for probabilistic programming, 2025. URL https://arxiv.org/abs/2507.15530.

Philipp Schröer, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. A deductive verification infrastructure for probabilistic programs. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023. doi: 10.1145/3622870. URL https://doi.org/10.1145/3622870.

Kevin Batz, Moritz Götz, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. Quantitative separation logic: A logic for reasoning about probabilistic pointer programs. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019)*, pages 1–32. ACM, 2019. doi: 10.1145/3290355.

Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2005. ISBN 978-0387206901.

Luis María Ferrer Fioriti and Holger Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 489–501, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2677001. URL https://doi.org/10.1145/2676726.2677001.

Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. A new proof rule for almost-sure termination. *Proc. ACM Program. Lang.*, 2 (POPL), dec 2017. doi: 10.1145/3158121. URL https://doi.org/10.1145/3158121.

Stéphanie Delaune and Lucca Hirschi. A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols. *Journal of Logical and Algebraic Methods in Programming*, 87:127–144, 2017. ISSN 2352-2208. doi: https://doi.org/10.1016/j.jlamp.2016.10.005. URL https://www.sciencedirect.com/science/article/pii/S235222081630133X.

D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. Inf. Theor.*, 29(2):198–208, September 2006. ISSN 0018-9448. doi: 10.1109/TIT.1983.1056650. URL https://doi.org/10.1109/TIT.1983.1056650.

Martin Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptol.*, 20(3):395, July 2007. ISSN 0933-2790.

Michael Backes, Birgit Pfitzmann, and Michael Waidner. A survey of symbolic methods in computational analysis of cryptographic protocols. *Journal of Automated Reasoning*, 45(4):499–545, 2010. doi: 10.1007/s10817-010-9187-9.

Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

Jonathan Herzog. A computational interpretation of dolev-yao adversaries. *Theoretical Computer Science*, 340:57–81, 2005.

Mathieu Baudet, Véronique Cortier, and Steve Kremer. Computationally sound implementations of equational theories against passive adversaries. *Information and Computation*, 207(4):496–520, 2009.

Martín Abadi, Mathieu Baudet, and Bogdan Warinschi. Guessing attacks and the computational soundness of static equivalence. In *FoSSaCS*, volume 3921 of *LNCS*, pages 398–412, 2006.

Peeter Laud. Semantics and program analysis of computationally secure information flow. In *ESOP*, volume 2028 of *LNCS*, pages 77–91, 2001.

Judicaël Courant, Cristian Ene, and Yassine Lakhnech. Computationally sound typing for non-interference: The case of deterministic encryption. In *FSTTCS*, volume 4855 of *LNCS*, pages 364–375, 2007.

Anupam Datta, Ante Derek, John C. Mitchell, Vitaly Shmatikov, and Mathieu Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *ICALP*, volume 3580 of *LNCS*, pages 16–29, 2005.

Anupam Datta, Ante Derek, John C. Mitchell, and Bogdan Warinschi. Computationally sound compositional logic for key exchange protocols. In *CSFW*, pages 321–334, 2006.

Prateek Gupta and Vitaly Shmatikov. Towards computationally sound symbolic analysis of key exchange protocols. In *FMSE*, pages 23–32, 2005.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 978-3-540-43376-7. doi: 10.1007/3-540-45949-9. URL https://doi.org/10.1007/3-540-45949-9.

Pengbo Yan, Toby Murray, Olga Ohrimenko, Van-Thuan Pham, and Robert Sison. Combining classical and probabilistic independence reasoning to verify the security of oblivious algorithms. In André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Formal Methods*, pages 188–205, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-71162-6.

J. I. den Hartog. Verifying probabilistic programs using a hoare like logic. In P. S. Thiagarajan and Roland Yap, editors, *Advances in Computing Science — ASIAN'99*, pages 113–125, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-46674-1.

Robert Rand and S. Zdancewic. Vphl: A verified partial-correctness logic for probabilistic programs. *Electronic Notes in Theoretical Computer Science*, 319:351–367, 12 2015. doi: 10.1016/j.entcs.2015.12.021.

Pengbo Yan. Combining classical and probabilistic independence reasoning to verify the security of oblivious algorithms, June 2024. URL https://doi.org/10.5281/zenodo.12755500.

Mário S Alvim, Konstantinos Chatzikokolakis, Annabelle McIver, Carroll Morgan, Catuscia Palamidessi, and Geoffrey Smith. *The Science of Quantitative Information Flow*. Springer, 2020.

Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 417–431, 2016. doi: 10.1109/CSF.2016.36.

Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 308–318, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978318. URL https://doi.org/10.1145/2976749.2978318.

H. Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models, 2018. URL https://arxiv.org/abs/1710.06963.

Lei Yu, Ling Liu, Calton Pu, Mehmet Emre Gursoy, and Stacey Truex. Differentially private model publishing for deep learning. *CoRR*, abs/1904.02200, 2019. URL http://arxiv.org/abs/1904.02200.

Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, page 13–24, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310918. doi: 10.1145/2133601.2133604. URL https://doi.org/10.1145/2133601.2133604.

Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, page 139–148, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938107. doi: 10.1145/1455770.1455790. URL https://doi.org/10.1145/1455770.1455790.

T.-H. Hubert Chan, Kai-Min Chung, Bruce Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. *J. ACM*, 69(4), aug 2022. ISSN 0004-5411. doi: 10.1145/3555984. URL https://doi.org/10.1145/3555984.

Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4):211–407, aug 2014. ISSN 1551-305X. doi: 10.1561/0400000042. URL https://doi.org/10.1561/0400000042.

Danfeng Zhang and Daniel Kifer. Lightdp: Towards automating differential privacy proofs. *SIGPLAN Not.*, 52(1):888–901, jan 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009884. URL https://doi.org/10.1145/3093333.3009884.

Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. Coupled relational symbolic execution for differential privacy. In *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings*, page 207–233, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-72018-6. doi: 10.1007/978-3-030-72019-3_8. URL https://doi.org/10.1007/978-3-030-72019-3_8.

Gilles Barthe, Marco Gaboardi, Emilio Jesus Gallego Arias, Justin Hsu, Cesar Kunz, and Pierre-Yves Strub. Proving differential privacy in hoare logic. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 411–424, 2014b. doi: 10.1109/CSF.2014.36.

Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. Dp-finder: Finding differential privacy violations by sampling and optimization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 508–524, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243863. URL https://doi.org/10.1145/3243734.3243863.