ABSTRACT

Title of dissertation: TRACE OBLIVIOUS PROGRAM EXECUTION

Chang Liu, Doctor of Philosophy, 2016

Dissertation directed by: Professor Michael Hicks
Department of Computer Science, University of Maryland
and Professor Elaine Shi
Department of Computer Science, Cornell University

The big data era has dramatically transformed our lives; however, security incidents such as data breaches can put sensitive data (e.g. photos, identities, genomes) at risk. To protect users' data privacy, there is a growing interest in building *secure cloud computing systems*, which keep sensitive data inputs hidden, even from computation providers. Conceptually, secure cloud computing systems leverage cryptographic techniques (e.g., secure multiparty computation) and trusted hardware (e.g. secure processors) to instantiate a secure abstract machine consisting of a CPU and encrypted memory, so that an adversary cannot learn information through either the computation within the CPU or the data in the memory. Unfortunately, evidence has shown that side channels (e.g. memory accesses, timing, and termination) in such a secure abstract machine may potentially leak highly sensitive information, including cryptographic keys that form the root of trust for the secure systems.

This thesis broadly expands the investigation of a research direction called

trace oblivious computation, where programming language techniques are employed to prevent side channel information leakage. We demonstrate the feasibility of trace oblivious computation, by formalizing and building several systems, including GhostRider, which is a hardware-software co-design to provide a hardware-based trace oblivious computing solution, SCVM, which is an automatic RAM-model secure computation system, and ObliVM, which is a programming framework to facilitate programmers to develop applications. All of these systems enjoy formal security guarantees while demonstrating a better performance than prior systems, by one to several orders of magnitude.

TRACE OBLIVIOUS PROGRAM EXECUTION

by

Chang Liu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2016

Advisory Committee:
Professor Michael W. Hicks, Co-Chair/Co-Advisor
Professor Elaine Shi, Co-Chair/Co-Advisor
Professor Charalampos Papamanthou
Professor Zia Khan
Professor Lawrence C. Washington

Dedication

To my loving mother.

Acknowledgments

I am very fortunate to have spent three fantastic years at UMD. Throughout my journey to learn how to do research, many people have helped me. This dissertation would not have been possible without them.

First, I want to thank my advisors, Elaine Shi and Michael Hicks, for their guidance to this amazing academic world. Through collaborating with them, Elaine and Mike have set up examples for me with both precept and practice. I not only benefited from insightful technical discussions with them, but also learnt a lot about other aspects of research: how to pick an important problem to work on, how to criticize a work, especially the one from myself, and so on.

I have learnt from many people that a piece of research is an execution of a great taste: I have learnt from Elaine a great taste of choosing research problems and pushing them closer to perfection. I have learnt from Mike a great taste to develop elegant theories to explain phenomenon and to communicate the ideas precisely and concisely. I also have learnt from Jon Froehlich, my proposal committee member, a great taste of giving presentation, and I have learnt from Dawn Song, who I worked closely in my last year, a great taste of envisioning novel research directions. I am grateful that I can work with these fantastic researchers during my PhD life.

My committee members – my advisor, Babis Papamanthou, Zia Khan, and Larry Washington – have been extremely supportive. My thanks also go to all other members of the MC2 faculty. I am also fortunate to be surrounded by a diverse and inspiring group of (former and current) students at MC2.

I am fortunate to work with my collaborators. Many ideas in this dissertation cannot be executed so well without their help. They are my advisors, Xiao Wang, Natik Kayak, Yan Huang, Martin Maas, Austin Harris, Mohit Tiwari and Jonathan Katz without an order.

Now, I would like to thank some dear friends. Xi Chen, Yuening Hu, Ke Zhai, He He, and I have spent a wonderful summer at Microsoft Research at Redmond, which is the most enjoyable summer during my PhD life. I am fortunate to be friends of Yulu Wang and Shangfu Peng, who made my early graduate life happy and substantial. Qian Wu and Hang Hu are my old friends who always support me on both life and research. Xi Yi and Yu Zhang brought both happiness and insightful suggestions into my life. I am happy to become the introducer of the couple to meet each other.

Life at UMD would have been much more difficult without members of the administrative and technical staff. Jennifer Story has always been patient and helpful beyond the call of duty. Joe Webster has made resources available just as we needed it. Jodie Gray and Sharron McElroy have minimized bureaucracy when it comes to tax and money. Thank you all.

Last but in no way the least, I thank my family who have given me unconditional love throughout my life. My mother, to whom I owe a lot due to my academic career, has been supporting me altruistically for me to pursuit my dream. Words are not enough to express my gratitude for them. Finally, I thank Danqi Hu, who has always been there for me as an inspiring partner. Thank you for steering me to a better self.

Table of Contents

List of Figures viii

1 Introduction 1
- 1.1 Beyond Oblivious RAM . . . 2
- 1.2 A hardware-software co-design for ensuring memory trace obliviousness 4
- 1.3 Automatic RAM-model secure computation . . . 5
- 1.4 A programming framework for secure computation . . . 7
- 1.5 Our Results and Contributions. . . . 8

2 Memory Trace Obliviousness: Basic Setting 11
- 2.1 Threat Model . . . 12
- 2.2 Motivating Examples . . . 13
- 2.3 Approach Overview . . . 14
- 2.4 Memory trace obliviousness by typing . . . 16
  - 2.4.1 Operational semantics . . . 17
  - 2.4.2 Memory trace obliviousness . . . 21
  - 2.4.3 Security typing . . . 22
  - 2.4.4 Examples . . . 27
- 2.5 Compilation . . . 29
  - 2.5.1 Type checking source programs . . . 30
  - 2.5.2 Allocating variables to ORAM banks . . . 32
  - 2.5.3 Inserting padding instructions . . . 33
- 2.6 Evaluation . . . 36
  - 2.6.1 Simulation Results . . . 37
- 2.7 Conclusion Remarks . . . 38

3 GhostRider: A Compiler-Hardware Approach 39
- 3.1 Introduction . . . 39
  - 3.1.1 Our Results and Contributions. . . . 40
- 3.2 Architecture and Approach . . . 42
  - 3.2.1 Motivating example . . . 42
  - 3.2.2 Threat model . . . 43

3.2.3 Architectural Overview . . . 44
3.3 Formalizing the target language . . . 46
3.3.1 Instruction set . . . 47
3.3.2 Example . . . 49
3.4 Security by typing . . . 50
3.4.1 Memory Trace Obliviousness . . . 50
3.4.2 Typing: Preliminaries . . . 53
3.4.3 Type rules . . . 58
3.4.4 Security theorem . . . 63
3.5 Compilation . . . 63
3.5.1 Source Language . . . 63
3.5.2 Memory bank allocation . . . 65
3.5.3 Basic compilation . . . 66
3.5.4 Padding and register allocation . . . 68
3.6 Hardware Implementation . . . 69
3.7 Empirical Evaluation . . . 72
3.8 Conclusion . . . 78

4 RAM-model Secure Computation 79
4.1 Technical Highlights . . . 81
4.2 Background: RAM-Model Secure Computation . . . 83
4.3 Technical Overview: Compiling for RAM-Model Secure Computation 87
4.3.1 Instruction-Trace Obliviousness . . . 87
4.3.2 Memory-Trace Obliviousness . . . 88
4.3.3 Mixed-Mode Execution . . . 89
4.3.4 Example: Dijkstra's Algorithm . . . 90
4.4 SCVM Language . . . 92
4.4.1 Syntax . . . 94
4.4.2 Semantics . . . 95
4.4.3 Security . . . 104
4.4.4 Type System . . . 106
4.4.5 From SCVM Programs to Secure Protocols . . . 109
4.5 Compilation . . . 111
4.6 Evaluation . . . 115
4.6.1 Evaluation Methodology . . . 116
4.6.2 Comparison with Automated Circuits . . . 118
4.6.2.1 Repeated sublinear-time queries . . . 119
4.6.2.2 Faster one-time executions . . . 121
4.6.3 Comparison with RAM-SC Baselines . . . 124
4.7 Conclusions . . . 128

5 ObliVM: A Programming Framework for Secure Computation 129
5.1 ObliVM Overview and Contributions . . . 130
5.1.1 Applications and Evaluation . . . 134
5.1.2 Threat Model, Deployment, and Scope . . . 135

5.2 Programming Language and Compiler . . . 136
5.2.1 Language features for expressiveness and efficiency . . . 136
5.2.2 Language features for security . . . 140
5.3 User-Facing Oblivious Programming Abstractions . . . 145
5.3.1 MapReduce Programming Abstractions . . . 145
5.3.2 Programming Abstractions for Data Structures . . . 150
5.3.3 Loop Coalescing and New Oblivious Graph Algorithms . . . 153
5.4 Implementing Rich Circuit Libraries . . . 158
5.4.1 Case Study: Basic Arithmetic Operations . . . 158
5.4.2 Case Study: Circuit ORAM . . . 160
5.5 Evaluation . . . 162
5.5.1 Back End Implementation . . . 162
5.5.2 Metrics and Experiment Setup . . . 162
5.5.3 Comparison with Previous Automated Approaches . . . 163
5.5.4 ObliVM vs. Hand-Crafted Solutions . . . 168
5.5.5 End-to-End Application Performance . . . 170
5.6 Conclusion . . . 175

6 Conclusion Remarks and Future Directions 176
6.1 Summary . . . 176
6.2 Future Direction . . . 177
6.2.1 Verifying Hardware ORAM Implementation . . . 177
6.2.2 Parallel Trace Oblivious Execution . . . 177
6.2.3 Differentially Privately Oblivious Execution . . . 178

A Proof of Theorem 1 179
A.1 Trace equivalence and lemmas . . . 179
A.2 Lemmas on trace pattern equivalence . . . 184
A.3 Proof of memory trace obliviousness . . . 185

B Proof of Theorem 2 195

C Proof of Theorem 3 236
C.1 Proof of Theorem 4 . . . 244

D The hybrid protocol and the proof of Theorem 5 246

Bibliography 256

List of Figures

2.1 Language syntax of $\mathcal{L}_{\text{basic}}$ . . . 17
2.2 Auxiliary syntax and functions for semantics in $\mathcal{L}_{\text{basic}}$ . . . 17
2.3 Operational semantics of $\mathcal{L}_{\text{basic}}$ . . . 18
2.4 Trace equivalence in $\mathcal{L}_{\text{basic}}$ . . . 22
2.5 Auxiliary syntax and functions for typing in $\mathcal{L}_{\text{basic}}$ . . . 23
2.6 Typing for $\mathcal{L}_{\text{basic}}$ . . . 24
2.7 Trace pattern equivalence in $\mathcal{L}_{\text{basic}}$ . . . 27
2.8 **Finding a short padding sequence using the greatest common subsequence algorithm.** An example with two abstract traces $\mathbf{T}_t = [T_1; T_2; T_3; T_4; T_5]$ and $\mathbf{T}_f = [T_1; T_3; T_2; T_4]$. One greatest common subsequence as shown is $[T_1; T_2; T_4]$. A shortest common super-sequence of the two traces is $\mathbf{T}_{tf} = [T_1; T_3; T_2; T_3; T_4; T_5]$. . . . 34
2.9 Simulation Results for Strawman, Opt 1, and Opt 2. . . . 37

3.1 Motivating source program of GhostRider. . . . 43
3.2 GhostRider architecture. . . . 44
3.3 Syntax for $\mathcal{L}_{\text{GhostRider}}$ language, comprising (1) **ldb** and **stb** instructions that move data blocks between scratchpad and a specific ERAM or ORAM bank, and (2) scratchpad-to-register moves and standard RISC instructions. . . . 47
3.4 $\mathcal{L}_{\text{GhostRider}}$ code implementing (part of) Figure 3.1 . . . 49
3.5 Symbolic values, labels, auxiliary judgments and functions . . . 53
3.6 Trace patterns and their equivalence in $\mathcal{L}_{\text{GhostRider}}$ . . . 56
3.7 Security Type System for $\mathcal{L}_{\text{GhostRider}}$ (Part 1) . . . 57
3.8 Security Type System for $\mathcal{L}_{\text{GhostRider}}$ (Part 2) . . . 60
3.9 Security Type System for $\mathcal{L}_{\text{GhostRider}}$ (Part 3) . . . 62
3.10 Simulator-based execution time results of GhostRider. . . . 73
3.11 Legends of Figure 3.10 . . . 74
3.12 FPGA based execution time results: Slowdown of `Baseline` and `Final` versions compared to non-secure version of the program. Note that unlike Figure 3.10, `Final` uses only a single ORAM bank and conflates ERAM and DRAM (cf. Section 3.6). . . . 76

4.1 Dijkstra's shortest distance algorithm in source (Part) . . . 90
4.2 . . . 91
4.3 Formal results in SCVM. . . . 93
4.4 Syntax of SCVM . . . 95
4.5 Auxiliary syntax and functions for SCVM semantics . . . 98
4.6 Operational semantics for expressions in SCVM . . . 101
4.7 Operational semantics for statements in SCVM (Part 1) . . . 102
4.8 Operational semantics for statements in SCVM (Part 2) . . . 103
4.9 Type System for SCVM . . . 107
4.10 SCVM vs. automated circuit-based approach (Binary Search) . . . 118
4.11 SCVM vs. hand-constructed linear scan circuit (Binary Search) . . . 119
4.12 Heap insertion in SCVM . . . 121
4.13 Heap extraction in SCVM . . . 122
4.14 KMP string matching for median n (fixing $m = 50$) in SCVM . . . 123
4.15 KMP string matching for large n (fixing $m = 50$) in SCVM . . . 124
4.16 Dijkstra's shortest-path algorithm's performance in SCVM . . . 125
4.17 Dijkstra's shortest-path algorithm's speedup of SCVM . . . 125
4.18 Aggregation over sliding windows's performance in SCVM . . . 126
4.19 Aggregation over sliding windows's speedup in SCVM . . . 126
4.20 SCVM's Savings by memory-trace obliviousness optimization (inverse permutation). the non-linearity (around 60) of the curve is due to the increase of the ORAM recursion level at that point. . . . 127
4.21 Savings by memory-trace obliviousness optimization (Dijkstra) . . . 127

5.1 **Streaming MapReduce in ObliVM-Lang.** See Section 5.3.1 for oblivious algorithms for the streaming MapReduce paradigm [36]. . . . 147
5.2 Oblivious stack by non-specialist programmers. . . . 151
5.3 Code by expert programmers to help non-specialists implement oblivious stack. . . . 152
5.4 **Loop coalescing.** The outer loop will be executed at most n times in total, the inner loop will be executed at most m times in total – over all iterations of the outer loop. A naive approach compiler would pad the outer and inner loop to n and m respectively, incurring $O(nm)$ cost. Our loop coalescing technique achieves $O(n + m)$ cost instead. 154
5.5 **Karatsuba multiplication in ObliVM-Lang.** . . . 159
5.6 **Part of our Circuit ORAM implementation (Type Definition) in ObliVM-Lang.** . . . 160
5.7 **Part of our Circuit ORAM implementation (ReadAndRemove) in ObliVM-Lang.** . . . 161
5.8 **Sources of speedup in comparison with state-of-the-art in 2012 [44]: an in-depth look.** . . . 166

B.1 Well formedness judgments for proof of Memory-Trace Obliviousness of $\mathcal{L}_{\text{GhostRider}}$ . . . 203
B.2 Symbolic Execution in $\mathcal{L}_{\text{GhostRider}}$ . . . 218

C.1 Operational semantics for sim_A 238
C.2 Operational semantics for statements in sim_A (part 1) 239
C.3 Operational semantics for statements in sim_A (part 2) 253
D.1 Hybrid Protocol $\pi^{\mathcal{G}}$ (Part I) 254
D.2 Hybrid Protocol $\Pi^{\mathcal{G}}(PartII)$ 255

Chapter 1: Introduction

Cloud computing allows users to outsource both data and computation to third-party cloud providers, and promises numerous benefits such as economies of scale, easy maintenance, and ubiquitous availability. These benefits, however, come at the cost of giving up physical control of one's computing infrastructure and private data. Privacy concerns have held back government agencies and businesses alike from outsourcing their computing infrastructure to the public cloud [18, 94].

To protect users' data privacy, there is a growing trend to build *secure cloud computing systems*, which enable computation over two or more parties' sensitive data, while revealing nothing more than the results to the participating parties, and nothing at all to any third party providers Conceptually, secure cloud computing systems leverage cryptographic techniques (e.g. secure multiparty computation) and trusted hardware (e.g. secure processors) to instantiate a "secure" abstract machine consisting of a CPU and encrypted memory, so that an adversary cannot learn information through either the computation within the CPU or the data in the memory. Unfortunately, evidence has shown that side channels (e.g., memory accesses, timing, and termination) in such a "secure" abstract machine may potentially leak highly sensitive information including cryptographic keys that form the

root of trust for the secure systems.

The thesis of this work is that programming language-based techniques— notably compilers and type systems— can be used to make programs secure, despite powerful adversaries with a fine-grained view of execution, by enforcing the property of *trace obliviousness*, which we define in this thesis.

1.1 Beyond Oblivious RAM

To cryptographically obfuscate memory access patterns, one can employ Oblivious RAM (ORAM) [33, 35], a cryptographic construction that makes memory address traces computationally indistinguishable from a random address trace. Encouragingly, recent theoretical breakthroughs [80, 84, 90] have allowed ORAM memory controllers to be built [28, 63] – these turn DRAM into oblivious, block-addressable memory banks.

The simplest way to deploy ORAM is to implement a single, large ORAM bank that contains all the code and data (assuming that the client can use standard PKI to safely transmit the code and data to a remote secure processor). A major drawback of this baseline approach is efficiency: every single memory-block access incurs the ORAM penalty which is roughly (poly-)logarithmic in the size of the ORAM [35, 80]. In practice, this translates to almost 100× additional bandwidth that, even with optimizations, incurs a ∼10× latency cost per block [28, 63]. Another issue is that, absent any padding, the baseline approach reveals the total number of memory accesses made by a program, which can leak information about the secret

inputs.

In practice, we observe that many programs are intrinsically *oblivious*, meaning that their execution traces observed by an adversary do not leak information. For example, let us consider a program to compute the summation of an array of integers, which are to be hidden from the adversary. This program will sequentially scan through the entire array, and assuming the array is encrypted in the memory, its memory access pattern will not leak any information about the secret integers stored in the array. In this case, ORAM is not necessary.

To enable such optimizations is not trivial. A program may not always be oblivious. For example, a program may mistakenly allocate an array whose access pattern indeed leaks sensitive information outside any ORAM banks. We take the approach to formalize the property of *memory trace obliviousness* (MTO), and design novel *type systems* to enforce that a well-typed program enjoys memory trace obliviousness.

A type system is a static analysis method to enforce a well-typed program to satisfy certain properties. Intuitively, the type systems we develop are an extension of language-based information flow systems [79], which is used to keep track of whether or not a variable used by a program contains sensitive information. Our extension further keeps track of whether the memory access patterns produced by a program's execution will leak information to the adversary.

In the following, we will discuss two main solutions to achieve secure cloud computing, i.e., secure-processor-based solution and secure-computation-based solution.

1.2 A hardware-software co-design for ensuring memory trace obliviousness

To protect the confidentiality of sensitive data in the cloud, thwarting software attacks alone is necessary but not sufficient. An attacker with physical access to the computing platform (e.g., an malicious insider or intruder) can launch various *physical attacks*, such as tapping memory buses, plugging in malicious peripherals, or using cold-(re)boots [40,81]. Such physical attacks can uncover secrets even when the software stack is provably secure.

A secure processor enables memory encryption [32, 56, 85–87] to hide the contents of memory from direct inspection, but an adversary can still observe memory addresses transmitted over the memory bus. As argued above, however, the memory address trace is a side channel that can leak sensitive information. We thus exploit the above idea to enable memory obfuscation within the secure processor, and employ programming language techniques to optimize program's execution.

Deploying the above idea to build a memory trace oblivious system in realistic hardware architectures is non-trivial. First, as explained above, the entire memory needs be split into regions of ORAM banks, whose access addresses are obfuscated, versus encrypted RAM, whose access addresses are not. Second, cache behaviors can break a program's memory trace oblivious execution, and it is hard to be tracked statically. The compiler needs hardware support to deterministically control the cache behavior. Third, the type system needs to deal with assembly code directly,

since otherwise the the MTO property may be broken during compilation.

To tackle these problems, we present a hardware-software co-design called GhostRider. GhostRider's hardware architecture supports both an encrypted RAM region, and multiple ORAM banks, which can be leveraged by programs to optimize the performance. GhostRider compiler can translate a program written in a C-like high-level language into assembly code using GhostRider's instruction set architecture (ISA). GhostRider provides a type checker over the assembly code to enforce a well-typed program enjoys MTO. We will explain GhostRider in detail in Chapter 3.

1.3 Automatic RAM-model secure computation

An alternative route to achieve secure cloud computing is through *secure computation*. Secure computation is a cryptographic technique that allows mutually distrusting parties to make collaborative use of their local data without harming privacy of their individual inputs. Since Yao's seminal paper [96], research on secure two-party computation—especially in the semi-honest model we consider here—has flourished, resulting in ever more efficient protocols [11, 38, 52, 97] as well as several practical implementations [16, 43, 45, 46, 54, 65]. Since the first system for general-purpose secure two-party computation was built in 2004 [65], efficiency has improved substantially [11, 46].

Almost all previous implementations of general-purpose secure computation assume the underlying computation is represented as a *circuit*. While theoretical developments using circuits are sensible (and common), compiling typical programs,

which assume a von Neumann-style Random Access Machine (RAM) model, to efficient circuits can be challenging. One significant challenge is handling *dynamic memory accesses* to an array in which the memory location being read/written depends on secret inputs. A typical program-to-circuit compiler typically makes an entire copy of the array upon every dynamic memory access, thus resulting in a huge circuit when the data size is large. Theoretically speaking, generic approaches for translating RAM programs into circuits incur, in general, $O(TN)$ blowup in efficiency, where T is an upper bound on the program's running time, and N is the memory size.

To address these limitations, researchers have more recently considered secure computation that works directly in the RAM model [38, 62]. The key insight is, to rely on ORAM to enable dynamic memory access with poly-logarithmic cost, while preventing information leakage through memory-access patterns. Gordon et al. [38] observed a significant advantage of RAM-model secure computation (RAM-SC) in the setting of *repeated sublinear-time queries* (e.g., binary search) on a large database. By amortizing the setup cost over many queries, RAM-SC can achieve *amortized* cost asymptotically close to the run-time of the underlying program in the insecure setting.

To enable secure computation with practical usage, it is ideal to have a complete system, so that developers can implement the applications in a high-level language (which are mostly in RAM-model) rather than implementing circuits directly, and the compiler can translate the program into an efficient secure computation protocol while enforcing security.

While pursuing this goal, the MTO approach that we developed for GhostRider can be leveraged in this setting as well. In particular, for a program, we can use the same kind of MTO analysis approach to decide whether or not we should store some data in an ORAM bank, and ensure that a program is MTO using a type system. Secure computation, however, imposes more security restrictions to be considered. For example, a secure computation requires the circuit to be evaluated by both parties. This means that both parties know the *instruction* being executed. Therefore, secure computation requires *instruction trace obliviousness* beyond memory trace obliviousness. Further, since ORAM protocols are implemented in circuits in secure computation, this allows programmers to make *non-blackbox* usage of ORAMs as well. For example, developers can use *tree-based non-recursive ORAM* [90], which is a less expensive building block of ORAM, directly to achieve better performance.

To address these issues, we developed the SCVM system which includes a SCVM intermediate representation, a compiler, and a secure type system to demonstrate how to achieve automatic efficient RAM-model computation. We detail SCVM in Chapter 4.

1.4 A programming framework for secure computation

As a last contribution, we also deliver the ObliVM system as an extension on top of SCVM. It provides a programming framework with more expressive power and easy-programmability to help developers write better algorithms more easily. ObliVM focuses more on how to facilitate developers to build secure computation

applications. The design goal is to allow both cryptographic experts to improve the efficiency of low-level cryptographic protocols, and application developers who may not be familiar with cryptography to implement efficient applications in a high-level language. To meet these goals, we designed a new high-level programming language, called ObliVM-Lang, as an extension to SCVM and implement a compiler. Using this language, we can implement programming abstractions, which enable application developers to implement algorithms in an easy and efficient way. ObliVM also provides a backend called ObliVM-SC to allow cryptographic experts to implement different protocols to further accelerate the execution of the whole system. We will explain ObliVM in Chapter 5.

1.5 Our Results and Contributions.

In this thesis, we propose the trace oblivious computation theory and bring it to practice. We design and build GhostRider, a hardware/software platform for provably secure, memory-trace oblivious program execution, which can compile programs to a realistic architecture while formally ensuring MTO. We also design and build SCVM and ObliVM to enable developers to develop efficient secure computation applications. In summary, our contributions are:

Trace obliviousness theory. We greatly extend the study of trace oblivious program execution by providing a theory to establish when if a program is trace oblivious. Particularly, we demonstrate the way how to formalize a language such that the adversary-observable execution traces generated by the program can be

modeled. We also demonstrate how type systems can be used to enforce the trace obliviousness of programs in these languages. On the one hand, using these type systems, we extend the set of oblivious programs that can be verified automatically, to include more efficient implementations. On the other hand, these type systems can be extended to analyze other side-channel leakages that can be expressed as traces, and thus used to defend against attacks leveraging these channels.

GhostRider system. GhostRider is the first system to bring trace oblivious computation theory to practice. By building GhostRider itself, we build the first memory-trace obliviousness compiler that emits target code for a realistic ORAM-capable processor architecture. GhostRider's compiler optimizes the generated assembly code while ensuring the optimized code still satisfies MTO. To enable these optimizations, GhostRider builds on the Phantom processor architecture [63] but exposes new features and knobs to the software. Our empirical results on a real processor demonstrate the feasibility of our architecture and show that compared to the baseline approach of placing everything in a single ORAM bank, our compile-time static analysis achieves up to nearly an order-of-magnitude speedup for many common programs.

SCVM system. We build SCVM as the first system to enable automatic RAM-model secure computation. SCVM provides a complete system that takes a program written in a high-level language and compiles it to a protocol for secure two-party computation of that program. To achieve this goal, SCVM provides an intermediate representation, a type system to ensure any well-typed program will generate a secure computation protocol secure in the semi-honest model, and a compiler

to transform a program written in a high-level language into a secure two-party computation protocol while integrating compile-time optimizations curcial for improving performance. Our evaluation shows a speedup of 1–2 orders of magnitude as compared to standard circuit-based approaches for securely computing the same programs.

ObliVM **system.** Building on top of SCVM, we design and implement ObliVM, which focuses more on richer expressive power and easy programmability while achieving state-of-the-art performance for secure computation. ObliVM provides an expressive programming language to allow both cryptographic experts and non-experts to use ObliVM to customize both front-end and back-end optimizations very easily. Using this programming language, we implement several programming abstractions in ObliVM to help developers design and implement new efficient oblivious algorithms. Experiments show that the automatically generated circuits incurs only 0.5% to 2% overhead over manually optimized implementations.

Chapter 2: Memory Trace Obliviousness: Basic Setting

In this chapter, we discuss a simple setting for memory trace oblivious program execution, which still captures the essential ideas. We consider a simple client-server scenario. A program is run on the client, but the data operated by the program is stored on the server. The goal is to enforce the server cannot learn any sensitive information from both the data stored on the server, and its interaction with the client-side program. We assume the server manages the data as a big memory, so that the only way a client program can interact with the server is through memory read/write APIs. Particularly, through a read call `read(i)` will request the data block indexed by `i` from the server, and a write call `write(i, data)` will update the data block indexed by `i` with `data`. The server thus can observe `i` and `data` during such interactions.

This chapter will explain how can we use ORAM to prevent the server to learn any information through these interactions, and how a compiler and a type system can help enforcing this security property over a program automatically while preserving efficiency.

Before we go into the details, we want to emphasize that although this setting itself has interesting applications in this setting, it can be extended to more appli-

cations such as secure processor applications, where CPU and RAM correspond to client and server respectively, and secure multi-party computation. We will discuss these extensions in Chapter 3, 4, 5. We want to explain the basic ideas and techniques to achieve *trace oblivious computation* in this chapter, which will be further extended in later chapters.

This chapter is based on a paper that I co-authored with Michael Hicks and Elaine Shi [58]. I developed the formalism and conducted the proof of the main memory trace obliviousness theorem under the help of Michael Hicks, and provided preliminary experimental results on performance.

2.1 Threat Model

Particularly, a server stores all the data, and a client runs programs interacting with the server to get data. We assume that the client has small local storage. The server manages the data as a random access memory (RAM), which supports read and write operation with random addresses. The adversarial model assumes that the server can observe all addresses that the client program is accessing, but not client programs' internal states. We assume the data stored on the server are all encrypted, so that the server cannot observe the data directly. Client program will decrypt the data once retrieved, and re-encrypt them before uploading to the server (via write operations). We consider the honest-but-curious model, such that the server does not modify the stored data. Orthogonal techniques, such as Merkle's hash tree, can be used to enforce data integrity.

Though they are a real threat, timing and other covert channels are not considered in this basic setting. Later, we will show how our GhostRider system (Chapter 3) and ObliVM (Chapter 5) can prevent leakage through these channels.

2.2 Motivating Examples

Let us consider the following program, where the client does not have enough space to store a big dataset, but offloads it to the server instead.

```
1: int findmax(public int n, secret int* data) {
2:     secret int max = data[0];
3:     for (public int i=1; i<n; ++i) {
4:         if (data[i] > max)
5:             max = data[i]
       }
   }
```

In this client program, `data` refers to the client's data stored on the server. The keyword `secret` is used to denote this data is sensitive. A straightforward idea to enforce that the server does not learn information through addresses is that the client program can run an ORAM protocol with the server and stores all data in a giant ORAM. This approach, however, has two drawbacks. First, this may not secure, because the total number of memory accesses may still leak the information. For example, in the above example, based on the total number of accesses, the server will learn how many times line 5 is executed. The server can infer about

some information such as whether `data` is in ascending order.

To mitigate this problem, the client program needs perform dummy accesses to the server even though the condition at line 4 is not satisfied. To this aim, this program needs be rewritten such that in the false-branch of line 4, a dummy statement `dummy_max = data[i]` is inserted, where `dummy_max` is an inserted dummy variable which has no side-effect to other data such as `max` and `data`.

Second, this is not efficient, as it will incur an ORAM overhead which is unnecessary. In particular, This program sequentially scans through the entire `data` array, and keeps the maximal value in `max`. In this case, ORAM is not necessary, since the data access addresses, i.e. `i`, are public information which can be inferred by the server without knowing any information about the secret data.

2.3 Approach Overview

In the following, we present several technical highlights in the following.

Memory Trace Obliviousness. The adversary can observe the stream of accesses to server, even if he cannot observe the content of those accesses, and such observations are sufficient to infer secret information. To eliminate this channel of information, we need a way to run the program so that the event stream does not depend on the secret data—no matter the values of the secret, the observable events will be the same. Programs that exhibit this behavior enjoy a property we call *memory trace obliviousness.*

Padding. Toward ensuring memory trace obliviousness, the compiler can add

padding instructions to either or both branches of `if` statements whose guards reference secret information (we refer to such guards as *high guards*). This idea is similar to inserting padding to ensure uniform timing [6, 10, 22, 42]. We need to insert dummy accesses to the two branches such that both branches have equivalent access patterns. We will detail our approach in Section 2.5.

ORAM for secret data. We store secret data in Oblivious RAM (ORAM), and extend our trusted computing base with an client side ORAM library. This library will encrypt/decrypt the secret data and maintain a mapping between addresses for variables used by the program and actual storage addresses for those variables on server. For each read/write issued for a secret address, the ORAM library will issue a series of reads/writes to the server with actual addresses, which has the effect of hiding which of the accesses was the real address. Moreover, with each access, the ORAM library will shuffle program/storage address mappings so that the physical location of any program variable is constantly in flux. Asymptotically, ORAM accesses are polylogarithmic blowup in the size of the ORAM [35]. Note that if we were concerned about integrity, we could compose the ORAM library with machinery for, say, authenticated accesses.

Multiple ORAM banks. ORAM can be an order of magnitude slower than regular DRAM [83]. Moreover, larger ORAM banks containing more variables incur higher overhead than smaller ORAM banks [35, 80]; as mentioned above, ORAM accesses are asymptotically related to the size of the ORAM. Thus we can reduce run-time overhead by allocating code/data in multiple, smaller ORAM banks rather than all of it in a single, large bank.

Arrays. Implicitly we have assumed that all of an array is allocated to the same ORAM bank, but this need not be the case. Indeed, for our example it is safe to simply encrypt the contents of `data[i]` because knowing which memory address we are accessing does not happen to reveal anything about the contents of `data[]`, as we have explained before.

If we allocate each array element in a separate ORAM bank, the running time of the program becomes roughly $2n$ accesses: each access to `data[i]` is in a bank of size 1. Each iteration will read `data` twice, and thus there are $2n$ accesses in total for n iterations.

In comparison, the naïve strategy of allocating all variables in a single ORAM bank would incur $2n \cdot \text{poly}\log(n+2))$ memory accesses (for secret variables), since each access to an ORAM bank of size m requires $O(\text{poly}\log(m))$ actual server accesses. This shows that we can achieve asymptotic gains in performance for some programs.

2.4 Memory trace obliviousness by typing

This section formalizes a type system for verifying that programs enjoy memory trace obliviousness. In the next section we describe a compiler to transform programs like the one in Section 2.2 so they can be verified by our type system.

We formalize our type system using a simple language $\mathcal{L}_{\text{basic}}$ presented in Figure 2.1. A program is a statement S which can be a sequence $S; S$, no-op **skip**, assignments to variables and arrays, conditionals, and loops. Expressions e consist

$$
\begin{array}{llll}
\text{Variables} & x, y, z & \in & \textbf{Vars} \\
\text{Numbers} & n & \in & \textbf{Nat} \\
\text{ORAM bank} & o & \in & \textbf{ORAMbanks} \\
\text{Expressions} & e & ::= & x \mid e \; op \; e \mid x[e] \mid n \\
\text{Statements} & S & ::= & \textbf{skip} \mid x := e \mid x[e] := e \mid \textbf{if}(e, S, S) \mid \textbf{while}\ (e, S) \mid S; S
\end{array}
$$

Figure 2.1: Language syntax of $\mathcal{L}_{\text{basic}}$

$$
\begin{array}{llll}
\text{Arrays} & m & \in & \textbf{Arrays} = \textbf{Nat} \rightharpoonup \textbf{Nat} \\
\text{Labels} & l & \in & \textbf{SecLabels} = \{L\} \cup \textbf{ORAMbanks} \\
\text{Memory} & M & \in & \textbf{Vars} \rightharpoonup (\textbf{Arrays} \cup \textbf{Nat}) \times \textbf{SecLabels} \\
\text{Traces} & t & ::= & \textbf{read}(x, n) \mid \textbf{readarr}(x, n, n) \mid \textbf{write}(x, n) \mid \textbf{writearr}(x, n, n) \\
 & & \mid & o \mid t@t \mid \epsilon
\end{array}
$$

$$
get(m, n) = \begin{cases} m(n) & \text{if } 0 \leq n < |m| \\ 0 & \text{otherwise} \end{cases}
$$

$$
upd(m, n_1, n_2) = \begin{cases} m[n_1 \mapsto n_2] & \text{if } 0 \leq n_1 < |m| \\ m & \text{otherwise} \end{cases}
$$

$$
evt(l, t) = \begin{cases} l & \text{if } l \in \textbf{ORAMbanks} \\ t & \text{otherwise} \end{cases}
$$

Figure 2.2: Auxiliary syntax and functions for semantics in $\mathcal{L}_{\text{basic}}$

of constant natural numbers, variable and array reads, and (compound) operations. For simplicity, arrays may contain only integers (and not other arrays), and bulk assignments between arrays (i.e., $x := y$ when y is an array) are not permitted.

2.4.1 Operational semantics

We define a big-step operational semantics for $\mathcal{L}_{\text{basic}}$ in Figure 2.3, which refers to auxiliary functions and syntax defined in Figure 2.2. Big-step semantics is simpler than the small-step alternative, and though it cannot be used to reason about non-terminating programs, our cloud computing scenario generally assumes that

$$\boxed{\langle M, e\rangle \Downarrow_t n}$$

$$\text{E-Var}\frac{M(x) = (n, l) \qquad t = evt(l, \mathbf{read}(x, n))}{\langle M, x\rangle \Downarrow_t n} \qquad \text{E-Const}\frac{}{\langle M, n\rangle \Downarrow_\epsilon n}$$

$$\text{E-Op}\frac{\langle M, e_1\rangle \Downarrow_{t_1} n_1 \qquad \langle M, e_2\rangle \Downarrow_{t_2} n_2 \qquad n = n_1 \; op \; n_2}{\langle M, e_1 \; op \; e_2\rangle \Downarrow_{t_1@t_2} n}$$

$$\text{E-Arr}\frac{\begin{array}{c}\langle M, e\rangle \Downarrow_t n \qquad M(x) = (m, l) \qquad n' = get(m, n) \\ t_1 = evt(l, \mathbf{readarr}(x, n, n'))\end{array}}{\langle M, x[e]\rangle \Downarrow_{t@t_1} n'}$$

$$\boxed{\langle M, s\rangle \Downarrow_t M'}$$

$$\text{S-Skip}\frac{}{\langle M, \mathbf{skip}\rangle \Downarrow_\epsilon M}$$

$$\text{S-Asn}\frac{\langle M, e\rangle \Downarrow_t n \qquad M(x) = (n', l) \qquad t' = evt(l, \mathbf{write}(x, n))}{\langle M, x := e\rangle \Downarrow_{t@t'} M[x \mapsto (n, l)]}$$

$$\text{S-AAsn}\frac{\begin{array}{c}\langle M, e_1\rangle \Downarrow_{t_1} n_1 \qquad \langle M, e_2\rangle \Downarrow_{t_2} n_2 \qquad M(x) = (m, l) \\ m' = upd(m, n_1, n_2) \qquad t = evt(l, \mathbf{writearr}(x, n_1, n_2))\end{array}}{\langle M, x[e_1] := e_2\rangle \Downarrow_{t_1@t_2@t} M[x \mapsto (m', l)]}$$

$$\text{S-Cond}\frac{\langle M, e\rangle \Downarrow_{t_1} n \qquad \langle M, S_i\rangle \Downarrow_{t_2} M \quad (i = ite(n, 1, 2))}{\langle M, \mathbf{if}\ (e, S_1, S_2)\ \rangle \Downarrow_{t_1@t_2} M'}$$

$$\text{S-WhileT}\frac{\begin{array}{c}\langle M, e\rangle \Downarrow_t n \qquad n \neq 0 \\ \langle M, (S; \mathbf{while}(e, S))\rangle \Downarrow_{t'} M'\end{array}}{\langle M, \mathbf{while}(e, S)\rangle \Downarrow_{t@t'} M'}$$

$$\text{S-WhileF}\frac{\langle M, e\rangle \Downarrow_t 0}{\langle M, \mathbf{while}(e, S)\rangle \Downarrow_t M}$$

$$\text{P-Stmts}\frac{\langle M, S_1\rangle \Downarrow_{t_1} M' \qquad \langle M', S_2\rangle \Downarrow_{t_2} M''}{\langle M, S_1; S_2\rangle \Downarrow_{t_1@t_2} M''}$$

Figure 2.3: Operational semantics of $\mathcal{L}_{\text{basic}}$

programs terminate. The main judgment of the former figure, $\langle M, S\rangle \Downarrow_t M'$ (shown at the bottom), indicates that program S when run under memory M will terminate with new memory M' and in the process produce a *memory access trace* t. We also define judgments $\langle M, e\rangle \Downarrow_t n$ for evaluating statements and expressions, respectively.

We model server side storage as memory M, which is a partial functions from variables to labeled values, where a value is either an array m or a number n, and a label is either L or an ORAM bank identifier o. Thus we can think of an ORAM bank (managed by the client side ORAM library) o containing all data for variables x such that $M(x) = (\_, o)$, whereas all data labeled L is stored on the server directly. We model an array m as a partial function from natural numbers to natural numbers. We write $|m|$ to model the length of the array; that is, if $|m| = n$ then $m(i)$ is defined for $0 \leq i < n$ but nothing else. To keep the formalism simple, we assume all of the data in an array is stored in the same place, i.e., all on the server directly or all in the same ORAM bank. We also assume data referred by non-array variables are also stored on the server. We will show how these assumptions can be relaxed while describing GhostRider (Chapter 3) and ObliVM (Chapter 3).

A memory access trace t is a finite sequence of events arising during program execution that are observable to the server. These events include read events $\mathbf{read}(x, n)$ which states that number n was read from variable x and $\mathbf{read}(x, n_1, n_2)$, which states number n_2 was read from $x[n_1]$. The corresponding events for writes to variables and arrays are similar, but refer to the number written, rather than read. Event o indicates an access to ORAM—only the storage bank o is discernable, not

the precise variable involved or even whether the access is a read or a write. (Each ORAM read/write in the program translates to several actual DRAM accesses, but we model them as a single abstract event.) Finally, $t@t$ represents the concatenation of two traces and ϵ is the empty trace.

The rules in Figure 2.3 are largely straightforward. Rule (E-Var) defines variable reads by looking up the variable in memory, and then emitting an event consonant with the label on the variable's memory. This is done using the *evt* function defined in Figure 2.2: if the label is some ORAM bank o then event o will be emitted, otherwise event $\mathbf{read}(x, n)$ is emitted since the access is to the server directly.

The semantics treats array accesses as "oblivious" to avoid information leakage due to out-of-bounds indexes. In particular, rule (E-Arr) indexes the array using auxiliary function *get*, also defined in Figure 2.2, that returns 0 if the index n is out of bounds. Rule (S-AAsn) uses the *upd* function similarly: if the write is out of bounds, then the array is not affected.[1] We could have defined the semantics to throw an exception, or result in a stuck execution, but this would add unnecessary complication. Supposing we had such exceptions, our semantics models wrapping array reads and writes with a try-catch block that ignores the exception, which is a common pattern, e.g., in Jif [19, 49], and has also been advocated by Deng and Smith [26].

The rule (S-Cond) for conditionals is the obvious one; we write *ite(x,y,z)* to denote y when x is 0, and z otherwise. Rule (S-WhileT) expands the loop one

[1]The syntax $m[n_1 \mapsto n_2]$ defines a partial function m' such that $m'(n_1) = n_2$ and otherwise $m'(n) = m(n)$ when $n \neq n_1$. We use the same syntax for updating memories M.

unrolling when the guard is true and evaluates that to the final memory, and rule (S-WhileF) does nothing when the guard is false. Finally rule (P-Stmts) handles sequences of statements.

2.4.2 Memory trace obliviousness

The security property of interest in our setting we call *memory trace obliviousness*. This property generalizes the standard (termination-sensitive) noninterference property to account for memory traces. Intuitively, a program satisfies memory trace obliviousness if it will always generate the same trace (and the same final memory) for the same adversary-visible memories, no matter the particular values stored in ORAM. We formalize the property in three steps. First we define what it means for two memories to be low-equivalent, which holds when they agree on memory contents having label L.

Definition 1 (Low equivalence)**.** *Two memories M_1 and M_2 are* low-equivalent*, denoted as $M_1 \sim_L M_2$, if and only if $\forall x, v. M_1(x) = (v, L) \Leftrightarrow M_2(x) = (v, L)$.*

Next, we define the notion of the Γ-*validity* of a memory M. Here, Γ is the *type environment* that maps variables to security types τ, which are either Nat l or Array l (both are defined in Figure 2.5). In essence, Γ indicates a mapping of variables to memory banks, and if memory M employs that mapping then it is valid with respect to Γ.

Definition 2 (Γ-validity)**.** *A memory M is valid under a environment Γ, or* Γ-valid*,*

$$\epsilon @ t \equiv t @ \epsilon \equiv t \qquad \frac{t_1 = t_2}{t_1 \equiv t_2} \qquad \frac{t_1 \equiv t_2}{t_2 \equiv t_1} \qquad \frac{t_1 \equiv t_2 \quad t_2 \equiv t_3}{t_1 \equiv t_3}$$

$$\frac{t_1 \equiv t_1' \quad t_2 \equiv t_2'}{t_1 @ t_2 \equiv t_1' @ t_2'} \qquad (t_1 @ t_2) @ t_3 \equiv t_1 @ (t_2 @ t_3)$$

Figure 2.4: Trace equivalence in $\mathcal{L}_{\text{basic}}$

if and only if, for all x

$$\Gamma(x) = Nat\ l \Leftrightarrow \exists n \in \boldsymbol{Nat}. M(x) = (n, l)$$

$$\Gamma(x) = Array\ l \Leftrightarrow \exists m \in \boldsymbol{Arrays}. M(x) = (m, l)$$

Finally, we define memory trace obliviousness. Intuitively, a program enjoys this property if all runs of the program on low-equivalent, Γ-valid memories will always produce the same trace and low-equivalent final memory.

Definition 3 (Memory trace obliviousness)**.** *Given a security environment* Γ*, a program* S *satisfies* Γ-memory trace obliviousness *if for any two* Γ*-valid memories* $M_1 \sim_L M_2$*, if* $\langle M_1, S\rangle \Downarrow_{t_1} M_1'$ *and* $\langle M_2, S\rangle \Downarrow_{t_2} M_2'$*, then* $t_1 \equiv t_2$*, and* $M_1' \sim_L M_2'$*.*

In this definition, we write $t_1 \equiv t_2$ to denote that t_1 and t_2 are equivalent. Equivalence is defined formally in Figure 2.4. Intuitively, two traces are equivalent if they are syntactically equivalent or we can apply associativity to transform one into the other. Furthermore, ϵ plays the role of the identity element.

2.4.3 Security typing

Figure 2.6 presents a type system that aims to ensure memory trace obliviousness. Auxiliary definitions used in the type rules are given in Figure 2.5. This

$$
\begin{array}{lcl}
\text{Types} & \tau & ::= \quad \text{Nat } l \mid \text{Array } l \\
\text{Environments} & \Gamma & \in \quad \textbf{Vars} \rightharpoonup \textbf{Types} \\
\text{Trace Pats} & T & ::= \quad \textbf{Read } x \mid \textbf{Write } x \mid \textbf{Readarr } x \mid \textbf{Writearr } x \\
 & & \quad \mid \quad \textbf{Loop}(T, T) \mid o \mid T@T \mid T + T \mid \epsilon
\end{array}
$$

$$
l_1 \sqcup l_2 \quad = \quad \begin{cases} l_1 & \text{if } l_1 \neq \text{L} \\ l_2 & \text{otherwise} \end{cases}
$$

$$
l_1 \sqsubseteq l_2 \quad \textit{iff} \quad \begin{array}{l} l_1 = L \text{ or} \\ l_2 \neq L \text{ and } l_2 \neq n \end{array}
$$

$$
\textit{select}(T_1, T_2) \quad = \quad \begin{cases} T_1 & \text{if } T_1 \sim_L T_2 \\ T_1 + T_2 & \text{otherwise} \end{cases}
$$

Figure 2.5: Auxiliary syntax and functions for typing in $\mathcal{L}_{\text{basic}}$

type system borrows ideas from standard security type systems that aim to enforce (traditional) noninterference. For the purposes of typing, we define a lattice ordering $\sqsubseteq$ among security labels l as shown in Figure 2.5, which also shows the $\sqcup$ (join) operation. Essentially, these definitions treat all ORAM bank labels o as equivalent for the purposes of typing (you can think of them as the "high" security label). In the definition of $\sqsubseteq$, we also consider the case when l_2 could be a program location n, which is treated as equivalent to L (this case comes up when typing labeled statements).

The typing judgment for expressions is written $\Gamma \vdash e : \tau; T$, which states that in environment Γ, expression e has type τ, and when evaluated will produce a trace described by the *trace pattern* T. The judgments for statements s and programs S are similar. Trace patterns describe families of run-time traces; we write $t \in T$ to say that trace t matches the trace pattern T.

Trace pattern elements are quite similar to their trace counterparts: ORAM accesses are the same, as are empty traces and trace concatenation. Trace pattern

$$\boxed{\Gamma \vdash e : \tau; T}$$

$$\text{T-Var}\frac{\Gamma(x) = \text{Nat } l \qquad T = evt(l, \mathbf{Read}(x))}{\Gamma \vdash x : \text{Nat } l; T} \qquad \text{T-Con}\frac{}{\Gamma \vdash n : \text{Nat L}; \epsilon}$$

$$\text{T-Op}\frac{\Gamma \vdash e_1 : \text{Nat } l_1; T_1 \qquad \Gamma \vdash e_2 : \text{Nat } l_2; T_2 \qquad l = l_1 \sqcup l_2}{\Gamma \vdash e_1 \ op \ e_2 : \text{Nat } l; T_1 @ T_2}$$

$$\text{T-Arr}\frac{\begin{array}{c}\Gamma(x) = \text{Array } l \qquad \Gamma \vdash e : \text{Nat } l'; T \qquad l' \sqsubseteq l \\ T' = evt(l, \mathbf{Readarr}(x))\end{array}}{\Gamma \vdash x[e] : \text{Nat } l \sqcup l'; T @ T'}$$

$$\boxed{\Gamma, l \vdash S; T}$$

$$\text{T-Skip}\frac{}{\Gamma, l_0 \vdash \mathbf{skip}; \epsilon}$$

$$\text{T-Asn}\frac{\Gamma \vdash e : \text{Nat } l; T \qquad \Gamma(x) = \text{Nat } l' \qquad l_0 \sqcup l \sqsubseteq l'}{\Gamma, l_0 \vdash x := e; T @ evt(l', \mathbf{Write}(x))}$$

$$\text{T-AAsn}\frac{\Gamma \vdash e_1 : \text{Nat } l_1; T_1 \qquad \Gamma \vdash e_2 : \text{Nat } l_2; T_2 \qquad \Gamma(x) = \text{Array } l \qquad l_1 \sqcup l_2 \sqcup l_0 \sqsubseteq l}{\Gamma, l_0 \vdash x[e_1] := e_2; T_1 @ T_2 @ evt(l, \mathbf{Writearr}(x))}$$

$$\text{T-Cond}\frac{\begin{array}{c}\Gamma \vdash e : \text{Nat } l; T \qquad \Gamma, l \sqcup l_0 \vdash S_i; T_i \ (i = 1, 2) \\ l \sqcup l_0 \neq \text{L} \Rightarrow T_1 \sim_L T_2 \qquad T' = select(T_1, T_2)\end{array}}{\Gamma, l_0 \vdash \mathbf{if}(e, S_1, S_2); T @ T'}$$

$$\text{T-While}\frac{\Gamma \vdash e : \text{Nat } l; T_1 \qquad \Gamma, l_0 \vdash S; T_2 \qquad l \sqcup l_0 \sqsubseteq \text{L}}{\Gamma, l_0 \vdash \mathbf{while}(e, S); \mathbf{Loop}(T_1, T_2)}$$

$$\text{T-Seq}\frac{\Gamma, l_0 \vdash S_1; T_1 \qquad \Gamma, l_0 \vdash S_2; T_2}{\Gamma, l_0 \vdash S_1; S_2; T_1 @ T_2}$$

Figure 2.6: Typing for $\mathcal{L}_{\text{basic}}$

events for reads and writes to variables and arrays are more abstract, mentioning the variable being read, and not the particular value (or index, in the case of arrays); we have $\mathbf{read}(x, n) \in \mathbf{Read}(x)$ for all n, for example. There is also the *or*-pattern $T_1 + T_2$ which matches traces t such that either $t \in T_1$ or $t \in T_2$. Finally, the trace pattern for loops, $\mathbf{Loop}(T_1, T_2)$, denotes the set of patterns T_1 and $T_1@T_2@T_1$ and $T_1@T_2@T_1@T_2@T_1$ and so on, and thus matches any trace that matches one of them.

Turning to the rules, we can see that each one is structurally similar to the corresponding semantics rule. Each rule likewise uses the *evt* function (Figure 2.2) to selectively generate an ORAM event o or a basic event, depending on the label of the variable being read/written. Rule (T-Var) thus generates a $\mathbf{Read}(x)$ pattern if x's label is L, or generates the ORAM event l (where $l \neq L$ implies l is some bank o). As expected, constants n are labeled L by (T-Con), and compound expressions are labeled with the join of the labels of the respective sub-expressions by (T-Op). Rule (T-Arr) is interesting in that we require $l \sqsubseteq l'$, where l is the label of the index and l' is the label of the array, but the label of the resulting expression is the join of the two. As such, we can have a public index of a secret array, but not the other way around. This is permitted because of our oblivious semantics: a public index reveals nothing about the length of the array when the returned result is secret, and no out-of-bounds exception is possible.

The judgment for statements $\Gamma, l_0 \vdash S; T$ is similar to the judgment for expressions, but there is no final type, and it employs the standard *program counter (PC) label* l_0 to prevent implicit flows. In particular, the (T-Asn) and (T-AAsn) rules both require that the join of the label l of the expression on the rhs, when joined

with the program counter label l_0, must be lower than or equal to the label l' of the variable; with arrays, we must also join with the label l_1 of the indexing expression. Rule (T-Cond) checks the statements S_i under the program counter label that is at least as high as the label of the guard. As such, coupled with the constraints on assignments, any branch on a high-security expression will not leak information about that expression via an assignment to a low-security variable. In a similar way, rule (T-Lab) requires that the statement location p is lower or equal to the program counter label, so that a public instruction fetch cannot be the source of an implicit flow.

Rule (T-Cond) also ensures that if the PC label or that of the guard expression is secret, then the actual run-time trace of the true branch (matched by the trace pattern T_1) and the false branch (pattern T_2) must be equal; if they were not, then the difference would reveal something about the respective guard. We ensure run-time traces will be equal by requiring the trace patterns T_1 and T_2 are *equivalent*, as axiomatized in Figure 3.6. The first two rows prove that ϵ is the identity, that $\sim_L$ is a transitive relation, and that concatenation is associative. The third row unsurprisingly proves that ORAM events to the same bank and fetches of the same location/bank are equivalent. More interestingly, the third row claims that public reads to the same variable are equivalent. This makes sense given that public writes are *not* equivalent. As such, reads in both branches will always return the same run-time value they had prior to the conditional. Notice that the public reads to the same arrays are also *not* equivalent, since indices may leak information. Finally, the (T-Cond) emits trace T', which according to the *select* function (Figure 2.5) will

$$\frac{T \sim_L T}{\epsilon @ T \sim_L T @ \epsilon \sim_L T} \qquad \epsilon \sim_L \epsilon \qquad \frac{T_1 \sim_L T_2 \quad T_2 \sim_L T_3}{T_1 \sim_L T_3} \qquad o \sim_L o$$

$$\frac{T_1 \sim_L T_1' \quad T_2 \sim_L T_2'}{T_1 @ T_2 \sim_L T_1' @ T_2'} \qquad (T_1 @ T_2) @ T_3 \sim_L T_1 @ (T_2 @ T_3) \qquad \textbf{Read}\ x \sim_L \textbf{Read}\ x$$

Figure 2.7: Trace pattern equivalence in $\mathcal{L}_{\text{basic}}$

be T_1 when the two are equivalent. As such, conditionals in a high context will never produce or-pattern traces (which are not equivalent to any other trace pattern).

In Rule (T-While), the constraint $l \sqcup l_0 \sqsubseteq L$ mandates that loop guards be public (which is why we need not join l_0 with l when checking the body S). This constraint ensures that the length of the trace as related to the number of loop iterations cannot reveal something about secret data. Fortunately, this restriction is not problematic for many examples because secret arrays can be safely indexed by public values, and thus looping over arrays reveals no information about them.

Finally, we can prove that well typed programs enjoy memory trace obliviousness.

Theorem 1. *If $\Gamma, l \vdash S; T$, then S satisfies memory trace obliviousness.*

The full proof can be found in Appendix A.

2.4.4 Examples

Now we consider a few programs that do and do not type check in our system. In the examples, public (low security) variables begin with `p`, and secret (high security) variables begin with `s`; we assume each secret variable is allocated in its own ORAM bank (and ignore statement labels).

There are some interesting differences in our type system and standard information flow type systems. One is that we prohibit low reads under high guards that could differ in both branches. For example, the program `if s > 0 then s :=` $\mathtt{p}_1$ `else s :=` $\mathtt{p}_2$ is accepted in the standard type system but rejected in ours. This is because in our system we allow the adversary to observe public reads, and thus he can distinguish the two branches, whereas an adversary can only observe public writes in the standard noninterference proof. On the other hand, the program `if s > 0 then s := p+1 else s := p+2` would be accepted, because both branches will exhibit the same trace.

Another difference is that we do not allow high guards in loops, so a program like the following is acceptable in the standard type system is rejected in ours:

```
s := slen; sum := 0;
while s ≥ 0 do
  sum := sum + sarr[p];
  s := s - 1;
done
```

The reason we reject this program is that the number of loop iterations, which in general cannot be determined at compile time, could reveal information about the secret at run-time. In this example, the adversary will observe $O(\mathtt{s})$ memory events and thus can infer `slen` itself. Prior work on mitigating timing channels often makes the same restriction for the same reason [6, 10, 22, 42]. Similarly, we can mitigate the restrictiveness of our type system by padding out the number of iterations to a

constant value. For example, we could transform the above program to be instead

```
p := N; sum := 0;
while p ≥ 0 do
  if p < slen then sum := sum + sarr[p];
   else sdummy := sdummy + sarr[p];
  p := p - 1;
done
```

Here, `N` is some constant and `sdummy` and `sum` are allocated in the same ORAM bank. The loop will always iterate `N` times but will compute the same `sum` assuming `N` $\geq$ `slen`.

We also do not allow loops with low guards to appear in a conditional with a high guard. As above, we may be able to transform a program to make it acceptable. For example, for some S, the program `if s > 0 then while (p > 0) do` S`; done` could be transformed to be `while (p > 0) do if s > 0 then` S`; done` (assuming `s` is not written in S). This ensures once again that we do not leak information about the loop guard.

2.5 Compilation

Rather than requiring programmers to write memory-trace oblivious programs directly, we would prefer that programmers could write arbitrary programs and rely on a compiler to transform those programs to be memory trace oblivious. While more fully realizing this goal remains future work, we have developed a compiler

algorithm that automates some of the necessary tasks.

In particular, given a program P in which the inputs and outputs are labeled as `secret` or `public`, our compiler will (a) infer the least labels (`secret` or `public`) for the remaining, unannotated variables; (b) allocate all `secret` variables to distinct ORAM banks; (c) insert padding instructions in conditionals to ensure their traces are equivalent; and finally, (d) allocate instructions appearing in high conditionals to ORAM banks. These steps are sufficient to transform the `max` program in section 2.2 into its memory-trace oblivious counterpart. We can also transform other interesting algorithms, such as k-means, Dijkstra's shortest paths, and matrix multiplication, as we discuss in the next section.

We now sketch the different steps of our algorithm.

2.5.1 Type checking source programs

The first step is to perform *label inference* on the source program to make sure that we can compile it. This is the standard, constraint-based approach to local type inference as implemented in languages like Jif [49] and FlowCaml [74]. We introduce fresh constraint variables for the labels of unannotated program variables, and then generate constraints based on the structure of the program text. This is done by applying a variant of the type rules in Figure 2.6, having three differences. First, we treat labels l has being either L, representing `public` variables; H, representing `secret` variables (we can think of this as the only available ORAM bank); or α, representing constraint variables. Second, premises like $l_1 \sqsubseteq l_2$ and $l_0 \sqcup l_1 \sqsubseteq l_2$

that appear in the rules are interpreted as *generating* constraints that are to be solved later. Third, all parts having to do with trace patterns T are ignored. Most importantly, we ignore the requirement that $T_1 \sim_L T_2$ for conditionals.

Given a set of constraints generated by an application of these rules, we attempt to find the least solution to the variables α that appear in these constraints, using standard techniques [30]. If we can find a solution, the compilation may continue. If we cannot find a solution, then we have no easy way to make the program memory-trace oblivious, and so the program is rejected.

As an example, consider the `findmax` program in Section 2.2, but assume that variables `i` and `max` are not annotated, i.e., they are missing the `secret` and `public` qualifiers. When type inference begins, we assign `i` the constraint variable α_i and `max` the constraint variable α_m. In applying the variant type rules (with the PC label l_0 set to L) to this program (that is, the part from lines 5–7), we will generate the following constraints:

$(\alpha_i \sqcup L) \sqcup L \sqsubseteq L$	*line* 3
$\alpha_i \sqsubseteq H$	*line* 4, for $h[i]$ in guard
$l_0 = \alpha_i \sqcup H \sqcup \alpha_m \sqcup L$	PC label for checking if branch
$\alpha_i \sqsubseteq H$	*line* 5, for $h[i]$ in assignment
$l_0 \sqcup (H \sqcup \alpha_i) \sqsubseteq \alpha_m$	*line* 5, assignment
$L \sqcup (\alpha_i \sqcup L) \sqsubseteq \alpha_i$	*line* 3

(For simplicity we have elided the constraints on location labels that arise due to

(T-Lab), but normally these would be included as well.) We can see that the only possible solution to these constraints is for α_i to be L and α_m to be H, i.e., the former is `public` and the latter is `secret`.

Assuming that the programmer minimally labels the source program, only indicating those data that *must* be secret and leaving all other variables unlabeled, then the main restriction on source programs is the restriction on the use of loops: all loop guards must be public, and no loop may appear in a conditional whose guard is high. As mentioned in the previous section, the programmer may transform such programs into equivalent ones, e.g., by using a constant loop bound, or by hoisting loops out of conditionals. We leave the automation of such transformations to future work.

2.5.2 Allocating variables to ORAM banks

Given all variables that were identified as `secret` in the previous stage, we need to allocate them to one or more ORAM banks. At one extreme, we could put all secret variables in a single ORAM bank. The drawback is that each access to a secret variable could cause significant overhead, since ORAM accesses are polylogarithmic in the size of the ORAM [35] (on top of the encryption/decryption cost). At the other extreme, we could put every secret variable in a separate ORAM bank. This lowers overhead by making each access cheaper but will force the next stage to insert more padding instructions, adding more accesses overall. Finally, we could attempt to choose some middle ground between these extreme methods: put some variables

in one ORAM bank, and some variables in others.

Ultimately, there is no analytic method for resolving this tradeoff, as the "break even" point for choosing padding over increased bank size, or vice versa, depends on the implementation. A profile-guided approach to optimizing might be the best approach. With our limited experience so far we observe that storing each secret variable in a separate ORAM bank generally achieves very good performance. This is because when conditional branches have few instructions, the additional padding adds only a small amount of overhead compared to the asymptotic slowdown of increased bank size. Therefore we adopt this method in our experiments. Nevertheless, more work is needed to find the best tradeoff in a practical setting.

We also need to assign secret statements (i.e., those statements whose location label must be H) to ORAM banks. At this stage, we assign all statements under a given conditional to the same ORAM bank, but we make a more fine-grained allocation after the next stage, discussed below.

2.5.3 Inserting padding instructions

The next step is to insert padding instructions into conditionals, to ensure the final premise of (T-Cond) is satisfied, so that both branches will generate the same traces.

To do this, we can apply algorithms that solve the *shortest common supersequence* problem [31] when applied to two traces (a.k.a. the 2-scs problem). That is, given the two trace patterns T_t and T_f for the true and false branches of an `if`

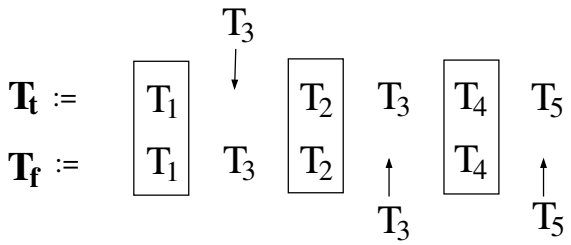

Figure 2.8: **Finding a short padding sequence using the greatest common subsequence algorithm.** An example with two abstract traces $\mathbf{T}_t = [T_1; T_2; T_3; T_4; T_5]$ and $\mathbf{T}_f = [T_1; T_3; T_2; T_4]$. One greatest common subsequence as shown is $[T_1; T_2; T_4]$. A shortest common super-sequence of the two traces is $\mathbf{T}_{tf} = [T_1; T_3; T_2; T_3; T_4; T_5]$.

(following ORAM bank assignment), let T_{tf} denote the 2-scs of T_t and T_f. The differences between T_{tf} and the original traces signal where, and what, padding instructions must be inserted. The standard algorithm builds on the dynamic programming solution to the *greatest common subsequence* (gcs) algorithm, which runs in time $O(nm)$ where n and m are the respective lengths of the two traces [23]. Using this algorithm to find the gcs reveals which characters must be inserted into the original strings, as illustrated in Figure 2.8.

When running 2-scs on traces, we view T_t and T_f as strings of characters which are themselves trace patterns due to single statements. Each statement-level pattern will always consist of zero or more of the following events: **Read**, o_i for ORAM bank i .[2] For example, suppose we have the program **skip**; $x[y] := z$ where, after ORAM bank assignment, the type of y is Nat o_1, the type of z is Array o_1, and the type of x is Nat o_2. This program generates trace pattern $\epsilon @ o_1 @ o_1 @ o_2$. For the purposes of running 2-scs, this trace consists of three characters: o_1, o_1, and o_2, which corresponds to the statement $x[y] := z$.

[2]Because of the restrictions imposed by the type system, T_t and T_f will never contain loop patterns, (public) read-array or write patterns, or or-patterns.

Once we have computed the 2-scs and identified the padding characters needed for each trace, we must generate "dummy" statements to insert in the program that generate the same events. This is straightforward. In essence, we can allocate a "dummy" variable d_o for each ORAM bank o in the program, and then read, write, and compute on that variable as needed to generate the correct event. Suppose we had the program $\mathbf{if}(e, \mathbf{skip}, x[y] := z)$ and thus $T_t = \epsilon$ and $T_f = o_1@o_1@o_2$. Computing the 2-scs we find that T_t can be pre-padded with $o_1@o_1@o_2$ while T_f needs not be padded. We can readily generate statements that correspond to both. For the second, we do not need to pad anything. For the first, we can produce $d_{o_2} := d_{o_1} + d_{o_1}$. When we must produce an event corresponding to a public read, or read from an array, we can essentially just insert a read from that variable directly.

Note that this approach will generate more padding instructions than is strictly needed. In the above example, the final program will be

$$\mathbf{if}(e, (d_{o_2} := d_{o_1} + d_{o_1}; \mathbf{skip}), (x[y] := z))$$

Peephole optimizations can be used to eliminate some superfluous instructions. However, a better approach is to use a finer-grained alphabet which in practice is available when using three address code, i.e., as the intermediate representation of an actual compiler. We will demonstrate this in GhostRider.

Once padding has been inserted, both branches have the same number of statements, and thus we can allocate each pair of statements in its own ORAM bank. Assuming we did not drop the **skip** statements in the program above, we

Table 2.1: **Programs and parameters used in our simulation.**

No.	Description
1	Dijkstra ($n = 100$ nodes)
2	K-means ($n = 100$ data points, $k = 2$, $I = 1$ iteraion)
3	Matrix multiplication ($n \times n$ matrix where $n = 40$)
4	Matrix multiplication ($n \times n$ matrix where $n = 25$)
5	Find max ($n = 100$ elements in the array)
6	Find max ($n = 10000$ elements in the array)

could allocate them both in ORAM bank o_3 and allocate the two assignments in ORAM bank o_4, rather than allocate all instructions in ORAM bank o as is the case now.

2.6 Evaluation

To demonstrate the efficiency gains achieved by our compiler in comparison with the straightforward approach of placing all secret variables in the same ORAM bank, we choose four example programs: Dijkstra single-source shortest paths, K-means, Matrix multiplication (naïve $O(n^3)$ implementation), and Find-max.

We will compare three different strategies:

Strawman: Place all secret variables in the same ORAM bank .

Opt 1: Store each variable in a separate ORAM bank, but store whole arrays in the same bank.

Opt 2: Store each variable and each member of an array in a separate ORAM bank .

In all three cases, we insert necessary padding to ensure obliviousness.

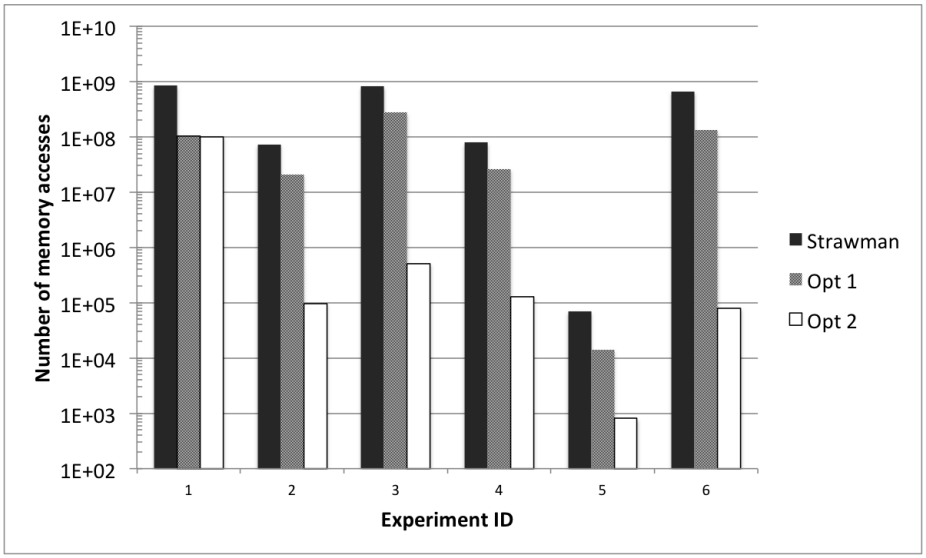

Figure 2.9: Simulation Results for Strawman, Opt 1, and Opt 2.

2.6.1 Simulation Results

We also performed simulation to measure the performance of the example programs when compiled by our compiler. Table 2.1 shows the parameters we choose for our experiment. We built a simulator in C++ that can measure the number of memory accesses for transformed programs. Implementing a full-fledged compiler that integrates with our ORAM-capable hardware concurrently being built [**?**] is left as future work.

Simulation results are given in Figure 2.9 for the six setups described in Table 2.1. The ORAM scheme we used in the simulation is due to Shi et al [80]. The figure shows that Opt 1 is 1.3 to 5 times faster the strawman scheme; and Opt 2 is *1 to 3 orders of magnitude* faster than the strawman for the chosen programs and parameters.

2.7 Conclusion Remarks

In this chapter, we have briefly explained how to achieve memory trace obliviousness (MTO) in a basic client-server setting. We have demonstrated how MTO is defined under $\mathcal{L}_{\text{basic}}$ syntax and semantics, and how a type system can enforce MTO by keeping track of traces. Throughout the rest of this thesis, we will exploit the same idea in more settings, where hardware (in GhostRider) or algorithmic (in ObliVM) constraints will require reasoning about more leakage. We will also empirically demonstrate the advantages of the MTO approach in these settings.

Chapter 3: GhostRider: A Compiler-Hardware Approach

3.1 Introduction

In Chapter 2, we have explained the basic idea how a type system can enforce a program to be memory trace oblivious. In this chapter, we consider a realistic setting to protect the programs against physical attackers who have control to everything except the processor. As explained in above, data encryption is necessary but not sufficient. Our approach is to build a processor with an ORAM controller [28, 63] so that it can obfuscate the memory accesses to the ORAM. As explained in Chapter 2, this is not the most efficient approach, since when a program's memory access pattern does not depend on secret data, encryption is sufficient and ORAM is not necessary. To enable such an optimization, we enhance the existing ORAM processor design to allow splitting memory into regions, consisting of one or more ORAM banks, encrypted memory, and normal DRAM.

On top of such a design, building a compiler from a high-level source language into a low-level assembly language is non-trivial, even given our results from Chapter 2. In particular, the cache behavior and handling assembly code make designing a MTO type system much more challenging.

In this Chapter, we present GhostRider as a hardware-compiler co-design ap-

proach to achieve memory trace obliviousness and efficiency at the same time.

This Chapter is based on a paper that I co-authored with Austin Harris, Michael Hicks, Martin Maas, Mohit Tiwari, and Elaine Shi [57]. I developed the formalism and the proof under the help of Michael Hicks. I also developed the compiler, which implements both the optimization and the type checker, and emits code that is runnable on an FPGA implementation developed by my co-authors Austin Harris, Martin Maas, and Mohit Tiwari. I conducted experiments to show the compiler's effectiveness with the help of Austin Harris, Martin Maas, and Mohit Tiwari.

3.1.1 Our Results and Contributions.

In this paper, we make the first endeavor to bring the theory of MTO to practice. We design and build GhostRider, a hardware/software platform for provably secure, memory-trace oblivious program execution. Compiling to a realistic architecture while formally ensuring MTO raises interesting challenges in the compiler and type system design, and ultimately requires a co-operative re-design of the underlying processor architecture. Our contributions are:

New compiler and type system. We build the first memory-trace oblivious compiler that emits target code for a realistic ORAM-capable processor architecture. The compiler must explicitly handle low-level resource allocation based on underlying hardware constraints, and while doing so is standard in non-oblivious compilers, achieving them while respecting the MTO property is non-trivial. Stan-

dard resource allocation mechanisms would fail to address the MTO property. For example, register allocation spills registers to the stack, thereby introducing memory events. Furthermore, caching serves memory requests from an on-chip cache, which suppresses memory events. If these actions are correlated with secret data, they can leak information. We introduce new techniques for resolving such challenges. In lieu of implicit caches we employ an explicit, on-chip *scratchpad*. Our compiler implements caching in software when its use does not compromise MTO.

To formally ensure the MTO property, we define a new type system for a RISC-style low-level assembly language. We show that any well-typed program in this assembly language will respect memory-trace obliviousness during execution. When starting from source programs that satisfy a standard information flow type system [26], our compiler generates type-correct, and therefore safe, target code. Specifically, we implement a type checker that can verify the type-correctness of the target code.

Processor architecture for MTO program execution. To enable an automated approach for efficient memory-trace oblivious program execution, we need new hardware features that are not readily available in existing ORAM-capable processor architectures [27, 29, 63]. GhostRider builds on the Phantom processor architecture [63] but exposes new features and knobs to the software. In addition to supporting a scratchpad, as mentioned above, the GhostRide architecture complements Phantom's ORAM support with *encrypted RAM* (ERAM), which is not oblivious and therefore more efficiently supports variables whose access patterns are not sensitive. Section 3.6 describes additional hardware-level contributions. We proto-

typed the GhostRider processor on a Convey HC2 platform [21] with programmable FPGA support. The GhostRider processor supports the RISC-V instruction set [93].

Implementation and Empirical Results. Our empirical results are obtained through a combination of software emulation and experiments on an FPGA prototype. Our FPGA prototype supports one ERAM bank, one code ORAM bank, and one data ORAM bank. The real processor experiments demonstrate the feasibility of our architecture, while the software simulator allows us to test a range of configurations not limited by the constraints of the current hardware. In particular, the software simulator models multiple ORAM banks at a higher clock rate.

Our experimental results show that compared to the baseline approach of placing everything in a single ORAM bank, our compile-time static analysis achieves up to nearly an order-of-magnitude speedup for many common programs.

3.2 Architecture and Approach

This section motivates our approach and presents an overview of GhostRider's hardware/software co-design.

3.2.1 Motivating example

We wish to support a scenario in which a client asks an untrusted cloud provider to run a computation on the client's private data. For example, suppose the client wants the provider to run the program shown in Figure 3.1, which is a simple histogram program written in a C-like source language. As input, the program takes

```
void histogram(secret int a[],   // ERAM
               secret int c[]) { // ORAM (output)
  public int i;
  secret int t, v;
  for(i=0;i<100000;i++) // 100000 <= len(c)
    c[i]=0;
  i=0;
  for(i=0;i<100000;i++) { // 100000 <= len(a)
    v=a[i];
    if(v>0) t=v%1000;
       else t=(0-v)%1000;
    c[t]=c[t]+1; } }
```

Figure 3.1: Motivating source program of GhostRider.

an integer array `a`, and as output it modifies integer array parameter `c`. We assume both arrays have size 100,000. The function's code is straightforward, computing the histogram of the absolute values of integers modulo 1000 appearing in the input array. The client's security goal is *data confidentiality*: the cloud provider runs the program on input array `a`, producing output array `c`, but nevertheless learns nothing about the contents of either `a` or `c`. We express this goal by labeling both arrays with the qualifier `secret` (data labeled `public` is non-sensitive).

3.2.2 Threat model

The adversary has physical access to the machine(s) being used to run client computations. As in prior work that advocates the minimization of the hardware trusted computing base (TCB) [85–87], we assume that trust ends at the boundary of the secure processor. Off-chip components are considered *insecure*, including memory, system buses, and peripherals. For example, we assume the adversary can observe the contents of memory, and can observe communications on the bus

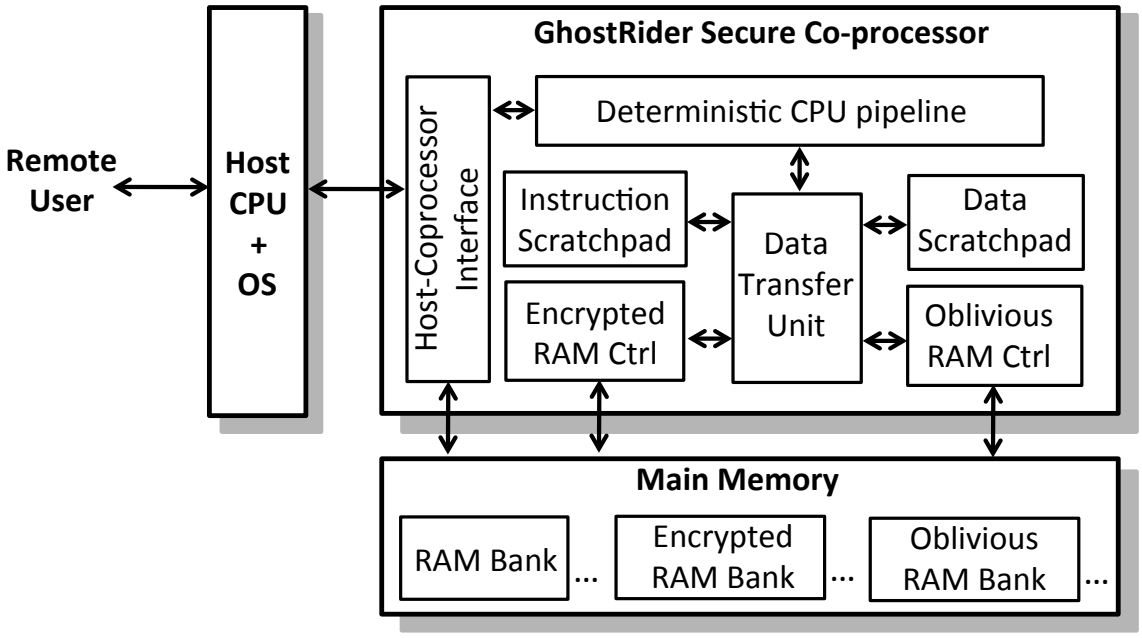

Figure 3.2: GhostRider architecture.

between the processor and the memory. By contrast, we assume that on-chip components are secure. Specifically, the adversary cannot observe the contents of the cache, the register file, or any on-chip communications. Finally, we assume the adversary can make fine-grained timing measurements, and therefore can learn, for example, the gap between observed events. Analogous side channels such as power consumption are outside the scope of this paper.

3.2.3 Architectural Overview

As mentioned in the introduction, one way to defend against such an adversary is to place all data in a single (large) ORAM; e.g., for the program in Figure 3.1 we place the arrays `a` and `c` in ORAM. Unfortunately this *baseline* approach is not only expensive, but also leaks information through the total number of ORAM accesses (if the access trace is not padded to a value that is independent of secret data). We

now provide an architectural overview of GhostRider (Figure 3.2) and contrast it with this baseline.

Joint ORAM-ERAM memory system In the GhostRider architecture, main memory is split into three types—normal (unencrypted) memory (RAM), encrypted memory (ERAM), and oblivious RAM (ORAM)—with one or more (logical) banks of each type comprising the system's physical memory. The differentiation of memory into banks allows a compiler to place only arrays with sensitive access patterns inside the more expensive ORAM banks, while keeping the remaining data in the significantly faster RAM or ERAM banks. For example, notice that in the program in Figure 3.1 the array `a` is always accessed sequentially while access patterns to the array `c` can depend on secret array contents. Therefore, our GhostRider compiler can place the array `a` inside an ERAM bank, and place the array `c` inside an ORAM bank. The program accesses different memory banks at the level of blocks using instructions that specify the bank and a block-offset within the bank (after moving data to on-chip memory as described below). Our hardware prototype fixes block sizes to be 4KB for both ERAM and ORAM banks (which is not an inherent limitation of the hardware design).

Software-directed scratchpad As mentioned earlier, cache hit and miss behavior can lead to differences in the observable memory traces. To prevent such cache-channel information leakage, the GhostRider architecture turns off implicit caching, and instead offers software-directed scratchpads for both instructions and data. These scratchpads are mapped into the program's address space so that the compiler can generate code to access them explicitly, and thereby avoid information leaks.

For example, the indices of array `a` in Figure 3.1 are deterministic; they do not depend on any secret input. As such, it is safe to use the scratchpad to cache array `a`'s accesses. The compiler generates code to check whether the relevant block is in the scratchpad, and if not loads the block from memory. On the other hand, all accesses to array `c` depend on the secret input `a`, so a memory request will always be issued independent of whether the requested block is in the scratchpad or not.

Deterministic Processor Pipeline To avoid timing-channel leakage, our pipelined processor ensures that instruction timings are deterministic. We do not use dynamic branch prediction and fix variable-duration instructions, such as division, to take the worst-case execution time, and disable concurrent execution of other instructions.

Initialization We design the oblivious processor and memory banks as a *co-processor* that runs the application natively (i.e., without an OS) and is connected to a networked host computer that can be accessed remotely by a user. We assume that the secure co-processor has non-volatile memory for storing a long-term public key (certified using PKI), such that the client can securely ship its encrypted code and data to the remote host, and initialize execution on the secure co-processor. Implementing the secure attestation is standard [4], and we leave it to future work.

3.3 Formalizing the target language

This section presents a small formalization of GhostRider's instruction set, which we call $\mathcal{L}_{\text{GhostRider}}$. The next section presents a type system for this language that guarantees security, and the following section describes our compiler from a

$$
\begin{array}{llll}
m, n & \in & \mathbb{Z} & o_1, ..., o_n \in \textbf{ORAMbanks} \\
k & \in & \textbf{Block IDs} & r \in \textbf{Registers} \\
l & \in & \textbf{Labels} = \{D, E\} \cup \textbf{ORAMbanks} & \\
\iota & ::= & \textbf{ldb}\ k \leftarrow l[r] & \text{load block to scratchpad} \\
 & | & \textbf{stb}\ k & \text{store block to memory} \\
 & | & r \leftarrow \textbf{idb}\ k & \text{retrieve the block ID} \\
 & | & \textbf{ldw}\ r_1 \leftarrow k[r_2] & \text{load a scratchpad val. to reg.} \\
 & | & \textbf{stw}\ r_1 \rightarrow k[r_2] & \text{store a reg. val. to scratchpad} \\
 & | & r_1 \leftarrow r_2\ aop\ r_3 & \text{compute an operation} \\
 & | & r_1 \leftarrow n & \text{assign a constant to a register} \\
 & | & \textbf{jmp}\ n & \text{(relative) jump} \\
 & | & \textbf{br}\ r_1\ rop\ r_2 \hookrightarrow n & \text{compare and branch} \\
 & | & \textbf{nop} & \text{empty operation} \\
I & ::= & \iota \mid I; \iota & \text{instruction sequence}
\end{array}
$$

Figure 3.3: Syntax for $\mathcal{L}_{\text{GhostRider}}$ language, comprising (1) **ldb** and **stb** instructions that move data blocks between scratchpad and a specific ERAM or ORAM bank, and (2) scratchpad-to-register moves and standard RISC instructions.

C-like source language to well-typed $\mathcal{L}_{\text{GhostRider}}$ programs.

3.3.1 Instruction set

The core instructions of $\mathcal{L}_T$ are in the style of RISC-V [93], our prototype's instruction set, and are formalized in Figure 3.3. We define *labels* l that distinguish the three kinds of main memory: D for normal (D)RAM, E for ERAM, and o_i for ORAM. For the last, the i identifies a particular ORAM bank. We can view each label as defining a distinct address space.

The instruction **ldb** $k \leftarrow l[r]$ loads a block from memory into the scratchpad.[1] Here, l is the address space, r is a register containing the address of the block to load from within that address space, and k is the scratchpad block identifier. Our

[1] In our hardware prototype the scratchpad is mapped into addressable memory, so this instruction and its counterpart, **stb**, are implemented as data transfers. In addition, the compiler implements **idb**. We model them in $\mathcal{L}_T$ explicitly for simplicity; see Section 3.6 for implementation details.

formalism refers to scratchpad blocks by their identifier, treating them similarly to registers. Our architecture remembers the address space and block address within that address space that the scratchpad block was loaded from so that writebacks, via the **stb** k instruction, will go to the original location. We enforce this one-to-one mapping to avoid information leaks via write-back from the scratchpad (e.g., that where a scratchpad block is written to, in memory, could reveal information about a secret, or that the effect of a write could do so, if blocks are aliased).

To access values from the scratchpad, we have scratchpad-load and scratchpad-store instructions. The scratchpad-load instruction loads a word from a scratchpad block, having the form **ldw** $r_1 \leftarrow k[r_2]$. Assuming register r_2 contains n, this instruction loads the n-th word in block k into register r_1 (notice that we use word-oriented addressing, not byte-oriented). The scratchpad-store instruction is similar, but goes in the reverse direction. The instruction $r \leftarrow$ **idb** k retrieves the block offset of a scratchpad block k.

We have two kinds of assignment instructions, one in the form of $r_1 \leftarrow r_2$ *aop* r_3, and the other in the form of $r \leftarrow n$. In $\mathcal{L}_T$ we only model integer arithmetic operations, such as addition, subtraction, multiplication, division, and modulus.

Jumps and branches use relative addressing. The jump instruction **jmp** n bumps the program counter by n instructions (where n can be negative). Branches, having the form **br** r_1 *rop* $r_2 \hookrightarrow n$, will compare the contents of r_1 and r_2 using *rop*, and will bump the pc by n if the comparison result is true. An instruction sequence I is defined to be a sequence of instructions concatenated using a logical operation ;.

```
; v=a[i]
 1  t1 ← ri div sizeblk
 2  t2 ← ri mod sizeblk
 3  ldb k1 ← E[t1]
 4  ldw rv ← k1[t2]
; if(v>0) t= ...
 5  br rv ≤ 0 ↪ 3
 6  rt ← rv % 1000
 7  jmp 3
; else t= ...
 8  t1 ← 0 − rv
 9  rt ← t1 % 1000
; c[t]=c[t]+1
10  t1 ← rt ≫ 9
11  t2 ← rt & 511
12  ldb k2 ← O[t1]
13  ldw t3 ← k2[t2]
14  t4 ← t3 + 1
15  stw t4 → k2[t2]
16  stb k2
```

Figure 3.4: $\mathcal{L}_{\text{GhostRider}}$ code implementing (part of) Figure 3.1

We overload ; to operate over two instruction sequences such that $I;(I';\iota) \triangleq (I;I');\iota$ and $I_1;I_2;I_3 \triangleq (I_1;I_2);I_3$.

Note that our formalism does not model the instruction scratchpad; essentially it assumes that all code is loaded on-chip prior to the start of its execution. Section 3.5 discusses how the instruction scratchpad is used in practice.

3.3.2 Example

Figure 3.4 shows $\mathcal{L}_{\text{GhostRider}}$ code that corresponds to the body of the second `for` loop in the source program from Figure 3.1. We write r_X for a register corresponding to variable X in the source program (for simplicity) and write t_i for $i \in \{1, 2, ...\}$ for temporary registers. In the explanation we refer to the names of variables in the source program when describing what the target program is computing.

The first four lines load the `i`th element of array `a` into `v`. Line 1 computes the address of the block in memory that contains the `i`th element of array `a` and

line 2 computes the offset of the element within that block. Here $size_{blk}$ is the size of each block, which is an architecture constant. Line 3 then loads the block from ERAM, and line 4 loads the appropriate value from the loaded block into `v`.

The next five lines implement the conditional. Line 5 jumps three instructions forward if `v` is *not* greater than 0, else it falls through to line 6, which computes `t`. Line 7 then jumps past the else branch, which begins on line 8, which negates `v` to make it positive before computing `t`.

The final seven lines increment `c[t]`. Lines 10–13 are analogous to lines 1–4; they compute the address of `t`th element of array `c` and load it into temporary t_3. Notice that this time the block is loaded from ORAM, not ERAM. Line 14 increments the temporary; line 15 stores it back to the block in the scratchpad; and line 16 stores the entire block back to ORAM.

3.4 Security by typing

This section presents a type system for $\mathcal{L}_T$ that guarantees programs obey the strong *memory trace obliviousness* (MTO) security property.

3.4.1 Memory Trace Obliviousness

Memory trace obliviousness is a noninterference property that also considers the address trace, rather than just the initial and final contents of memory [79]. MTO's definition relies on the notion of *low equivalence* which relates memories whose RAM contents are identical. We formally define this notion below, using the

following formal notation:

$$\begin{aligned} M &\in \textbf{Addresses} \to \textbf{Blocks} \\ a &\in \textbf{Addresses} = \textbf{Labels} \times \textbf{Nat} \\ b &\in \textbf{Blocks} = \textbf{Nat} \to \mathbb{Z} \end{aligned}$$

We model a *memory* M as a map from addresses to blocks, where an address is a pair consisting of a label l (corresponding to an ORAM, ERAM, or RAM bank, as per Figure 3.3) and an address n in that bank. A block is modeled as a map from an address n to a (integer) value. Here is the definition of memory low-equivalence:

Definition 4 (Memory low equivalence)**.** *Two memories M_1, M_2 are* low equivalent*, written $M_1 \sim_L M_2$, if and only if for all n such that $0 \leq n < size(D)$ we have $M_1(D, n) = M_2(D, n)$.*

The definition states that memories M_1 and M_2 are low equivalent when only the RAM bank's values of the memories are the same, but all of the other values could differ.

Intuitively, memory trace obliviousness says two things given two low-equivalent memories. First, if the program will terminate under one memory, then it will terminate under the other. Second, if the program will terminate and lead to a trace t under one memory, then it will do so under the other memory as well while also finishing with low-equivalent memories.

To state this intuition precisely, we need a formal definition of a $\mathcal{L}_T$ execution, which we give as an operational semantics. The semantics is largely stan-

dard, and can be found in the Appendix. The key judgment has the form $I \vdash (R, S, M, pc) \longrightarrow_t (R', S', M', pc')$, which states that program I, with a register file R, a (data) scratchpad S, a memory M, and a program counter pc, executes some number of steps, producing memory trace t and resulting in a possibly modified register file R', scratchpad S', memory M', and program counter pc'.

Definition 5 (Memory trace obliviousness)**.** *A program I is* memory trace oblivious *if and only if for all memories $M_1 \sim_L M_2$ we have $I \vdash (R_0, S_0, M_1, 0) \longrightarrow_{t_1} (R'_1, S'_1, M'_1, pc_1)$, and $I \vdash (R_0, S_0, M_2, 0) \longrightarrow_{t_2} (R'_2, S'_2, M'_2, pc_2)$, and $|t_1| = |t_2|$ implies $t_1 \equiv t_2$ and $M'_1 \sim_L M'_2$.*

Here R_0 is a mapping that maps every register to 0, and S_0 maps every address to a all-0 block. Traces t consist of reads/writes to RAM (both address and value) and ERAM (just the address), accesses to ORAM (just the bank), and instruction fetches. For the last we only model that a fetch happened, not what instruction it is, as we assume code will be stored in a scratchpad on chip. We write $t_1 \equiv t_2$ to say that traces t_1 and t_2 are *indistinguishable* to the attacker; i.e., they consist of the same events in the same order. Our formalism models every instruction as taking *unit* time to execute – thus the trace event also models the time taken to execute the instruction. On the real GhostRider architecture, each instruction takes *deterministic but non-uniform* time; as this difference is conceptually easy to handle (by accounting for instruction execution times in the compiler), we do not model it formally, for simplicity (see Section 3.5).

Sym. vals. $sv \in \textbf{SymVals} = n \mid ? \mid sv_1\ aop\ sv_2 \mid \mathtt{M}_l[k, sv]$
Sym. Store $Sym \in \textbf{Registers} \cup \textbf{Block IDs} \to \textbf{SymVals}$
Sec. Labels $\ell \in \textbf{SecLabels} = \mathtt{L} \mid \mathtt{H}$
Label Map $\Upsilon \in (\textbf{Registers} \to \textbf{SecLabels}) \cup (\textbf{Block IDs} \to \textbf{Labels})$

$\boxed{sv_1 \equiv sv_2}$

$$\frac{\vdash_{safe} sv_1 \quad \vdash_{safe} sv_2 \quad sv_1 = sv_2}{sv_1 \equiv sv_2}$$

Auxiliary Functions

$$select(l, a, b, c) = \begin{cases} a & \text{if } l = D \\ b & \text{if } l = E \\ c & \text{if otherwise.} \end{cases}$$

$$slab(l) = select(l, \mathtt{L}, \mathtt{H}, \mathtt{H})$$

$$ite(x, a, b) = \begin{cases} a & \text{if } x \text{ is true.} \\ b & \text{otherwise.} \end{cases}$$

$\boxed{\vdash_{safe} sv}$

$$\frac{l = D \quad \vdash_{safe} sv}{\vdash_{safe} \mathtt{M}_l[k, sv]} \qquad \vdash_{safe} n$$

$$\frac{\vdash_{safe} sv_1 \quad \vdash_{safe} sv_2}{\vdash_{safe} sv_1\ aop\ sv_2}$$

$\boxed{\vdash_{const} sv}$

$$\vdash_{const} n \qquad \vdash_{const} ?$$

$$\frac{\vdash_{const} sv_1 \quad \vdash_{const} sv_2}{\vdash_{const} sv_1\ aop\ sv_2}$$

$\boxed{\vdash_{const} Sym}$

$$\frac{\forall r.\ \vdash_{const} Sym(r) \quad \forall k.\ \vdash_{const} Sym(k)}{\vdash_{const} Sym}$$

Figure 3.5: Symbolic values, labels, auxiliary judgments and functions

3.4.2 Typing: Preliminaries

Now we give a type system for $\mathcal{L}_{\text{GhostRider}}$ programs and prove that type correct programs are MTO.

Symbolic values To ensure that the execution of a program cannot leak information via its address trace, we statically approximate what events a program can produce. An important determination made by the type system is when a secret

variable can be stored in ERAM—because the address trace will leak no information about it—and when it must be accessed in ORAM. As an example, suppose we had the source program

```
if(s) then x[i]=1 else x[i]=2;
```

If `x` is secret but stored in RAM, then the value in `x[i]` after running this program will leak the contents of secret variable `s`. We could store `x` in ORAM to avoid this problem, but this is unnecessary: both branches will modify the same element of `x`, so encrypting the content of `x` is enough to prevent the address trace from leaking information about `s`. The type system can identify this situation by *symbolically* tracking the contents of the registers, blocks, etc.

To do this, the type rules maintain a *symbolic store* Sym, which is a map from register and block IDs to symbolic values. Figure 3.5 defines symbolic values sv, which consist of constants n, (symbolic) arithmetic expressions, values loaded from memory $\mathtt{M}_l[k, sv]$, and unknowns ?. Most interesting is memory values, which represent the address of a loaded value: l indicates the memory bank it was loaded from, sv corresponds to the offset (i.e., the block number) within that bank, and k is the scratchpad block into which the memory block is loaded.[2]

The type rules also make use of a *label map* Υ mapping registers to security labels and block IDs to (memory) labels; the latter tracks the memory bank from which a scratchpad block was loaded.

The figure defines several judgments; the form of each judgment is boxed. The

[2]In actual traces t, the block number k is not visible; we track it symbolically to model the scratchpad's contents, in particular to ensure that the same memory block is not loaded into two different scratchpad blocks.

first defines when two symbolic values can be deemed equivalent, written $sv_1 \equiv sv_2$: they must be syntactically identical and safe static approximations. The latter is defined by the judgment $\vdash_{safe} sv$, which accepts constants, memory accesses to RAM involving safe indexes, and arithmetic expressions involving safe values. Judgement $\vdash_{const} sv$ says that symbolic value sv is not a memory value. That is, sv is either a constant, a ?, or a binary expression not involving memory values. Further, for a symbolic store Sym, if all the symbolic values that it maps to can be accepted by $\vdash_{const} sv$, then we have $\vdash_{const} Sym$. The latter judgment is needed when checking conditionals.

Finally, we give three auxiliary functions used in the type system. Based on whether l is D, E, or an ORAM bank, function $select(l, a, b, c)$ returns a, b, or c respectively. Function $slab(\cdot)$ maps a normal label l to a security label ℓ, which is either `L` or `H`. The label `H` classifies encrypted memory—any ORAM bank and ERAM—while label `L` classifies RAM. These two labels form the two-point lattice with `L` $\sqsubset$ `H`. Note that `L` is equivalent to the `public` label used in Figure 3.1, and `H` is equivalent to `secret`. Finally, function $ite(x, a, b)$ returns a if x is true, and returns b if x is false.

Trace patterns Figure 3.6 defines *trace patterns* T, which are largely similar to those for $\mathcal{L}_{\text{basic}}$ that approximate traces t. The first line in the definition of T defines single events. The first two indicate reads and writes to RAM or ERAM; they reference the memory bank, block identifier in the scratchpad, and a symbolic value corresponding to the block *address* (not the actual value) read or written. Pattern **F** corresponds to a non memory-accessing instruction. The next pattern indicates

$$\text{Trace Pats. } T ::= \mathbf{read}(l, k, sv) \mid \mathbf{write}(l, k, sv) \mid \mathbf{F} \mid o \mid T_1@T_2 \mid T_1 + T_2 \mid \mathbf{loop}(T_1, T_2)$$

$$\frac{sv_1 \equiv sv_2}{\mathbf{read}(l, k, sv_1) \equiv \mathbf{read}(l, k, sv_2)} \qquad o \equiv o \qquad \frac{T_1 \equiv T_2}{T_2 \equiv T_1} \qquad \mathbf{F} \equiv \mathbf{F}$$

$$\frac{sv_1 \equiv sv_2}{\mathbf{write}(l, k, sv_1) \equiv \mathbf{write}(l, k, sv_2)} \qquad \frac{T_1 \equiv T_1' \quad T_2 \equiv T_2'}{T_1@T_2 \equiv T_1'@T_2'}$$

$$T_1@(T_2@T_3) \equiv (T_1@T_2)@T_3$$

Figure 3.6: Trace patterns and their equivalence in $\mathcal{L}_{\text{GhostRider}}$

a read or write from ORAM bank o: this bank is the trace event itself because the adversary cannot determine whether an access is a read or a write, or which block within the ORAM is accessed. Trace pattern $T_1@T_2$ is the pattern resulting from the concatenation of patterns T_1 and T_2. Pattern $T_1 + T_2$ represents *either* T_1 or T_2, and is used to type conditionals. Finally, pattern $\mathbf{loop}(T_1, T_2)$ represents zero or more loop iterations where the guard's trace is T_1 and the body's trace is T_2.

Trace pattern equivalence $T_1 \equiv T_2$ is defined in Figure 3.6. In this definition, reads are equivalent to other reads accessing exactly the same location; the same goes for writes. Two ORAM accesses to the same ORAM bank are obviously treated as equivalent. Sum patterns specify possibly different trace patterns, and loop patterns do not specify the number of iterations; as such we cannot determine their equivalence statically. The concatenation operator @ is associative with respect to equivalence.

Instructions

$$\text{T-LOAD}\ \frac{\begin{array}{c} l \notin \mathbf{ORAMbanks} \Rightarrow \Upsilon(r) = \mathtt{L} \\ \Upsilon' = \Upsilon[k \mapsto l] \qquad Sym' = Sym[k \mapsto Sym(r)] \\ T_0 = \mathbf{read}(l, k, Sym(r)) \qquad T = select(l, T_0, T_0, l) \end{array}}{\ell \vdash \mathbf{ldb}\ k \leftarrow l[r] : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T}$$

$$\text{T-STORE}\ \frac{\begin{array}{c} Sym(k) = sv \qquad \Upsilon(k) = l \\ T_0 = \mathbf{write}(l, k, sv) \qquad T = select(l, T_0, T_0, l) \end{array}}{\ell \vdash \mathbf{stb}\ k : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon, Sym \rangle; T}$$

$$\text{T-LOADW}\ \frac{\begin{array}{c} l = \Upsilon(k) \qquad \Upsilon(r_2) \sqsubseteq slab(l) \\ \Upsilon' = \Upsilon[r_1 \mapsto slab(l)] \qquad sv = \mathtt{M}_l[k, Sym(r_2)] \\ Sym' = Sym[r_1 \mapsto sv] \end{array}}{\ell \vdash \mathbf{ldw}\ r_1 \leftarrow k[r_2] : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; \mathbf{F}}$$

$$\text{T-STOREW}\ \frac{\ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq slab(\Upsilon(k))}{\ell \vdash \mathbf{stw}\ r_1 \rightarrow k[r_2] : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon, Sym \rangle; \mathbf{F}}$$

$$\text{T-IDB}\ \frac{\begin{array}{c} Sym(k) = sv \qquad \Upsilon(k) = l \\ \Upsilon' = \Upsilon[r \mapsto select(l, \mathtt{L}, \mathtt{L}, \mathtt{H})] \qquad Sym' = Sym[r \mapsto sv] \end{array}}{\ell \vdash r \leftarrow \mathbf{idb}\ k : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; \mathbf{F}}$$

$$\text{T-BOP}\ \frac{\begin{array}{c} \ell' = \Upsilon(r_2) \sqcup \Upsilon(r_3) \qquad \Upsilon' = \Upsilon[r_1 \mapsto \ell'] \\ sv = Sym(r_2)\ aop\ Sym(r_3) \qquad Sym' = Sym[r_1 \mapsto sv] \end{array}}{\ell \vdash r_1 \leftarrow r_2\ aop\ r_3 : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; \mathbf{F}}$$

$$\text{T-ASSIGN}\ \frac{\Upsilon' = \Upsilon[r \mapsto \mathtt{L}] \qquad Sym' = Sym[r \mapsto n]}{\ell \vdash r \leftarrow n : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; \mathbf{F}}$$

$$\text{T-NOP}\quad \ell \vdash \mathbf{nop} : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon, Sym \rangle; \mathbf{F}$$

$$\text{T-SEQ}\ \frac{\begin{array}{c} \ell \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T_1 \\ \ell \vdash \iota : \langle \Upsilon', Sym' \rangle \rightarrow \langle \Upsilon'', Sym'' \rangle; T_2 \end{array}}{\ell \vdash I; \iota : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon'', Sym'' \rangle; T_1 @ T_2}$$

Figure 3.7: Security Type System for $\mathcal{L}_{\text{GhostRider}}$ (Part 1)

3.4.3 Type rules

Figures 3.7, 3.8, 3.9 define the security type system for $\mathcal{L}_T$. The figures are divided into three parts.

Instructions Figure 3.7 presents judgment $\ell \vdash \iota : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T$, which is used to type instructions ι. Here, ℓ is the *security context*, used in the standard way to prevent implicit flows. The rules are *flow sensitive*: The judgement says that instruction ι has a type $\langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle$, and generates trace pattern T. Informally, we can say by executing ι, a state corresponding to security type $\langle \Upsilon, Sym \rangle$ will be changed to have type $\langle \Upsilon', Sym' \rangle$.

Rule T-LOAD types load instructions. The first premise ensures that the contents of register r, the indexing register, are not leaked by the operation. In particular, the loaded memory bank l must either be ORAM, or else the register r may only contain public data (from RAM). In the latter case, there is no issue with leaking r, and in the former case r will not be leaked indirectly by the address of the loaded memory since it is stored in ORAM. The final two premises determine the final trace pattern. When the memory bank l is D or E, then the trace pattern indicates a read event from the appropriate block and address. When reading from an ORAM bank the event is just that bank itself. The other premises in the rule update Υ to map the loaded block k to the label of the memory bank, and update Sym to track the address of the block.

We defer discussion of rule T-STORE for the moment, and look at the next three rules, T-LOADW, T-STOREW, T-IDB, which are used to load and store

values related blocks in the scratchpad. The first two rules resemble standard information flow rules. The second premise of T-LOADW is similar to the first premise of T-LOAD in preventing an indirect leak of index register r_2, which would occur if the label of r_2 was `H` but the label of k was `L`. Likewise, the premise of T-STOREW prevents leaking the contents of r_1 and r_2 into the stored block, and also prevents an implicit flow from ℓ (the security context). As such, these two rules ensure that a block k with label ℓ never contains information from memory labeled ℓ' such that $\ell' \sqsupset \ell$. The remaining premises of Rule T-LOADW flow-sensitively track the label and symbolic value of the loaded register. In particular, they set the label of r_1 to be that of the block loaded, and the symbolic value of r_1 to be the address of the loaded value in memory. T-STOREW changes neither Υ nor Sym: even though the content of the scratchpad has changed, its memory label and its address in memory has not. Both rules emit trace pattern $\mathbf{F}$ as the operations are purely on-chip. We emit this event to account for the time taken to execute an instruction; assuming uniform times for instructions and memory accesses, MTO executions will also be free of timing channels.

Returning to rule T-STORE, we can see that the store takes place unconditionally—no constraints on the labels of the memory or block must be satisfied. This is because the other type rules ensure that all blocks k never contain information higher than their security label ℓ, and thus the block can be written straight to memory having the same security label. That said, information could be leaked through the memory trace, so the emitted trace pattern will differ depending on the label of the block: If the label is D or E then the trace pattern will be a write event, and otherwise it will

Branching

$$\text{T-IF}\ \frac{\begin{array}{c} I = \iota_1; I_t; \iota_2; I_f \quad |I_t| = n_1 - 2 \quad |I_f| + 1 = n_2 \\ \iota_1 = \mathbf{br}\ r_1\ rop\ r_2 \hookrightarrow n_1 \qquad \iota_2 = \mathbf{jmp}\ n_2 \\ \ell' = \ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \\ \ell' \vdash I_t : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T_1 \\ \ell' \vdash I_f : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T_2 \\ \ell' = \mathtt{H} \Rightarrow \left\{ \begin{array}{l} T_1@\mathbf{F} \equiv T_2 \wedge \\ \ell = \mathtt{L} \Rightarrow \vdash_{const} Sym \wedge \\ \forall r.\, \Upsilon'(r) = \mathtt{L} \Rightarrow \vdash_{safe} Sym'(r) \end{array} \right\} \\ T = ite(\ell' = \mathtt{H},\ \mathbf{F}@T_1@\mathbf{F},\ \mathbf{F}@((T_1@\mathbf{F}) + T_2)) \end{array}}{\ell \vdash I : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T}$$

$$\text{T-LOOP}\ \frac{\begin{array}{c} I = I_c; \iota_1; I_b; \iota_2 \\ |I_b| = n_1 - 2 \qquad |I_c| + n_1 = 1 - n_2 \\ \iota_1 = \mathbf{br}\ r_1\ rop\ r_2 \hookrightarrow n_1 \quad \iota_2 = \mathbf{jmp}\ n_2 \\ \ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathtt{L} \\ \ell \vdash I_c : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T_1 \\ \ell \vdash I_b : \langle \Upsilon', Sym' \rangle \to \langle \Upsilon, Sym \rangle; T_2 \end{array}}{\ell \vdash I : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; \mathbf{loop}(T_1, T_2)}$$

Figure 3.8: Security Type System for $\mathcal{L}_{\text{GhostRider}}$ (Part 2)

be the appropriate ORAM event. Leaks via the memory trace are then prevented by T-IF and T-LOOP, discussed shortly.

Rule T-IDB is similar to rule T-LOADW. For the third premise, if l is either D or E, the block k has a public address, and thus the value assigned to register r is public; otherwise, when l is an ORAM bank, the register r is secret.

Rule T-BOP types binary operations, updating the security label of the target register to be the join of labels of the source registers. Rule T-ASSIGN gives the target register label L as constants are not secret. Rules T-NOP is always safe and has no effect on the symbolic store or label environment. All of these operations occur on-chip, and so have pattern **F**. Finally, rule T-SEQ types instruction sequences by composing the symbolic maps, label environments, and traces in the obvious way.

Branching Figure 3.8 presents rules T-IF and T-LOOP for structured control flow. Rule T-IF deals with instruction sequences of the form of $I = \iota_1; I_t; \iota_2; I_f$, where ι_1 is a branching instruction deciding, ι_2 is a jump instruction jumping over the false branch, and I_t and I_f are the true and false branches respectively; the relative offsets n_1 and n_2 are based on the length of these code sequences. We require both branches to have the same type, i.e. $\langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle$, as for the sequence I itself.

When the security context is high, i.e. $\ell = \mathtt{H}$, or when the if-condition is private, i.e. $\Upsilon(r_1) \sqcup \Upsilon(r_2) = \mathtt{H}$, then ℓ' will be $\mathtt{H}$ and we impose three restrictions. First, both of the blocks I_t and I_f must have equivalent trace patterns. (The trace of the true branch is $T_1@\mathbf{F}$ where T_1 covers I_t and $\mathbf{F}$ covers the jump instruction ι_2.) Second, if the security context is public, i.e. $\ell = \mathtt{L}$, then we restrict $\vdash_{const} Sym$ to enforce $Sym(r)$ does not map to memory values. The reason is that in a public context, two equivalent symbolic memory values may refer to two different concrete values, since the memory region D can be modified. Third, for any register r, its value after taking either branch must be the same, or the register r must have a high security label (i.e. $\Upsilon'(r) = \mathtt{H}$). So if $\Upsilon'(r) = \mathtt{L}$, the type system enforces that its symbolic values on the two paths are equivalent, i.e. $Sym'(r) \equiv Sym'(r)$, which only requires $\vdash_{safe} Sym'(r)$.

The final premise for rule T-IF states that the sequence's trace pattern T is either $\mathbf{F}@T_1@\mathbf{F}$ when both branches' patterns must be equal, or else is an or-pattern involving the trace T_2 from the else branch.

Rule T-LOOP imposes structural requirements on I similar to T-IF. The

Subtyping

$$\text{T-SUB}\ \frac{\ell \vdash \iota : \langle \Upsilon, Sym\rangle \to \langle \Upsilon', Sym'\rangle; T \quad \Upsilon' \preceq \Upsilon'' \quad Sym' \preceq Sym''}{\ell \vdash \iota : \langle \Upsilon, Sym\rangle \to \langle \Upsilon'', Sym''\rangle; T}$$

$$\text{S-LABEL}\ \frac{\forall r.\Upsilon(r) \sqsubseteq \Upsilon'(r) \quad \forall k.\Upsilon(k) = \Upsilon'(k)}{\Upsilon \preceq \Upsilon'}$$

$$\text{S-SYM}\ \frac{\forall r.Sym'(r) = ? \vee Sym(r) = Sym'(r) \quad \forall k.Sym'(k) = ? \vee Sym(k) = Sym'(k)}{Sym \preceq Sym'}$$

Figure 3.9: Security Type System for $\mathcal{L}_{\text{GhostRider}}$ (Part 3)

premise $\ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathtt{L}$ implies two restrictions. On the one hand, $\ell \sqsubseteq \mathtt{L}$, prevents any loop from appearing in a secret if-statement, because otherwise the number of loop iterations may leak information about which branch is taken. On the other hand, $\Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathtt{L}$ implies that the loop condition must be public, or otherwise, similarly, the number of iterations would leak secret information about r_1 and/or r_2.

Subtyping Finally, Figure 3.9 presents subtyping rules. Rule T-SUB supports subtyping on the symbolic store and the label map. For the first, a symbolic store Sym can approximated by a store Sym' that either agrees on the symbolic values mapped to be Sym or maps them to ?. For the second, a register's security label can be approximated by one higher in the lattice; block labels may not change. Subtyping is important for typing join points after branches or loops. For example, if a conditional assigned a register r the value 1 in the true branch but assigned r to 2 in the false branch, we would use subtyping to map r to ? to ensure that the

symbolic store at the end of both branches agrees, as required by T-IF.

3.4.4 Security theorem

All well-typed programs are memory-trace oblivious:

Theorem 2. *Given I, Υ, and Sym, if there exists some Υ', Sym' and T such that $\mathtt{L} \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T$, where $\forall r.Sym(r) = ?$ and $\Upsilon(r) = \mathtt{L}$ and $\forall k.Sym(k) = ?$ and $\Upsilon(k) = D$ then program I is memory-trace oblivious.*

The proof can be found in the Appendix B.

3.5 Compilation

We have developed a compiler from an imperative, C-like source language, which we call $\mathcal{L}_S$, to $\mathcal{L}_{\text{GhostRider}}$. Our compiler is implemented in about 7600 lines of Java, with roughly 400 LoC dedicated to the parser, 700 LoC to the type checker, 3500 LoC to the compiler/optimizer, 950 LoC to the code generator, and the remainder to utility functions. This section informally describes our compilation approach.

3.5.1 Source Language

Syntax An $\mathcal{L}_S$ program is a collection of (possibly mutually recursive) functions and a collection of (possibly mutually recursive) type definitions. A type definition is simply a mapping of a type name to a type where types are either natural numbers, arrays, or pointers to records (i.e., C-style `structs`). Each type is annotated with a

security label which is either `secret` or `public` indicating whether the data should be visible/inferrable by the adversary or not.

A function consists of a sequence of statements s which are either no-ops, variable assignments, array assignments, conditionals, while loops, or returns. As usual, conditional branches and loop bodies may consist of one or more statements. Expressions e appearing in statements (e.g., in assignments) consist of variables x, arithmetic ops e_1 *aop* e_2, array reads $e[e]$, and numeric constants n. Variables may hold any data other than functions (i.e., there are no function pointers). Guards in conditionals and while loops consist of predicates involving relational operators.

Typing $\mathcal{L}_S$ programs are type checked before they are compiled. We do this using an information flow-style type system (cf. the survey of Sabelfeld and Myers [79]). As is standard, the type system prevents explicit flows and implicit flows. In particular, it disallows assignments like `p = s` where `p` is a public variable and `s` is a secret variable, and disallows conditionals like `if (s == 0) then p = 0 else p = 1`, which leaks information about `s` since after the conditional the adversary knows `p == 0` implies `s == 0`. It also disallows array writes like `p[s] = 5` since the adversary can learn the value of `s` by seeing which element of the public array has changed. Note that accessing `s[p]` is safe because, despite knowing the index, an adversary cannot learn the value being accessed.

To prevent the length of a memory trace from revealing information, we require that loop guard expressions only involve public values (which is a standard restriction [79]). One can work around this problem by "padding out" loop iterations, e.g., by converting a loop like `while (slen > 0) { sarr[slen--]++; }`

to be `plen = N; while (plen > 0) { if (plen <= slen) sarr[--plen]++; }` where `N` is a large, fixed constant. For similar reasons we also require that whether a function is called or returned from, and which function it is, may not depend on secret information (e.g., the call or return may not occur in a conditional whose guard involves secret information).

Compilation overview After source-language type checking, compilation proceeds in four stages—memory layout, translation, padding, and register allocation—after which the result is type checked using the $\mathcal{L}_{\text{GhostRider}}$ type system, to confirm that it is memory-trace oblivious.[3]

3.5.2 Memory bank allocation

The first stage of compilation allocates global variables to memory banks. Public variables are always stored in RAM, while secret variables will be allocated either to ERAM or ORAM. Two blocks in the scratchpad are reserved for secret and public variables, respectively, that will fit entirely within the block; these are essentially those that contain numbers, (pointers to) records, and small arrays. Such variables will be loaded into the scratchpad at the start of executing a program, and written back to memory at the end. The remaining scratchpad blocks are used for handling (large) arrays; the compiler will always use the same block for the same array. Public arrays are allocated in RAM, and secret arrays always indexed by public values are allocated in ERAM, and ORAM otherwise. The compiler initially

[3]This is essentially a kind of *translation validation* [73], which removes the compiler from the trusted computing base. We believe that well typed $\mathcal{L}_S$ programs yield well typed $\mathcal{L}_{\text{GhostRider}}$ programs, but leave a proof as future work.

assigns a distinct logical ORAM bank for each secret array, and allocates logical banks up to the hardware limit.

3.5.3 Basic compilation

The next stage is basic compilation (translation). Expressions are compiled by loading relevant variables/data into registers, performing the computation, and then storing back the result. Statements are compiled idiomatically to match the structure expected by the type rules in Figure 3.7, Figure 3.8,and Figure 3.9 (with some work deferred to the padding stage).

Perhaps the most interesting part is handling variable accesses. Variables permanently resident in the scratchpad are loaded at the start of the program, and stored back at the end. Each read/write results in a **ldw**, to load a variable into a temporary register, and a **stw** to store back the result. Accesses to data (i.e., arrays) not permanently stored in the scratchpad will also require a **ldb** to load the relevant block into the scratchpad first and likewise a **stb** to store it back. A standard software cache, rather than a scratchpad, could eliminate repeated loads and stores of blocks from memory but could violate MTO. This is because a non-present block will induce memory traffic while a present block will not, and the presence/absence of traffic could be correlated with secret information. To avoid this, we have the compiler emit instructions that perform caching explicitly, using the scratchpad, with caching only enabled when in a public context, i.e., in a portion of code whose control flow does not depend on secret data. To support software-based caching,

the compiler statically maps memory-resident data to particular scratchpad blocks, always loading the same data to the same block. Prior to doing so, and when safe, the compiler uses the **idb** instruction to check whether the relevant scratchpad block contains the memory block we want and loads directly from it, if so.

Supporting functions requires handling calling contexts and local variables. We do this with two stacks, one in RAM and one in ERAM. Function calls are only permitted in a public context, which means that normal stack allocation and deallocation reveal no information, so no ORAM stack is needed. When a function is called, the current scratchpad variable blocks are pushed on the relevant stacks. At the start of a function, we load the blocks that hold the local variables. Local variables implementing ORAM arrays are stored by reference, with the variable pointing to the actual array stored in ORAM. This array is deallocated when its variable is popped from the stack, when the function returns (which like calls are allowed only in a public context).

The compiler is also responsible for emitting instructions that load code into the instruction scratchpad, as implicit instruction fetches could reveal information. (To bootstrap, the first code block is loaded automatically.) At the moment, our compiler emits code that loads the entire program into the scratchpad at the start; we leave to future work support for on-the-fly instruction scratchpad use.

3.5.4 Padding and register allocation

Both branches of a secret conditional must produce the same trace. We ensure they do so by inserting extra instructions in one or both branches according to the solution to the *shortest common supersequence* problem [31]. When matching the two branches, we must account for the memory trace and instruction execution times. Only **ldb** and **stb** emit memory events; we discuss these shortly. While our formalism assumes each instruction takes unit time, the reality is different (cf. Table 3.2): times are deterministic, but non-uniform. For single-cycle operations (e.g., 64b ALU ops), we pad with **nop**s. For two-cycle **ldw** and **stw** instructions, we pad with two **nops**. For multiply and divide instructions, which take 70 cycles each, we could pad with 70 **nops** but this results in a large space overhead. As such, we pad both with the instruction $r0 \leftarrow r0 * r0$, where $r0$ is always 0. For conditionals, we pad the not-taken branch with two **nop**s, to account for the hardware-induced delay on the taken branch.

Padding for **stb** and **ldb** requires instructions that generate matching trace events. An access to ORAM is the simplest to pad, since the adversary cannot distinguish a read from a write. We can load any block (e.g., the first block of the ORAM) into a dedicated "dummy" scratchpad block, i.e. this block is used for loading and saving dummy memory blocks only.

For RAM and ERAM, the address being accessed is visible, so we need to make sure that the equivalent padding accesses the same address. To do this, the compiler should insert further instructions to compute the address. These instructions can

be computed using the symbolic value: (1) if the symbolic value is a constant, then insert an assign instruction; (2) if the symbolic value is a binary operation of two symbolic values, then insert instructions to compute the two symbolic values respectively, and then another instruction to compute the binary operation; and (3) if the symbolic value is a memory value, then insert instructions to compute the offset first, and then insert a **ldw** instruction.

With instructions inserted to compute the address, we must emit either a load or a store depending on the instruction we are trying to match. For RAM, this instruction will always be a load because we perform padding in the `H` context, and the type system prevents writing to RAM. To mimic the $\mathbf{read}(l, k, sv)$ trace pattern, we first compute sv and then insert a **ldb** $k \leftarrow l[r]$ instruction where r stores the value for sv. To handle ERAM writes is challenging because we want the write to be a no-op but not appear to be so. To do this, we require the compiler to *always* follow an ERAM **ldb** with a **stb** back to the same address. In doing so, the compiler also prevents the padded instruction from overwriting a dirty scratchpad block.

At the conclusion of the padding stage we perform standard register allocation to fill in actual registers for the temporaries we have used to this point.

3.6 Hardware Implementation

We implement our deterministic processor by modifying Rocket, a single-issue, in-order, 6-stage pipelined CPU developed at UC Berkeley [77]. Rocket implements the RISC-V instruction set [93] and is comparable to an ARM Cortex A5 CPU.

We modified the baseline processor to remove branch prediction logic (so that conditional branches are always not-taken) and to make each instruction execute in a fixed number of cycles. We describe the remaining changes below.

Instruction-set Extension. We customize RISC-V to add a single data transfer instruction that implements **ldb** and **stb** from the formalism. We do this using a Data Transfer accelerator (Figure 3.2) that attaches to the processor's accelerator interface [88]. We also interface the Data Transfer accelerator with the x86-Linux host through Rocket's control register file so that it can load an `elf`-formatted binary into GhostRider's memory and reset its processor. Once this is done, the host performs processor control register writes to initiate transfers from the co-processor memory to the code ORAM for the code and data sections of the binary. The first code block of a program is loaded into the instruction scratchpad to begin execution; if subsequent instruction blocks are needed they must be loaded explicitly.

Scratchpads. GhostRider has two scratchpads, one for code and one for data, each of which can hold eight 4KB blocks. The instruction scratchpad is implemented similar to an 8-way set-associative cache, where each way contains one block. The accelerator transfers one block at a time to a specified way in the instruction scratchpad. Once a block has been written, the valid and tag bits for that block are updated. The architecture does not implement the **idb** instruction from the formalism; instead, the compiler uses the first 8 bytes of every block to remember its address.

ORAM controller. We implement ORAM by building on the Phantom ORAM controller [63] and implement an ORAM tree 13 levels deep (i.e., 2^{12} leaf buckets),

with 4 blocks per bucket and an effective capacity of 64MB. ORAM controllers include an on-chip *stash* to temporarily buffer ORAM blocks before they are written out to memory. We set this stash to be 128 blocks. The Phantom design (and likewise, Ascend's [27–29]) treats the stash as a cache for ORAM lookups, which is safe when handling timing channels by controlling the memory access rate. GhostRider mitigates timing channels by having the compiler enforce MTO while assuming that events take the same time. As such, we modify Phantom's design to generate an access to a random leaf in case the requested block is found in the stash, to ensure uniform access times.

FPGA Implementation. GhostRider is implemented on one of Convey HC-2ex's [21] four Xilinx Virtex-6 LX760 FPGAs. We measure hardware design size in terms of FPGA *slices* for logic and *Block RAMs* for on-chip memory. A slice comprises four 6-input, 2-output lookup tables (implementing configurable logic) and eight flip-flops (as storage elements) in addition to multiplexers, while each BRAM on Virtex-6 is either an 18Kb or 36Kb SRAM with up to two configurable read-write ports. The GhostRider prototype uses 47,357 such slices (39% of total) to implement both the CPU and the ORAM controller, and requires 685 of 1440 18Kb BRAMs (47.5%). Table 3.1 shows how these resources are broken up between the Rocket CPU and the ORAM controller, with the remaining resources being used by Convey HC-2ex's boilerplate logic to interface with the x86 core and DRAM. Note that this breakdown is a synthesis estimate before place and route.

Our prototype currently supports one data ORAM bank, one code ORAM bank, and one ERAM bank. We do not implement encryption (it is a small, fixed

cost and uninteresting in terms of performance trends), and do not have separate DRAM; all public data is stored in ERAM when running on the hardware.

The Convey machine requires the hardware design to be run at 150 MHz while our ORAM controller prototype currently synthesizes to a maximum operating frequecy of 140MHz. Pending further optimization to meet 150 MHz timing, we run *both* the CPU and the ORAM controller in a 75 MHz clock domain, and use asynchronous FIFOs to connect the ORAM controller to the DDR DRAM controllers.

GhostRider simulator timing model. In addition to demonstrating feasibility with our hardware prototype, we study the effect of GhostRider 's compiler on alternate, more efficient ORAM configurations, e.g., Phantom at 150MHz [63] with two ORAM banks and a distinct (non-encrypting) DRAM bank. Hence we generate a timing model for both the modified processor and ORAM banks based on Phantom's hardware implementation [63], and incorporate the timing model into an ISA-level emulator for the RISC-V architecture; the model is shown in Table 3.2.

	Slices	BRAMs
Rocket	9287 (8.8%)	36 (10.5%)
ORAM	12845 (12.2%)	211 (61.5%)

Table 3.1: FPGA synthesis results on Convey HC-2ex.

3.7 Empirical Evaluation

Programs. Table 3.3 lists all the programs we use in our evaluation. These programs range from standard algorithms to data structures and include predictable,

Feature	Latency (# cycles)
64b ALU	1
Jump taken/not taken	3/1
64b Multiply/Divide	70/70
Load/Store from Scratchpad	2
DRAM (4kB access)	634
Encrypted RAM (4kB access)	662
ORAM 13 levels (4kB block)	4262

Table 3.2: Timing model for GhostRider simulator.

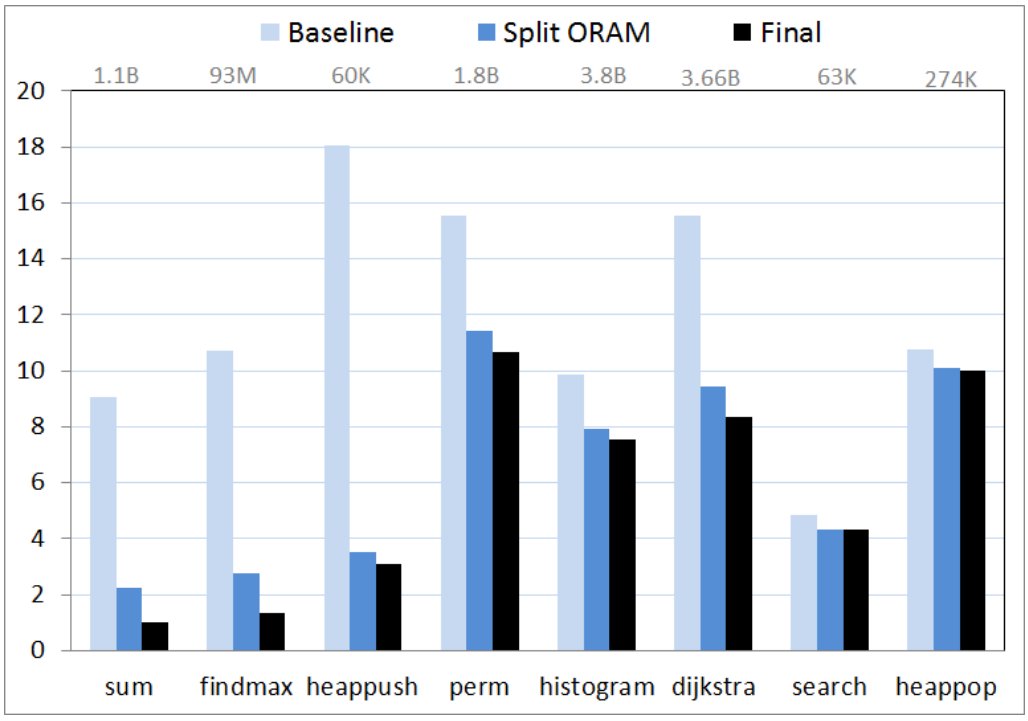

Figure 3.10: Simulator-based execution time results of GhostRider.

partially predictable, and predominantly irregular (data-driven) memory access patterns.

Execution time results. We present measurements both for the simulator and for the actual FPGA hardware, starting with the former because the simulator allows us to evaluate the benefits from splitting memory into ERAM and ORAM banks

Name	Brief Description	Input Size (KB)
sum	Summing up all positive elements in an array	10^3
findmax	Find the max element in an array	10^3
heappush	insert an element into a min-heap	10^3
perm	computing a permutation executing $a[b[i]] = i$ for all i	10^3
histogram	compute the number of occurances of each last digit	10^3
dijkstra	Single-source shortest path	10^3
search	binary search algorithm	1.7×10^4
heappop	pop the minimal element from a min-heap	1.7×10^4

Table 3.3: Benchmark programs for GhostRider organized into programs with predictable, partially predictable, and data dependent memory access patterns (in order from top).

`Non-secure`	Non-secure program: all variables in ERAM, no padding, and uses scratchpad.
`Baseline`	Secure baseline: all secret variables in a single ORAM, no scratchpad.
`Split ORAM`	Variables can be split across multiple ORAM banks, or placed in ERAM. Performs padding. No scratchpad.
`Final`	Scratchpad on top of `Split ORAM`.

Figure 3.11: Legends of Figure 3.10

v. additionally using a scratchpad. We also discuss the execution time results by categorizing them based on the regularity in the programs' access patterns.

Simulator-based results. Figure 3.10 depicts the slowdown of various configurations relative to a non-secure configuration that simply stores data in ERAM and employs the scratchpad. The legends are explained in Figure 3.11. Our non-secure baseline uses a scratchpad instead of a hardware cache in order to isolate the cost of MTO/ORAM. The secure `Baseline` configuration places all secret variables in a single ORAM, while `Split ORAM` employs the GhostRider optimization of using

ERAM and multiple ORAM banks, and `Final` further adds the (secure) use of a scratchpad.

Three out of eight programs—`sum`, `findmax`, and `heap- push`—have a predictable access pattern and the secure program generated by GhostRider relies mainly on ERAM. Hence, each MTO program (`Final`) has almost no slowdown to 3.08× slowdown in comparison its non-secure counterpart (`Non-secure`), and correspondingly faster than `Baseline` by 5.85× to 9.03×.

For `perm`, `histogram`, and `dijkstra`, which have partially predictable and partially sensitive memory access patterns, our compiler attempts to place sensitive arrays inside both ERAM and ORAM and also favors splitting into several smaller ORAM banks without breaking MTO. As shown in Figure 3.10, for such programs, `Final` can achieve a 1.30× to 1.85× speedup over `Baseline` (with 7.56× to 10.68× slowdown compared to `Non-secure`, respectively).

For `search` and `heappop`, which have predominantly sensitive memory access patterns, the speedup of `Final` over `Baseline` is not as significant, i.e. 1.07× and 1.12× respectively, and is due mostly to the usage of two ORAMs to store arrays instead of a single ORAM.

Examining the impact of the use of the scratchpad in the results, we can see that for the first six programs, `Final` reduces execution time compared to `Split ORAM` by a factor from 1.05× up to 2.23×.For `search` and `heappop`, the scratchpad provides no benefit because for these programs all data is allocated in ORAM, as array indices are secret (so the access pattern is sensitive), and our type system disallows caching of ORAM blocks. The reason is that the presence of the data in

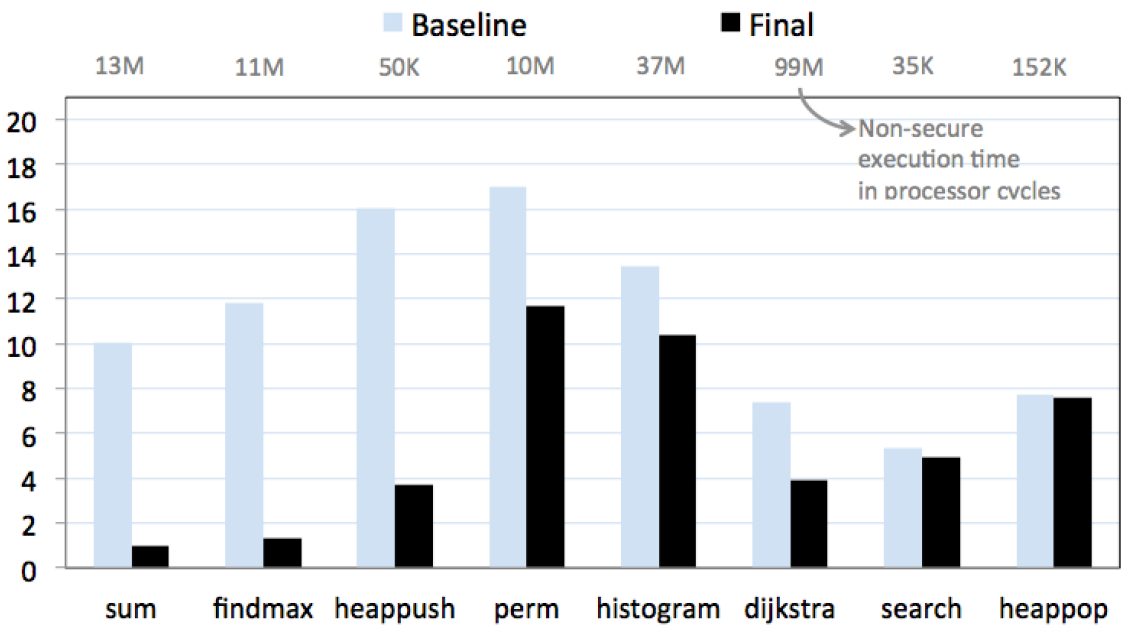

Figure 3.12: FPGA based execution time results: Slowdown of `Baseline` and `Final` versions compared to non-secure version of the program. Note that unlike Figure 3.10, `Final` uses only a single ORAM bank and conflates ERAM and DRAM (cf. Section 3.6).

the cache could reveal something about the secret indices. A more sophisticated type system, or a relaxation of MTO, could do better; we plan to explore such improvements in future work.

FPGA-based results. For the FPGA we run the same set of programs as in Table 3.3, but restrict the input size to be around 100 KB, due to limitations of our prototype. Speedups of `Final` over the secure `Baseline` follow a trend similar to the simulator, as shown in Figure 3.12. Regular programs have speedups in the range of 4.33× (for `heappush`) to 8.94× (for `findmax`). Partially regular programs like `perm` and `histogram` get a speedup of 1.46× and 1.3× respectively. Finally, irregular programs such as `search` and `heappop` see very little improvements (1.08× and 1.02× respectively).

Differences between the simulator and hardware numbers can be attributed to mulitple factors. First, the simulator imperfectly models the Convey memory system's latency, always assuming the worst case, and thus slowdowns compared to the non-secure baseline are often worse on the simulator (cf. `heappop` and `heappush`).

Second, the timing of certain hardware operations is different on the prototype and the simulator (where we consider the latter to be aspirational, per the end of Section 3.6). In particular, per Table 3.2, the simulator models access latency for ORAM as 4262 cycles and ERAM as 662 cycles, accounting for both reading data blocks from DRAM and moving the chosen 4KB block into the scratchpad BRAMs on the FPGA. On the hardware, ORAM and ERAM latencies are 5991 and 1312 cycles, respectively, measured using performance counters in the hardware design. The higher ERAM and ORAM access times reduce the slowdown on the simulator by amplifying the benefit of the scratchpad, which is used by the non-secure baseline, but not by the secure baseline (cf. `findmax` and `sum`).

Third, the benefit of using the scratchpad can differ depending on the input size. This effect is particularly pronounced for Dijkstra, where the ratio of secure to non-secure baseline execution is smaller for the hardware than for the simulator. The reason is that the hardware experiment uses a smaller input that fills only about 1/5 of a scratchpad block. Hence, in the non-secure baseline, the block is reloaded after relatively fewer accesses, resulting in a relatively greater number of block loads and thus bringing the performance of the non-secure program closer to that of the secure baseline.

Finally, note that the simulator's use of multiple ORAM banks, and DRAM

with different timings, is a source of differences, but this effect is dwarfed by the other effects.

3.8 Conclusion

We have presented the first complete memory trace oblivious system—GhostRider—comprising of a novel compiler, type system, and hardware architecture. The compiled programs not only provably satisfy memory trace obliviousness, but also exhibit up to nearly order-of-magnitude performance gains in comparison with placing all variables in a single ORAM bank. By enabling compiler analyses to target a joint ERAM-ORAM memory system, and by employing a compiler-controlled scratchpad, this work opens up several performance optimization opportunities in tuning bank configurations (size and access granularity) and, on a broader level, into co-designing data structures and algorithms for a heterogeneous yet oblivious memory hierarchy.

Chapter 4: RAM-model Secure Computation

Secure computation is a cryptographic technique allowing mutually distrusting parties to make collaborative use of their local data without harming privacy of their individual inputs. Since the first system for general-purpose secure two-party computation was built in 2004 [65], efficiency has improved substantially [11, 46].

Almost all previous implementations of general-purpose secure computation assume the underlying computation is represented as a *circuit*. While theoretical developments using circuits are sensible (and common), typical programs are mostly expressed in von Neumann-style Random Access Machine (RAM) model. Compiling a RAM-model program into its efficient circuit-based representation can be challenging, especially when handling *dynamic memory accesses* to an array in which the memory location being read/written depends on secret inputs. Existing program-to-circuit compiler typically makes an entire copy of the array upon every dynamic memory access, thus resulting in a huge circuit when the data size is large.

To address this limitations, recent research [38] shows that secure computation ORAM can be used to compile a dynamic memory access into a circuit with poly-logarithmic size while preventing information leakage through memory-access patterns. We refer such an approach as a RAM-model secure computation (RAM-

SC) approach, and Gordon et al. [38] observed that RAM-SC exhibits a significant advantage in the setting of *repeated sublinear-time queries* (e.g., binary search) on a large database, where an initial setup cost can be amortized over subsequent repeated queries.

Our Contributions. We continue work on secure computation in the RAM model, with the goal of providing a complete system that takes a program written in a high-level language and compiles it to a protocol for secure two-party computation of that program.[1] In particular, we

- Define an *intermediate representation* (which we call SCVM) suitable for efficient two-party RAM-model secure computation;
- Develop a *type system* ensuring that any well-typed program will generate a RAM-SC protocol secure in the semi-honest model, if all subroutines are implemented with a protocol secure in the semi-honest model.
- Build an *automated compiler* that transforms programs written in a high-level language into a secure two-party computation protocol, and integrate compile-time optimizations crucial for improving performance.

We use our compiler to compile several programs including Dijkstra's shortest-path algorithm, KMP string matching, binary search, and more. For moderate data sizes (up to the order of a million elements), our evaluation shows a speedup of *1–2 orders of magnitude* as compared to standard circuit-based approaches for securely

[1] Note that Gordon et al. [38] do not provide such a compiler; they only implement RAM-model secure computation for the particular case of binary search.

computing these programs. We expect the speedup to be even greater for larger data sizes.

SCVM is our first attempt to demonstrate the feasibility to optimize secure computation in the RAM-model using ORAMs. In the next Chapter, we extend SCVM to build ObliVM which focuses more on richer expressiveness power and easy-programmability while achieving the state-of-the-art performance for secure computation.

This chapter is based on a paper that I co-authored with Michael Hicks, Yan Huang, Jonathan Katz, and Elaine Shi [59]. I developed the formalism and the proof under the help of Michael Hicks, Elaine Shi and Jonathan Katz. I developed the compiler, which implements the optimization and the type checker, and emits code that is runnable over a secure computation backend. I conducted experiments to show the compiler's effectiveness with the help of Yan Huang.

4.1 Technical Highlights

As explained in Sections 4.2 and 4.3, the standard implementation of RAM-SC entails placing all data and instructions inside a single Oblivious RAM. The secure evaluation of one instruction then requires *i)* fetching instruction and data from ORAM; and *ii)* securely executing the instruction using a universal next-instruction circuit (similar to a machine's ALU). This approach is costly since each step must be done using a secure-computation sub-protocol.

An efficient representation for RAM-SC. Our type system and SCVM inter-

mediate representation are capable of expressing RAM-SC tasks more efficiently by avoiding expensive next-instruction circuits and minimizing ORAM operations when there is no risk to security. These language-level capabilities allow our compiler to apply compile-time optimizations that would otherwise not be possible. Thus, we not only obtain better efficiency than circuit-based approaches, but we also achieve order-of-magnitude performance improvements in comparison with straightforward implementations of RAM-SC (see Section 4.2).

Program-trace simulatability. A well-typed program in our language is guaranteed to be both *instruction-trace oblivious* and *memory-trace oblivious*. Instruction-trace obliviousness ensures that the values of the program counter during execution of the protocol do not leak information about secret inputs other than what is revealed by the output of the program. As such, the parties can avoid securely evaluating a universal next-instruction circuit, but can instead simply evaluate a circuit corresponding to the current instruction. Memory-trace obliviousness ensures that memory accesses observed by one party during the protocol's execution similarly do not leak information about secret inputs other than what is revealed by the output. In particular, if access to some array does not depend on secret information (e.g., it is part of a linear scan of the array), then the array need not be placed into ORAM.

We formally define the security property ensured by our type system as *program-trace simulatability*. We define a mechanism for compiling programs to protocols that rely on certain ideal functionalities. We prove that if every such ideal functionality is instantiated with a semi-honest secure protocol computing that functionality, then any well-typed program compiles to a semi-honest secure protocol computing that

program.

Additional language features. SCVM supports several other useful features. First, it permits *reactive* computations by allowing output not only at the end of the program's execution, but also while it is in progress. Our notation of program-trace simulatability also fits this reactive model of computation.

SCVM also integrates state-of-the-art optimization techniques that have been suggested previously in the literature. For example, we support public, local, and secure modes of computation, a technique recently explored (in the circuit model) by Kerschbaum [52] and Rastogi et al. [76] Our compiler can identify and encode portions of computation that can be safely performed in the clear or locally by one of the parties, without incurring the cost of a secure-computation sub-protocol.

Our SCVM intermediate representation generalizes circuit-model approaches. For programs that do not rely on ORAM, our compiler effectively generates an efficient circuit-model secure-computation protocol. This paper focuses on the design of the intermediate representation language and type system for RAM-model secure computation, as well as the compile-time optimization techniques we apply. Our work is complementary to several independent, ongoing efforts focused on improving the cryptographic back end.

4.2 Background: RAM-Model Secure Computation

In this section, we review some background for RAM-model secure computation. Our treatment is adapted from that of Gordon et al. [38], with notation

adjusted for our purposes.

A key underlying building block of RAM-model secure computation is *Oblivious RAM (ORAM)*. ORAM is a cryptographic primitive that hides memory-access patterns by randomly reshuffling data in memory. With ORAM, each memory read or write operation incurs $\text{poly}\log n$ actual memory accesses.

Existing RAM-model secure computation, which we refer as straightforward RAM-SC, employs the following scheme. The entire memory denoted `mem`, containing both program instructions and data, is placed in ORAM, and the ORAM is secret-shared between the two participating parties as discussed above, e.g., using a simple XOR-based secret-sharing scheme. With ORAM, a memory access thus requires each party to access the elements of their respective arrays at pseudorandom locations (the addresses are dictated by the ORAM algorithm), and the value stored at each position is then obtained by XORing the values read by each of the parties. Alternatively, the server can hold an encryption of the ORAM array, and the client holds the key. The latter was done by Gordon et al. to ensure that one party holds only $O(1)$ state. All CPU states are also secret-shared between the two parties.

Straightforward RAM-SC proceeds as follows. Each step of the computation must be done using some secure computation subprotocol. In particular, SC-U is a secure computation protocol that securely evaluates the *universal next instruction circuit*, and SC-ORAM is a secure computation protocol that securely evaluates the ORAM algorithm. For `ORAM.Read`, each party supplies a secret share of the `raddr`, and during the course of the protocol, the `ORAM.Read` protocol will emit obfuscated

	Scenario	Potential benefits of RAM-model secure computation
1	Repeated sublinear queries over a large dataset (e.g., binary search, range query, shortest path query)	• Amortize preprocessing cost over multiple queries • Achieve *sublinear* amortized cost per query
2	One-time computation over a large dataset	Avoid paying $O(n)$ cost per dynamic memory access

Table 4.1: Two main scenarios and advantages of RAM-model secure computation

addresses for each party to read from. At the end of the protocol, each party obtains a share of the fetched data. For `ORAM.Write`, each party supplies a secret share of `waddr` and `wdata`, and during the course of the protocol, the `ORAM.Read` protocol will emit obfuscated addresses for each party to write to, and secret shares of values to write to those addresses.

Deployment scenarios and threat model for RAM-model secure computation. SCVM presently supports a two-party semi-honest protocol. We consider the following primary deployment scenarios:

1. Two parties, Alice and Bob, each comes with their own private data, and engage in a two-party protocol. For example, Goldman Sachs and Bridgewater would like to perform joint computation over their private market research data to learn market trends.

2. One or more users break their private data (e.g., genomic data) into secret shares, and split the shares among two non-colluding cloud providers. The shares at each cloud provider are completely random and reveal no information. To perform computation over the secret-shared data, the two cloud

providers engage in a secure 2-party computation protocol.

3. Similar as the above, but the two servers are within the same cloud or under the same administration. This can serve to mitigate Advanced Persistent Threats or insider threats, since compromise of a single machine will no longer lead to the breach of private data. Similar architectures have been explored in commercial products such as RSA's distributed credential protection [3].

In the first scenario, Alice and Bob should not learn anything about each other's data besides the outcome of the computation. In the second and third scenarios, the two servers should learn nothing about the users' data other than the outcome of the computation – note that the outcome of the computation can also be easily hidden simply by XORing the outcome with a secret random mask (like a one-time pad). We assume that the program text (i.e., code) is public.

With respect to the types of applications, while Gordon et al. describe RAM-model secure computation mainly for the amortized setting, where repeated computations are carried out starting from a single initial dataset, we note that RAM-model secure computation can also be meaningful for one-time computation on large datasets, since a straightforward RAM-to-circuit compiler would incur linear (in the size of dataset) overhead for every dynamic memory access whose address depends on sensitive inputs. Table 4.1 summarizes the two main scenarios for RAM-model secure computation, and potential advantages of using the RAM model in these cases.

4.3 Technical Overview: Compiling for RAM-Model Secure Computation

This section describes our approach to optimize RAM-model secure computation. Our key idea is use static analysis during compilation to minimize the use of heavyweight cryptographic primitives such as garbled circuits and ORAM.

4.3.1 Instruction-Trace Obliviousness

The standard RAM-model secure computation protocol described in Section 4.2 is relatively inefficient because it requires a secure-computation sub-protocol to compute the universal next-instruction circuit U. This circuit has large size, since it must interpret every possible instruction. In our solution, we will avoid relying on a universal next-instruction circuit, and will instead arrange things so that we can securely evaluate instruction-specific circuits.

Note that it is not secure, in general, to reveal what instruction is being carried out at each step in the execution of some program. As a simple example, consider a branch over a secret value `s`:

```
if(s) x[i]:=a+b; else x[i]:=a-b
```

Depending on the value of `s`, a different instruction (i.e., `add` or `subtract`) will be executed. To mitigate such an implicit information leak, our compiler transforms a program to an *instruction-trace oblivious* counterpart, i.e., a program whose program-counter value (which determines which instruction will be executed next)

does not depend on secret information. The key idea there is to use a **mux** operation to rewrite a secret if-statement. For example, the above code can be re-factored to the following:

```
t1 := s;

t2 := a+b;

t3 := a-b;

t4 := mux(t1, t2, t3);

x[i] := t4
```

At every point during the above computation, the instruction being executed is pre-determined, and so does not leak information about sensitive data. Instruction-trace obliviousness is similar to *program-counter security* proposed by Molnar et al. [67] (for a different application).

4.3.2 Memory-Trace Obliviousness

Using ORAM for memory accesses is also a heavyweight operation in RAM-model secure computation. The standard approach is to place *all* memory in a single ORAM, thus incurring $O(\text{poly}\log n)$ cost per data operation, where n is a bound on the size of the memory.

We have demonstrated in the context of securing remote execution against physical attacks (Chapter 2,3) that not all access patterns of a program are sensitive. For example, a `findmax` program that sequentially scans through an array to find the maximum element has predictable access patterns that do not depend on sensitive

inputs. We propose to apply a similar idea to the context of RAM-model secure computation. Our compiler performs static analysis to detect safe memory accesses that do not depend on secret inputs. In this way, we can avoid using ORAM when the access pattern is independent of sensitive inputs. It is also possible to store various subsets of memory (e.g., different arrays) in different ORAMs, when information about which portion of memory (e.g., which array) is being accessed does not depend on sensitive information.

4.3.3 Mixed-Mode Execution

We also use static analysis to partition a program into code blocks, and then for each code block use either a public, local, or secure mode of execution (described next). Computation in public or local modes avoids heavyweight secure computation. In the intermediate language, each statement is labeled with its mode of execution.

Public mode. Statements computing on publicly-known variables or variables that have been declassified in the middle of program execution can be performed by both parties independently, without having to resort to a secure-computation protocol. Such statements are labeled `P`. For example, the loop iterators (in lines 1, 3, 10) in Dijkstra's algorithm (see Figure 4.2) do not depend on secret data, and so each party can independently compute them.

Local mode. For statements computing over Alice's variables, public variables, or previously declassified variables, Alice can perform the computation independently

```
1 for(i = 0; i < n; ++i) {
2   int bestj = -1; bestdis = -1;
3   for(int j=0; j<n; ++j) {
4     if( !  vis[j] && (bestj < 0
5       || dis[j] < bestdis))
6       bestj = j;
7       bestdis = dis[j];
8   }
9   vis[bestj] = 1;
10   for(int j=0; j<n; ++j) {
11     if( !vis[j] && (bestdis +
12        e[bestj][j] < dis[j]))
13        dis[j] = bestdis + e[bestj][j];
14   }
15 }
```

Figure 4.1: Dijkstra's shortest distance algorithm in source (Part)

without interacting with Bob (and vice versa). Here we crucially rely on the fact that we assume semi-honest behavior. Alice-local statements are labeled `A`, and Bob-local statements are labeled `B`.

Secure mode. All other statements that depend on variables that must be kept secret from both Alice and Bob will be computed using secure computation, making ORAM accesses along the way if necessary. Such statements are labeled `O` (for "oblivious").

4.3.4 Example: Dijkstra's Algorithm

In Figure 4.2, we present a complete compilation example for part of Dijkstra's algorithm in Figure 4.1. Here one party, Alice, has a private graph represented by a pairwise edge-weight array `e[][]` and the other party, Bob, has a private source/destination pair. Bob wishes to compute the shortest path between his source

```
O: orame :=oram( e );
P: i :=0;  P: cond1 := i < n;
P:while( cond1 ) do
    O:bestj:=-1;  O:bestdis:=-1;
    P: j :=0;  P: cond2 := j < n;
    P:while( cond2 ) do
        O:t1:= vis [ j ];  O:t2:=!t1;  O:t3:=best<0;
        O:t4:= dis [ j ];  O:t5:=t4<bestdis;
        O:t6:=t3||t5;  O:cond3:=t2 && t6;
        O:best   :=mux(cond3,  j , best);
        O:bestdis:=mux(cond3, t4, bestdis);
        P: j := j +1;  P: cond2 := j < n ;;
    O:  vis [ bestj ]:=1;
    P: j :=0;  P: cond2 := j < n ;
    P:while(cond2) do
        O:t7:= vis [ j ];  O:t8:=!t7;
        O:idx:=bestj*n;  O:idx:=idx+ j ;  O:t9:= orame [idx];
        O:t10:=bestdis + t9;  O:t11:= dis [ j ];
        O:t12:=t10 < t11;  O:cond4:=t8 && t12
        O:t13:=mux(cond4, t10, t11);  O: dis [ j ]:=t13;
        P: j := j +1;  P: cond2 := j < n ;
```

Figure 4.2: **Compilation example: Part of Dijkstra's shortest-path algorithm.** The code on the left is compiled to the annotated code on the right. Array variable e is Alice's local input array containing the graph's edge weights; Bob's input, a source/destination pair, is not used in this part of the algorithm. Array variables vis and orame are placed in ORAMs. Array variable dis is placed in non-oblivious (but secret-shared) memory. (Prior to the shown code, vis is initialized to all zeroes except that vis[source]—where source is Bob's input—is initialized to 1, and dis[i] is initialized to e[source][i].) Variables n, i, j and others boxed in white background are public variables. All other variables are secret-shared between the two parties.

and destination in Alice's graph. The figure shows the code that computes shortest paths (Bob's inputs are elided).

Our specific implementation of Dijkstra's algorithm uses three arrays, a `dis` array which keeps track of the current shortest distance from the source to any other node; an edge-weight array `orame` which is initialized by Alice's local array `e`, and an indicator array `vis`, denoting whether each node has been visited. In this case, our compiler places arrays `vis` and `e` in separate ORAMs, but does not place array `dis` in ORAM since access to `dis` always follows a sequential pattern.

Note that parts of the algorithm can be computed publicly. For example, all the loop iterators are public values; therefore, loop iterators need not be secret-shared, and each party can independently compute the current loop iteration. The remaining parts of program all require ORAM accesses; therefore, our compiler annotates these instructions to be run in secure mode, and generates equivalent instruction- and memory-trace oblivious target code.

4.4 SCVM Language

This section presents SCVM, our language for RAM-model secure computation, and presents our formal results.

In Section 4.4.1, we present SCVM's formal syntax. In Section 4.4.2, we give a formal, *ideal world* semantics for SCVM that forms the basis of our security theorem. Informally, each party provides their inputs to an ideal functionality $\mathcal{F}$ that computes the result and returns to each party its result and a trace of events it is

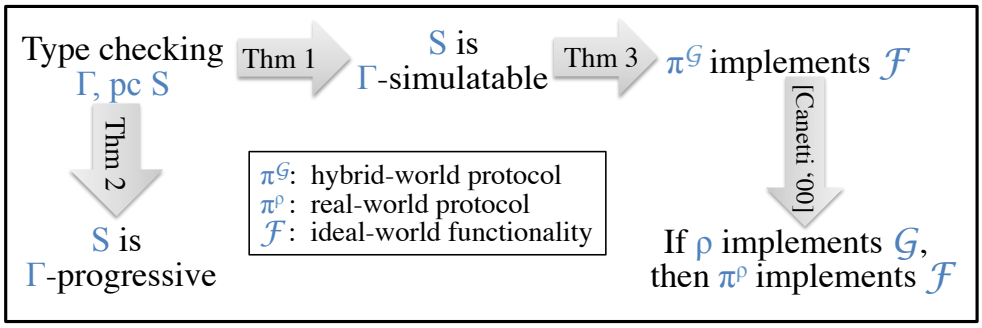

Figure 4.3: Formal results in SCVM.

allowed to see; these events include instruction fetches, memory accesses, and *declassification* events, which are results computed from both parties' data. Section 4.4.3 formally defines our security property, Γ-*simulatability*. Informally, a program is secure if each party, starting with its own inputs, memory, the program code, and its trace of declassification events, can simulate (in polynomial time) its observed instruction traces and memory traces without knowing the other party's data. We present a type system for SCVM programs in Section 4.4.4, and in Theorem 3 prove that well-typed programs are Γ-simulatable. Theorem 4 additionally shows that well-typed programs will not get *stuck*, e.g., because one party tries to access memory unavailable to it. Finally, in Section 4.4.5 we define a *hybrid world* functionality that more closely models SCVM's implemented semantics using ORAM, garbled circuits, etc. and prove that for Γ-simulatable programs, the hybrid-world protocol securely implements the ideal functionality. The formal results are summarized in Figure 4.3.

4.4.1 Syntax

The syntax of SCVM is given in Figure 4.4. In SCVM, each variable and statement has a security label from the lattice {P,A,B,O}, where $\sqsubseteq$ is defined to be the smallest partial order such that $\texttt{P} \sqsubseteq l \sqsubseteq \texttt{O}$ for $l \in \{\texttt{A}, \texttt{B}\}$. The label of each variable indicates whether its memory location should be public, known to either Alice or Bob (only), or secret. For readability, we do not distinguish between oblivious secret arrays and non-oblivious secret arrays at this point, and simply assume that all secret arrays are oblivious. Support for non-oblivious, secret arrays will be added in Section 4.5.

An information-flow control type system, which we discuss in Section 4.4.4, enforces that information can only flow from low (i.e., lower in the partial order) security variables to high security variables. For example, for a statement $x := y$ to be secure, y's security label should be less than or equal to x's security label. An exception is the declassification statement $x := \textbf{declass}_l(y)$ which may declassify a variable y labeled O to a variable x with lower security label l.

The label of each statement indicates the statement's mode of execution. A statement with the label P is executed in *public mode*, where both Alice and Bob can see its execution. A statement with the label A or B is executed in *local mode*, and is visible to only Alice or Bob, respectively. A statement with the label O is executed securely, so both Alice and Bob know the statement was executed but do not learn the underlying values that were used.

Most SCVM language features are standard. We highlight the statement $x :=$

Variables	x, y, z	$\in$	**Vars**
Security Labels	l	$\in$	**SecLabels** $= \{\mathtt{P}, \mathtt{A}, \mathtt{B}, \mathtt{O}\}$
Numbers	n	$\in$	**Nat**
Operation	op	$::=$	$+ \mid - \mid ...$
Expressions	e	$::=$	$x \mid n \mid x\ op\ x \mid x[x] \mid \mathbf{mux}(x, x, x)$
Statements	s	$::=$	$\mathbf{skip} \mid x := e \mid x[x] := x \mid \mathbf{if}\ (x)\ \mathbf{then}\ S\ \mathbf{else}\ S \mid$ $\mathbf{while}\ (x)\ \mathbf{do}\ S \mid x := \mathbf{declass}_l(y) \mid x := \mathbf{oram}(y)$
Labeled Statements	S	$::=$	$l : s \mid S; S$

Figure 4.4: Syntax of SCVM

oram(y), by which variable x is assigned to an ORAM initialized with array y's contents, and the expression **mux**(x_0, x_1, x_2), which evaluates to either x_1 or x_2, depending on whether x_0 is 0 or 1.

4.4.2 Semantics

We define a formal semantics for SCVM programs which we think of as defining a computation carried out, on Alice and Bob's behalf, by an *ideal* functionality $\mathcal{F}$. However, as we foreshadow throughout, the semantics is endowed with sufficient structure that it can be interpreted as using the mechanisms (like ORAM and garbled circuits) described in Sections 4.3. We discuss such a *hybrid world* interpretation more carefully in Section 4.4.5 and prove it also satisfies our security properties.

Memories and types. Before we begin, we consider a few auxiliary definitions given in Figure 4.5. A memory M is a partial map from variables to value-label pairs. The value is either a natural number n or an array m, which is a partial map from naturals to naturals. The security labels $l \in \{\mathtt{P},\mathtt{A},\mathtt{B},\mathtt{O}\}$ indicate the conceptual

visibility of the value as described earlier. Note that in a real-world implementation, data labeled `O` is stored in ORAM and secret-shared between Alice and Bob, while other data is stored locally by Alice or Bob. We sometimes find it convenient to project memories whose values are visible at particular labels:

Definition 6 (L-projection)**.** *Given memory M and a set of security labels L, we write $M[L]$ as M's L-*projection*, which is itself a memory such that for all x, $M[L](x) = (v, l)$ if and only if $M(x) = (v, l)$ and $l \in L$.*

We define types **Nat** l and **Array** l, for numbers and arrays, respectively, where l is a security label. A *type environment* Γ associates variables with types, and we interpret it as a partial map. We sometimes consider when a memory is consistent with a type environment Γ:

Definition 7 (Γ-compatibility)**.** *We say a memory M is* Γ-compatible *if and only if for all x, when $M(x) = (v, l)$, then $v \in \textbf{\textit{Nat}} \Leftrightarrow \Gamma(x) = \textbf{\textit{Nat}}\ l$ and $v \in \textbf{\textit{Array}} \Leftrightarrow \Gamma(x) = \textbf{\textit{Array}}\ l$.*

Ideal functionality. Once Alice and Bob have agreed on a program S, we imagine an ideal functionality $\mathcal{F}$ that executes S. Alice and Bob send to $\mathcal{F}$ memories M_A and M_B, respectively. Alice's memory contains data labeled `A` and `P`, while Bob's memory contains data labeled `B` and `P`. (Data labeled `O` is only constructed during execution.) $\mathcal{F}$ then proceeds as follows:

1. It checks that M_A and M_B agree on `P`-labeled values, i.e., that $M_A[\{\texttt{P}\}] = M_B[\{\texttt{P}\}]$. It also checks that they do not share any `A`/`B`-labeled values, i.e.,

that the domain of $M_A[\{\texttt{A}\}]$ and the domain of $M_B[\{\texttt{B}\}]$ do not intersect. If either of these conditions fail, $\mathcal{F}$ notifies both parties and aborts the execution. Otherwise, it constructs memory M from M_A and M_B:

$$M = \{x \mapsto (v, l) \mid M_A[\{\texttt{A,P}\}](x) = (v, l) \ \vee M_B[\{\texttt{B}\}](x) = (v, l)\}$$

2. $\mathcal{F}$ executes S according to semantics rules having the form $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : D$. This judgment states that starting in memory M, statement S runs, producing a new memory M' and a new statement S' (representing the partially executed program) along with *instruction traces* i_a and i_b, *memory traces* t_a and t_b, and *declassification event* D. We discuss these traces/events shortly. The ideal execution will produce one of three outcomes (or fail to terminate):

   - $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : D$, where $D = (d_a, d_b)$. In this case, $\mathcal{F}$ outputs d_a to Alice, and d_b to Bob. Then $\mathcal{F}$ sets M to M' and S to S' and restarts step 2.
   - $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', l : \textbf{skip}\rangle : \epsilon$. In this case, $\mathcal{F}$ notifies both parties that computation finished successfully.
   - $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : \epsilon$, where $S' \neq l : \textbf{skip}$, and no rules further reduce $\langle M', S'\rangle$. In this case, $\mathcal{F}$ aborts and notifies both parties.

Notice that the only communications between $\mathcal{F}$ and each party about the computation are declassifications d_a and d_b (to Alice and Bob, respectively) and notification

$$
\begin{array}{lcll}
\text{Arrays} & m & \in & \mathbf{Array} = \mathbf{Nat} \rightharpoonup \mathbf{Nat} \\
\text{Memory} & M & \in & \mathbf{Vars} \rightharpoonup (\mathbf{Array} \cup \mathbf{Nat}) \times \mathbf{SecLabels} \\
\text{Type} & \tau & ::= & \mathbf{Nat}\ l \mid \mathbf{Array}\ l \\
\text{Type Environment} & \Gamma & ::= & x : \tau \mid \cdot \\
\text{Instruction Traces} & i & ::= & l : x := e \mid l : x[x] := x \mid l : \mathbf{declass}(x, y) \mid l : \mathbf{init}(x, y) \\
 & & & l : \mathbf{if}(x) \mid l : \mathbf{while}(x) \mid i@i \mid \epsilon \\
\text{Memory Traces} & t & ::= & \mathbf{read}(x, n) \mid \mathbf{readarr}(x, n, n) \mid \\
 & & & \mathbf{write}(x, n) \mid \mathbf{writearr}(x, n, n) \mid x \mid t@t \mid \epsilon \\
\text{Declassification} & d & ::= & (x, n) \mid \epsilon \\
\text{Declass. event} & D & ::= & (d, d) \mid \epsilon
\end{array}
$$

$$
\mathit{select}(l, t_1, t_2) = \begin{cases} (t_1, t_1) & \text{if } l = \mathtt{P} \\ (t_1, \epsilon) & \text{if } l = \mathtt{A} \\ (\epsilon, t_1) & \text{if } l = \mathtt{B} \\ (t_2, t_2) & \text{if } l = \mathtt{O} \end{cases}
$$

$$
\mathit{inst}(l, i) = \mathit{select}(l, l : i, l : i)
$$

$$
\mathit{get}(m, i) = \begin{cases} m(i) & 0 \leq i < |m| \\ 0 & \text{otherwise} \end{cases}
$$

$$
\mathit{set}(m, i, v) = \begin{cases} m[i \mapsto v] & 0 \leq i < |m| \\ m & \text{otherwise} \end{cases}
$$

$$
\mathit{arr}(x, m) = \mathbf{readarr}(x, 0, m(0))@...@\mathbf{readarr}(x, n, m(n)) \quad \text{where } n = |m| - 1
$$

$$
\boxed{t_1 \equiv t_2} \quad t \equiv t \quad t@\epsilon \equiv \epsilon@t \equiv t \quad \frac{t_1 \equiv t_1' \qquad t_2 \equiv t_2'}{t_1@t_2 \equiv t_1'@t_2'}
$$

Figure 4.5: Auxiliary syntax and functions for SCVM semantics

of termination. This is because we assume that secure programs will always explicitly declassify their final output (and perhaps intermediate outputs, e.g., when processing multiple queries), while all other variables in memory are not of consequence. The memory and instruction traces, though not explicitly communicated by $\mathcal{F}$, will be visible in a real implementation (described later), but we prove that they provide no additional information beyond that provided by the declassification events.

Traces and events. The formal semantics incorporate the concept of traces to define information leakage. There are three types of traces, all given in Figure 4.5. The first is an instruction trace i. The instruction trace generated by an assignment statement is the statement itself (e.g., $x := e$); the instruction trace generated by a branching statement is denoted $\textbf{if}(x)$ or $\textbf{while}(x)$. Declassification and ORAM initialization will generate instruction traces $\textbf{declass}(x, y)$ and $\textbf{init}(x, y)$, respectively. The trace ϵ indicates an unobservable statement execution (e.g., Bob cannot observe Alice executing her local code). Trace equivalence (i.e. $t_1 \equiv t_2$) is defined in Figure 4.5.

The second sort of trace is a memory trace t, which captures reads or writes of variables visible to one or the other party. Here are the different memory trace events:

- `P`: Operations on public arrays generate memory event $\textbf{readarr}(x, n, v)$ or $\textbf{writearr}(x, n, v)$ visible to both parties, including the variable name x, the index n, and the value v read or written. Operations on public variables generate memory event $\textbf{read}(x, v)$ or $\textbf{write}(x, v)$. To initialize an ORAM from a public array will access each item in the array, so a sequence of $\textbf{readarr}(x, i, m(i))$ for $i = 0, ..., |m| - 1$, is visible to both Alice and Bob. We use $arr(x, m)$ to indicate such a sequence of memory events.

- `A`/`B`: Operations on Alice's secret arrays generate memory event $\textbf{readarr}(x, n, v)$ or $\textbf{writearr}(x, n, v)$ visible to Alice only. Operations on Alice's secret variables generate memory event $\textbf{read}(x, v)$ or $\textbf{write}(x, v)$ visible to Alice only. Initial-

izing an ORAM from Alice's secret array generate memory events $arr(x, m)$ visible to Alice only. Operations on Bob's secret arrays/variables are handled similarly.

- `O`: Operations on a secret array generate memory event x visible to both Alice and Bob, containing only the variable name, but not the index or the value. A special case is the initialization of ORAM bank x with y's value: a memory trace y, but not its content, is observed.

Memory-trace equivalence is defined similarly to instruction-trace equivalence.

Finally, each declassification executed by the program produces a declassification event (d_a, d_b), where Alice learns the declassification d_a and Bob learns d_b. There is also an empty declassification event ϵ, which is used for non-declassification statements. Given a declassification event $D = (d_a, d_b)$, we write $D[A]$ to denote Alice's declassification d_a and $D[B]$ to denote Bob's declassification d_b.

Semantics rules. Now we turn to the semantics, which consists of two judgments. Figure 4.6 defines rules for the judgment $l \vdash \langle M, e\rangle \Downarrow_{(t_a,t_b)} v$, which states that in mode l, under memory M, expression e evaluates to v. This evaluation produces memory trace t_a (resp., t_b) for Alice (resp., Bob). Which memory trace event to emit is chosen using the function *select*, which is defined in Figure 4.5. The security label l is passed in by the corresponding assignment statement (i.e. $l : x := e$ or $l : y[x_1] := x_2$). If l is `A` or `B`, then the accesses to public variables are not observable to the other party, whereas if l is `O` then both parties know that an access took place; the $l^\star$ label defined in E-Var and E-Array ensures the proper visibility of such

$$\boxed{l \vdash \langle M, e\rangle \Downarrow_{(t_a,t_b)} v}$$

$$\text{E-Const}\quad l \vdash \langle M, n\rangle \Downarrow_{(\epsilon,\epsilon)} n$$

$$\text{E-Var}\frac{\begin{array}{c} M(x) = (v, l') \quad v \in \mathbf{Nat} \quad l' \sqsubseteq l \\ l = \mathsf{0} \Rightarrow l^\star = l' \quad l \neq \mathsf{0} \Rightarrow l^\star = l \\ (t_a, t_b) = select(l^\star, \mathbf{read}(x, v), x) \end{array}}{l \vdash \langle M, x\rangle \Downarrow_{(t_a,t_b)} v}$$

$$\text{E-Array}\frac{\begin{array}{c} M(x) = (m, l') \quad m \in \mathbf{Array} \quad l' \sqsubseteq l \\ l = \mathsf{0} \Rightarrow l^\star = l' \quad l \neq \mathsf{0} \Rightarrow l^\star = l \\ l \vdash \langle M, y\rangle \Downarrow_{(t'_a,t'_b)} v \quad v' = get(m, v) \\ (t''_a, t''_b) = select(l^\star, \mathbf{readarr}(x, v, v'), x) \\ t_a = t'_a @ t''_a \quad t_b = t'_b @ t''_b \end{array}}{l \vdash \langle M, x[y]\rangle \Downarrow_{(t_a,t_b)} v'}$$

$$\text{E-Op}\frac{\begin{array}{c} l \vdash \langle M, x_i\rangle \Downarrow_{(t_{ia},t_{ib})} v_i \quad i = 1, 2 \\ v = v_1 \; op \; v_2 \\ t_a = t_{1a} @ t_{2a} \quad t_b = t_{1b} @ t_{2b} \end{array}}{l \vdash \langle M, x_1 \; op \; x_2\rangle \Downarrow_{(t_a,t_b)} v}$$

$$\text{E-Mux}\frac{\begin{array}{c} l \vdash \langle M, x_i\rangle \Downarrow_{(t_{ia},t_{ib})} v_i \quad i = 1, 2, 3 \\ v_1 = 0 \Rightarrow v = v_2 \quad v_1 \neq 0 \Rightarrow v = v_3 \\ t_a = t_{1a} @ t_{2a} @ t_{3a} \quad t_b = t_{1b} @ t_{2b} @ t_{3b} \end{array}}{l \vdash \langle M, \mathbf{mux}(x_1, x_2, x_3)\rangle \Downarrow_{(t_a,t_b)} v}$$

Figure 4.6: Operational semantics for expressions in SCVM

events. Note the E-Array rule uses the $get()$ function to retrieve an element from an array; this function will return a default value 0 if the index is out of bounds. Most elements of the rules are otherwise straightforward.

Figure 4.7 and 4.8 define rules for the judgment $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : D$, which says that under memory M, the statement S reduces to memory M' and statement S', while producing instruction trace i_a (resp., i_b) and memory trace t_a (resp., t_b) for Alice (resp., Bob), and generating declassification D. Most rules are standard, except for handling memory traces and instruction traces. Instruction traces are handled using function *inst* defined in Figure 4.5. This function is defined

$$\boxed{\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : D}$$

$$\text{S-Skip}\ \langle M, l : \mathbf{skip}; S\rangle \xrightarrow{(\epsilon,\epsilon,\epsilon,\epsilon)} \langle M, S\rangle : \epsilon$$

$$\text{S-Assign}\ \frac{\begin{array}{c} l \vdash \langle M, e\rangle \Downarrow_{(t'_a,t'_b)} v \\ M' = M[x \mapsto (v, l)] \quad (i_a, i_b) = inst(l, x := e) \\ (t''_a, t''_b) = select(l, \mathbf{write}(x, v), x) \\ t_a = t'_a @ t''_a \qquad\qquad t_b = t'_b @ t''_b \end{array}}{\langle M, l : x := e\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', l : \mathbf{skip}\rangle : \epsilon}$$

$$\text{S-Declass}\ \frac{\begin{array}{c} M(y) = (v, \mathtt{O}) \qquad l \neq \mathtt{O} \qquad t_a = t_b = y \\ M' = M[x \mapsto (v, l)] \quad i = \mathtt{O} : \mathbf{declass}(x, y) \\ D = select(l, (x, v), \epsilon) \end{array}}{\langle M, \mathtt{O} : x := \mathbf{declass}_l(y)\rangle \xrightarrow{(i,t_a,i,t_b)} \langle M', \mathtt{O} : \mathbf{skip}\rangle : D}$$

$$\text{S-Cond}\ \frac{\begin{array}{c} (i_a, i_b) = inst(l, \mathbf{if}(x)) \quad M(x) = (v, l) \\ (t_a, t_b) = select(l, \mathbf{read}(x, v), x) \\ v = 1 \Rightarrow c = 1 \qquad v \neq 1 \Rightarrow c = 2 \end{array}}{\langle M, l : \mathbf{if}(x)\mathbf{then}\ S_1\mathbf{else}\ S_2\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M, S_c\rangle : \epsilon}$$

$$\text{S-ORAM}\ \frac{\begin{array}{c} M(y) = (m, l) \qquad\qquad l \neq \mathtt{O} \\ M' = M[x \mapsto (m, \mathtt{O})] \quad (t'_a, t'_b) = select(l, arr(y, m), \epsilon) \\ i = \mathtt{O} : \mathbf{init}(x, y) \qquad t_a = t'_a @ x \qquad t_b = t'_b @ x \end{array}}{\langle M, \mathtt{O} : x := \mathbf{oram}(y)\rangle \xrightarrow{(i,t_a,i,tb)} \langle M', \mathtt{O} : \mathbf{skip}\rangle : \epsilon}$$

Figure 4.7: Operational semantics for statements in SCVM (Part 1)

such that if the label l of a statement is A or B, then the other party cannot observe the statement; otherwise, both parties observe the statement.

A skip statement generates empty instruction traces and memory traces for both parties regardless of its label. An assignment statement first evaluates the expression to assign, and its trace and the write event constitute the memory trace for this statement. Note that expression is evaluated using the label l of the assignment statement as per the discussion of E-Var and E-Array above.

Declassification $x := \mathbf{declass}_l(y)$ declassifies a secret variable y (labeled O) to

$$\text{S-ArrAss}\ \frac{\begin{array}{c} M(y) = (m, l) \quad l \vdash \langle M, x_i\rangle \Downarrow_{(t_{ia}, t_{ib})} v_i \quad i = 1, 2 \\ m' = set(m, v_1, v_2) \qquad M' = M[y \mapsto (m', l)] \\ (t'_a, t'_b) = select(l, \mathbf{writearr}(y, v_1, v_2), y) \\ t_a = t_{1a} @ t_{2a} @ t'_a \qquad t_b = t_{1b} @ t_{2b} @ t'_b \\ (i_a, i_b) = inst(l, y[x_1] := x_2) \end{array}}{\langle M, l : y[x_1] := x_2\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M', l : \mathbf{skip}\rangle : \epsilon}$$

$$\text{S-While-False}\ \frac{\begin{array}{c} M(x) = (0, l) \quad (i_a, i_b) = inst(l, \mathbf{while}(x)) \\ (t_a, t_b) = select(l, \mathbf{read}(x, 0), x) \\ S = l : \mathbf{while}(x) \mathbf{do}\ S' \end{array}}{\langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M, l : \mathbf{skip}\rangle : \epsilon}$$

$$\text{S-While-True}\ \frac{\begin{array}{c} M(x) = (v, l) \qquad v \neq 0 \\ (t_a, t_b) = select(l, \mathbf{read}(x, v), x) \\ S = l : \mathbf{while}(x) \mathbf{do}\ S' \end{array}}{\langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M, S'; S\rangle : \epsilon}$$

$$\text{S-Seq}\ \frac{\langle M, S_1\rangle \xrightarrow{(i_a, t_a, i_b, i_b)} \langle M', S'_1\rangle : D}{\langle M, S_1; S_2\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M', S'_1; S_2\rangle : D}$$

$$\text{S-Concat}\ \frac{\begin{array}{c} \langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, i_b)} \langle M', S'\rangle : \epsilon \\ \langle M', S'\rangle \xrightarrow{(i'_a, t'_a, i'_b, i'_b)} \langle M'', S''\rangle : D \end{array}}{\langle M, S\rangle \xrightarrow{(i_a @ i'_a, t_a @ t'_a, i_b @ i'_b, t_b @ t'_b)} \langle M'', S'\rangle : D}$$

Figure 4.8: Operational semantics for statements in SCVM (Part 2)

a non-secret variable `x` (not labeled `0`). Both Alice and Bob will observe that y is accessed (as defined by t_a and t_b), whereas the label l of variable x determines who sees the declassified value as indicated by the declassification event D.

ORAM initialization produces a shared, secret array x from an array y provided by one party. Thus, the security label of x must be `0`, and the security label of y must not be `0`. This rule implies that the party who holds y will observe memory events $arr(y, m)$, and then both parties can observe accesses to x.

Rule S-ArrAss handles an array assignment. Similar to rule E-Array, out-of-bounds indices are ignored (cf. the $set()$ function in Figure 4.5). For if-statements and while-statements, no memory traces are observed other than those observed from evaluating the guard x.

Rule S-Seq sequences execution of two statements in the obvious way. Finally, rule S-Concat says that if $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M'', S''\rangle : D$, the transformation may perform one or more small-step transformations that generate no declassification.

4.4.3 Security

The ideal functionality $\mathcal{F}$ defines the baseline of security, emulating a trusted third party that runs the program using Alice and Bob's data, directly revealing to them only the explicitly declassified values. In a real implementation run directly by Alice and Bob, however, each party will see additional events of interest, in particular an instruction trace and a memory trace (as defined by the semantics). Importantly, we want to show that these traces provide no additional information

about the opposite party's data beyond what each party could learn from observing $\mathcal{F}$. We do this by proving that in fact these traces can be *simulated* by Alice and Bob using their local data and the list of declassification events provided by $\mathcal{F}$. As such, revealing the instruction and memory traces (as in a real implementation) provides no additional useful information.

We call our security property Γ-*simulatability*. To state this property formally, we first define a multi-step version of our statement semantics:

$$\frac{\begin{array}{c}\langle M, P\rangle \xrightarrow{\Gamma,(i_a,t_a,i_b,t_b)}{}^{\star} \langle M_n, P_n\rangle : D_1, ..., D_n \qquad n \geq 0 \\ \langle M_n, P_n\rangle \xrightarrow{(i'_a,t'_a,i'_b,t'_b)} \langle M', P'\rangle : D' \\ D' \neq \epsilon \vee P' = l : \textbf{skip} \qquad M \text{ and } M' \text{ are both } \Gamma\text{-compatible}\end{array}}{\langle M, P\rangle \xrightarrow{\Gamma,(i'_a,t'_a,i'_b,t'_b)}{}^{\star} \langle M', P'\rangle : D_1, ..., D_n, D'}$$

This allows programs to make multiple declassifications, accumulating them as a trace, while remembering only the most recent instruction and memory traces and ensuring that intermediate memories are Γ-compatible.

Definition 8 (Γ-simulatability)**.** *Let Γ be a type environment, and P a program. We say P is* Γ-simulatable *if there exist simulators sim_A and sim_B, which run polynomial time in the data size, such that for all $M, i_a, t_a, i_b, t_b, M', P', D_1, ..., D_n$, if $\langle M, P\rangle \xrightarrow{\Gamma,(i_a,t_a,i_b,t_b)}{}^{\star} \langle M', P'\rangle : D_1, ..., D_n$, then $sim_A(M[\{\texttt{P},\texttt{A}\}], D_1[A], ..., D_{n-1}[A]) \equiv (i_a, t_a)$ and $sim_B(M[\{\texttt{P},\texttt{B}\}], D_1[B], ..., D_{n-1}[B]) \equiv (i_b, t_b)$.*

Intuitively, if P is Γ-simulatable there exists a simulator sim_A that, given public data $M[\{\texttt{P}\}]$, Alice's secret data $M[\{\texttt{A}\}]$, and all outputs $D_1[A], ..., D_{n-1}[A]$ declassified to Alice so far, can compute the instruction traces i_a and memory traces

t_a produced by the ideal semantics up until the next declassification event D_n, regardless of the values of Bob's secret data.

Note that Γ-simulatability is *termination insensitive*, and information may be leaked based upon whether a program terminates or not [9]. However, as long as all runs of a program are guaranteed to terminate (as is typical for programs run in secure-computation scenarios), no information leakage occurs.

4.4.4 Type System

This section presents our type system, which we prove ensures Γ-simulatability. There are two judgments, both defined in Figure 4.9. The first, written $\Gamma \vdash e : \tau$, states that under environment Γ, expression e evaluates to type τ. The second judgment, written $\Gamma, pc \vdash S$, states that under environment Γ and a *label context* pc, a labeled statement S is type-correct. Here, pc is a label that describes the ambient control context; pc is set according to the guards of enclosing conditionals or loops. Note that since a program cannot execute an if-statement or a while-statement whose guard is secret, pc can be one of `P`, `A`, or `B`, but not `O`. Intuitively, if pc is `A` (resp., `B`), then the statement is part of Alice's (resp., Bob's) local code. In general, for a labeled statement $S = l : s$ we enforce the invariant $pc \sqsubseteq l$, and if $pc \neq \texttt{P}$, then $pc = l$. In so doing, we ensure that if the security label of a statement is `A` (including if-statements and while-statements), then all nested statements also have security label `A`, thus ensuring they are only visible to Alice. On the other hand, under a public context, the statement label is unrestricted.

$$\boxed{\Gamma \vdash e : \tau} \qquad \text{T-Var}\frac{\Gamma(x) = \mathbf{Nat}\ l}{\Gamma \vdash x : \mathbf{Nat}\ l} \qquad \text{T-Const}\frac{}{\Gamma \vdash n : \mathbf{Nat}\ \mathtt{P}}$$

$$\text{T-Op}\frac{\Gamma(x_1) = \mathbf{Nat}\ l_1 \quad \Gamma(x_2) = \mathbf{Nat}\ l_2}{\Gamma \vdash x_1\ op\ x_2 : \mathbf{Nat}\ l_1 \sqcup l_2}$$

$$\text{T-Array}\frac{\Gamma(y) = \mathbf{Array}\ l_1 \quad \Gamma(x) = \mathbf{Nat}\ l_2 \quad l_2 \sqsubseteq l_1}{\Gamma \vdash y[x] : \mathbf{Nat}\ l_1} \qquad \text{T-Mux}\frac{\Gamma(x_i) = \mathbf{Nat}\ l_i \quad i = 1,2,3 \quad l = l_1 \sqcup l_2 \sqcup l_3}{\Gamma \vdash \mathbf{mux}(x_1, x_2, x_3) : \mathbf{Nat}\ l}$$

$$\boxed{\Gamma, pc \vdash S} \qquad \text{T-Skip}\frac{pc \sqsubseteq l \quad pc \neq \mathtt{P} \Rightarrow pc = l}{\Gamma, pc \vdash l : \mathbf{skip}} \qquad \text{T-Seq}\frac{\Gamma, pc \vdash S_1 \quad \Gamma, pc \vdash S_2}{\Gamma, pc \vdash S_1; S_2}$$

$$\text{T-Assign}\frac{\Gamma(x) = \mathbf{Nat}\ l \quad \Gamma \vdash e : \mathbf{Nat}\ l' \quad pc \sqcup l' \sqsubseteq l \quad pc \neq \mathtt{P} \Rightarrow l = pc}{\Gamma, pc \vdash l : x := e} \qquad \text{T-Declass}\frac{pc = \mathtt{P} \quad \Gamma(y) = \mathbf{Nat}\ \mathtt{0} \quad \Gamma(x) = \mathbf{Nat}\ l \quad l \neq \mathtt{0}}{\Gamma, pc \vdash \mathtt{0} : x := \mathbf{declass}_l(y)}$$

$$\text{T-ORAM}\frac{pc = \mathtt{P} \quad \Gamma(x) = \mathbf{Array}\ \mathtt{0} \quad \Gamma(y) = \mathbf{Array}\ l \quad l \neq \mathtt{0}}{\Gamma, pc \vdash \mathtt{0} : x := \mathbf{oram}(y)} \qquad \text{T-ArrAss}\frac{\Gamma(y) = \mathbf{Array}\ l \quad \Gamma(x_1) = \mathbf{Nat}\ l_1 \quad \Gamma(x_2) = \mathbf{Nat}\ l_2 \quad pc \sqcup l_1 \sqcup l_2 \sqsubseteq l \quad pc \neq \mathtt{P} \Rightarrow l = pc}{\Gamma, pc \vdash l : y[x_1] := x_2}$$

$$\text{T-Cond}\frac{\Gamma(x) = \mathbf{Nat}\ l \quad pc \sqsubseteq l \quad l \neq \mathtt{0} \quad pc \neq \mathtt{P} \Rightarrow l = pc \quad \Gamma, l \vdash S_i \quad i = 1,2}{\Gamma, pc \vdash l : \mathbf{if}(x)\mathbf{then}\ S_1\mathbf{else}\ S_2} \qquad \text{T-While}\frac{\Gamma(x) = \mathbf{Nat}\ l \quad pc \sqsubseteq l \quad l \neq \mathtt{0} \quad \Gamma, l \vdash S \quad pc \neq \mathtt{P} \Rightarrow l = pc}{\Gamma, pc \vdash l : \mathbf{while}(x)\mathbf{do}\ S}$$

Figure 4.9: Type System for SCVM

Now we consider some interesting aspects of the rules. Rule T-Assign requires $pc \sqcup l' \sqsubseteq l$, as is standard: $pc \sqsubseteq l$ prevents implicit flows, and $l' \sqsubseteq l$ prevents explicit ones. We further restrict that $\Gamma(x) = \mathbf{Nat}\ l$, i.e., the assigned variable should have the same security label as the instruction label. Rule T-ArrAss and rule T-Array require that for an array expression $y[x]$, the security label of x should be lower than the security label of y. For example, if x is Alice's secret variable, then y should be either Alice's local array, or an ORAM shared between Alice and Bob. If y is

Bob's secret variable, or a public variable, then Bob can observe which indices are accessed, and then infer the value of x. In the example from Figure 4.2, the array access `vis[bestj]` on line 9 requires that `vis` be an ORAM variable since `bestj` is.

For rules T-Declass and T-ORAM, since declassification and ORAM initialization statements both require secure computation, we restrict the statement label to be `0`. Since these two statements cannot be executed in Alice's or Bob's local mode, we restrict that $pc = \texttt{P}$.

Rule T-Cond deals with if-statements; T-While handles while loops similarly. First of all, we restrict $pc \sqsubseteq l$ and $\Gamma(x) = \textbf{Nat}\ l$ for the same reason as above. Further, the rule forbids l to be equal to `0` to avoid an implicit flow revealed by the program's control flow. An alternative way to achieve instruction- and memory- trace obliviousness is through padding [58]. However, in the setting of secure-computation, padding achieves the same performance as rewriting a secret-branching statement into a **mux** (or a sequence of them). And, using padding would require reasoning about trace patterns, a complication our type system avoids.

A well-typed program is Γ-simulatable:

Theorem 3. *If* $\Gamma, \texttt{P} \vdash S$*, then* S *is* Γ*-simulatable.*

Notice that some rules allow a program to get stuck. For example, in rule S-ORAM, if the statement is $l : x := \textbf{oram}(y)$ but $l \neq \texttt{0}$, then the program will not progress. We define a property called Γ*-progress* that formalizes the notion of a program that never gets stuck.

Definition 9 (Γ-progress)**.** *Let* Γ *be a type environment, and let* $P = P_0$ *be a*

program. We say P enjoys Γ-progress *if for any Γ-compatible memories $M_0, \ldots, M_n$ for which $\langle M_j, P_j\rangle \xrightarrow{(i_a^j, t_a^j, i_b^j, t_b^j)} \langle M_{j+1}, P_{j+1}\rangle : D^j$ for $j = 0, ..., n-1$, either $P_n = l :$ **skip**, or there exist $i'_a, t'_a, i'_b, t'_b, M', P'$ such that $\langle M_n, P_n\rangle \xrightarrow{(i'_a, t'_a, i'_b, t'_b)} \langle M', P'\rangle : D'$.*

Γ-progress means, in particular, that the third bullet in step (2) of the ideal functionality (Section 4.4.2) does not occur for type-correct programs.

A well-typed program never gets stuck:

Theorem 4. *If $\Gamma, \mathtt{P} \vdash S$, then S enjoys Γ-progress.*

Proofs of both theorems above can be found in Appendix C.

4.4.5 From SCVM Programs to Secure Protocols

Let P be a program, and let $\mathcal{F}$ be the ideal functionality based on this program as described earlier. Here we define a *hybrid-world* protocol $\pi^{\mathcal{G}}$ based on P, where $\mathcal{G} = (\mathcal{F}_{\text{op}}, \mathcal{F}_{\text{mux}}, \mathcal{F}_{\text{oram}}, \mathcal{F}_{\text{declass}})$ is a fixed set of ideal functionalities that implement simple binary operations ($\mathcal{F}_{\text{op}}$), a MUX operation ($\mathcal{F}_{\text{mux}}$), ORAM access ($\mathcal{F}_{\text{oram}}$), and declassification ($\mathcal{F}_{\text{declass}}$). Input to each of these ideal functionalities can either be Alice or Bob's local inputs, public inputs, and/or the shares of secret inputs (each share supplied by Alice and Bob respectively). Each ideal functionality is explicitly parameterized by the types of the inputs. Further, except for $\mathcal{F}_{\text{declass}}$ which performs an explicit declassification, all other ideal functionalities return shares of the computation or memory fetch result to Alice and Bob, respectively. Further details of the ideal functionalities are given in Appendix D, along with formal definitions of the simulator and hybrid world semantics.

Informally, the hybrid world protocol $\pi^{\mathcal{G}}$ runs as follows:

1. Alice and Bob first agree on public values, ensuring that $M_A[\{\mathtt{P}\}] = M_B[\{\mathtt{P}\}]$. During the protocol each maintains a *declassification list*, for keeping track of previously declassified values, and a *secret memory* that contains shares of secret (non-ORAM) variables. To start, both the lists and memories are empty, i.e., $\mathtt{decls}_A := \mathtt{decls}_B := \epsilon$ and $M_A^S = M_B^S = []$.

2. Alice runs her simulator (locally) on her initial memory to obtain $(i_a, t_a) = sim_A(M_A, \mathtt{decls}_A)$, where i_a and t_a cover the portion of the execution starting from just after the last provided declassification (i.e., the final d_a in the list $\mathtt{decls}_A$) up to the next declassification instruction or the terminating **skip** statement. Bob does likewise to get $(i_b, t_b) = sim_B(M_B, \mathtt{decls}_B)$.

3. Alice executes the instructions in i_a using the hybrid-world semantics, which reads (and writes) secret shares from (to) M_A^S and obtains the values of other reads from events observed in t_a. Bob does similarly with i_b, M_B^S and t_b. The semantics covers three cases:

   - If an instruction in i_a is labeled `P`, then so is the corresponding instruction in i_b. Both parties execute the instruction.
   - If an instruction in i_a is labeled `A`, then Alice executes this instruction locally. Bob does similarly for instructions labeled `B`.
   - If an instruction in i_a is labeled `O`, then so is the corresponding instruction in i_b. Alice and Bob call the appropriate ideal-world functionality from

$\mathcal{G}$ to execute this instruction. If the instruction is a declassification, then $\mathcal{F}_{\text{declass}}$ will generate an event (d_a, d_b).

4. If the last instruction executed in step 3 is a declassification, then Alice appends her declassification to her local declassification list (i.e., $\texttt{decls}_A :=$ $\texttt{decls}_A\texttt{++}[d_a]$), and Bob does likewise; then both repeat step 2. Otherwise, the protocol completes.

We have proved that if P is Γ-simulatable, then $\pi^{\mathcal{G}}$ securely implements $\mathcal{F}$ against semi-honest adversaries.

Theorem 5. *(Informally) Let P be a program, $\mathcal{F}$ the ideal functionality corresponding to P, and $\pi^{\mathcal{G}}$ the protocol corresponding to P as described above. If P is Γ-simulatable, then $\pi^{\mathcal{G}}$ securely implements $\mathcal{F}$ against semi-honest adversaries in the $\mathcal{G}$-hybrid model.*

Using standard composition results for cryptographic protocols, we obtain as a corollary that if all ideal functionalities in $\mathcal{G}$ are implemented by semi-honest secure protocols, the resulting (real-world) protocol securely implements $\mathcal{F}$ against semi-honest adversaries.

A formal definition of $\pi^{\mathcal{G}}$, formal theorem statement, and a proof of the theorem can be found in Appendix D.

4.5 Compilation

We informally discuss how to compile an annotated C-like source language into a SCVM program. An example of our source language is:

```
int sum(alice int x, bob int y) {
  return x<y ? 1 : 0;
}
```

The program's two input variables, x and y, are annotated as Alice's and Bob's data, respectively, while the unannotated return type `int` indicates the result will be known to both Alice and Bob. Programmers need not annotate any local variables. To compile such a program into a SCVM program, the compiler takes the following steps.

Typing the source language. As just mentioned, source level types and initial security label annotations are assumed given. With these, the type checker infers security labels for local variables using a standard security type system [79] using our lattice (Section 4.4.4). If no such labeling is possible without violating security (e.g., due to a conflict in the initial annotation), the program is rejected.

Labeling statements. The second task is to assign a security label to each statement. For assignment statements and array assignment statements, the label is the least upper bound of all security labels of the variables occurring in the statement. For an if-statement or a while-statement, the label is the least upper bound of all security labels of the guard variables, and all security labels in the branches or loop body.

On secret branching. The type system defined in Section 4.4.4 will reject an if-statement whose guard has security label `0`. As such, if the program branches on secret data, we must compile it into *if-free* SCVM code, using **mux** instructions. The idea is to execute both branches, and use **mux** to activate the relevant effects,

based on the guard. To do this, we convert the code into Static-Single-Assignment form (SSA) [7], and then replace occurrences of the ϕ-operator with a **mux**. The following example demonstrates this process:

```
if(s) then x:=1; else x:=2;
```

The SSA form of the above code is

```
if(s) then x1:=1; else x2:=2; x:=phi(x1, x2);
```

Then we eliminate the if-structure and substitute the ϕ-operator to achieve the final code:

```
x1:=1; x2:=2; x:=mux(s, x1, x2)
```

(Note that, for simplicity, we have omitted the security labels on the statements in the example.)

On secret while loops. The type system requires that while loop guards only reference public data, so that the number of iterations does not leak information. A programmer can work around this restriction by imposing a constant bound on the loop; e.g., manually translating `while (s) do` S to `while (p) do if (s)` S `else skip`, where `p` defines an upper bound on the number of iterations.

Declassification. The compiler will emit a declassification statement for each return statement in the source program. To avoid declassifying in the middle of local code, the type checker in the first phase will check for this possibility and relabel statements accordingly.

Extension for non-oblivious secret RAM. The discussion so far supports only secret ORAMs. To support non-oblivious secret RAM in SCVM, we add an ad-

ditional security label `N` such that `P` $\sqsubseteq$ `N` $\sqsubseteq$ `O`. To incorporate such a change, the memory trace for the semantics should include two more kinds of trace event, **nread**(x, i) and **nwrite**(x, i), which represent that only the index of an access is leaked, but not the content. Since label `N` only applies to arrays, we allow types **Array** `N` but not types **Nat** `N`. The rules T-Array and T-ArrAss should be revised to deal with the non-oblivious RAM. For example, for rule T-ArrAss, where l is the security label for the array, l_1 is the security label of the index variable and l_2 is the security label of the value variable, the type system should still restrict $l_1 \sqsubseteq l$, but if $l =$ `N`, the type system accepts $l_2 =$ `O`, but requires $l_1 =$ `P`.

Correctness. We do not prove the correctness of our compiler, but instead can use a SCVM type checker (using the above extension) for the generated SCVM code, ensuring it is Γ-simulatable. Ensuring the correctness of compilers is orthogonal and outside the scope of this work, and existing techniques [20] can potentially be adapted to our setting.

Compiling Dijkstra's algorithm. We explain how compilation works for Dijkstra's algorithm, previously shown in Figure 4.2. First, the type checker for the source program determines how memory should be labeled. It determines that the security labels for `bestj` and `bestdis` should be `O`, and the arrays `dis` and `vis` should be secret-shared between Alice and Bob, since their values depend on both Alice's input (i.e., the graph's edge weights) and Bob's input (i.e., the source). Then, since on line 9 array `vis` is indexed with `bestj`, variable `vis` should also be put in an ORAM. Similarly, on line 12, array `e` is indexed by `bestj` so it must also be secret; as such we must promote `e`, owned by Alice, to be in ORAM, which we do by

initializing a new ORAM-allocated variable `orame` to `e` at the start of the program.

The type checker then uses the variable labeling to determine the statement labeling. Statements on lines 4–7, 9, and 11–13, require secure computation and thus are labeled as `O`. Loop control-flow statements are computed publicly, so they are labeled as `P`.

The two if-statements both branch on ORAM-allocated data, so they must be converted to **mux** operations. Lines 4–7 are transformed (in source-level syntax) as follows

```
cond3 := !vis[j] && (bestj<0||dis[j]<bestdis);
bestj := mux(cond3, j, bestj);
bestdis := mux(cond3, dis[j], bestdis);
```

Lines 11-13 are similarly transformed

```
tmp := bestdis + orame[bestj*n+j];
cond4 := !vis[j] && (tmp<dis[j]);
dis[j] := mux(cond4, tmp, dis[j]);
```

Finally, the code is translated into SCVM's three-address code style syntax.

4.6 Evaluation

Programs. We have built several secure two-party computation applications. As run-once tasks, we implemented both the Knuth-Morris-Pratt (KMP) string-matching algorithm as well as Dijkstra's shortest-path algorithm. For repeated sublinear-time database queries, we considered binary search and the heap data

Name	Alice's Input	Bob's Input
Run-once setting		
Dijkstra's shortest path	a graph	a (src, dest) pair
Knuth-Morris-Pratt string matching	a sequence	a pattern
Aggregation over sliding windows	a key-value table	an array of keys
Inverse permutation	share of permutation	share of permutation
Repeated sublinear-time query		
Binary search	sorted array	search key
Heap (insertion/extraction)	share of the heap	share of the heap

Table 4.2: Programs used in evaluation of SCVM

structure. All applications are listed in Table 4.2.

Compilation time. All programs took little time (e.g., under 1 second) to compile. In comparison, some earlier circuit-model compilers involve copying datasets into circuits, and therefore the compile-time can be large [54,65] (e.g., Kreuter et al. [54] report a compilation time of roughly 1000 seconds for an implementation of an algorithm to compute graph isomorphism on 16-node graphs).

In our experiments, we manually checked the correctness of compiled programs (we have not yet implemented a type checker for SCVM, though doing so should be straightforward).

4.6.1 Evaluation Methodology

Although our techniques are compatible with any cryptographic back-end secure in the semi-honest model by the definition of Canetti [15], we use the garbled circuit approach in our evaluation [46].

We measure the computational cost by calculating the number of encryptions required by the party running as the circuit generator (the party running as the

evaluator does less work). For every non-XOR binary gate, the generator makes 3 block-cipher calls; for every oblivious transfer (OT), 2 block-cipher operations are required since we rely on OT extension [47]. For the *run-once* applications (i.e., Dijkstra shortest distance, KMP-matching, aggregation, inverse permutation), we count in the ORAM initialization cost when comparing to the automated circuit approach (which doesn't require RAM initialization). The ORAM initialization can be done using a Waksman shuffling network [89]. For the applications expecting multiple executions we do not count the ORAM initialization cost since this one-time overhead will be amortized to (nearly) 0 over many executions.

We implemented the binary tree-based ORAM of Shi et al. [80] using garbled circuits, so that array accesses reveal nothing about the (logical) addresses nor the outcomes. Throughout the experiments, we set the ORAM bucket size to 32 (i.e., each tree-node can store up to 32 blocks), which corresponds to roughly 25-bit of statistical security (according to the simulation of ORAM failures). Following Gordon et al.'s ORAM encryption technique [38], every block is XOR-shared (i.e., the client stores secret key k while the server stores $(r, f_k(r) \oplus m)$ where f is a family of psuedorandom permutations and m the data block). This adds one additional cipher operation per block (when the length of an ORAM block is less than the width of the cipher). We note specific choices of the ORAM parameters in related discussion of each application.

Metrics. We use the number of block-cipher evaluations as our performance metric. Measuring the performance by the number of symmetric encryptions (instead of wall clock times) makes it easier to compare with other systems since the numbers can

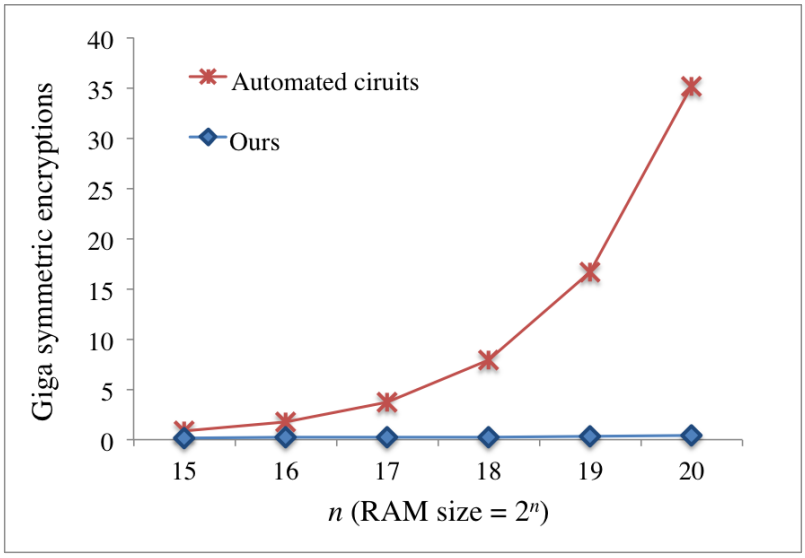

Figure 4.10: SCVM vs. automated circuit-based approach (Binary Search)

be independent of the underlying hardware and ciphering algorithms. Additionally, in our experiments these numbers represent bandwidth consumption since every encryption is sent over the network. Therefore, we do not report separately the bandwidth used. Modern processors with AES support can compute 10^8 AES-128 operations per second.

4.6.2 Comparison with Automated Circuits

Presently, automated secure computation implementations largely focus on the circuit-model of computation, handling array accesses by linearly scanning the entire array with a circuit every time an array lookup happens; this incurs prohibitive overhead when the dataset is large. In this section, we compare our approach with the existing compiled circuits, and demonstrate that our approach scales much better with respect to dataset size.

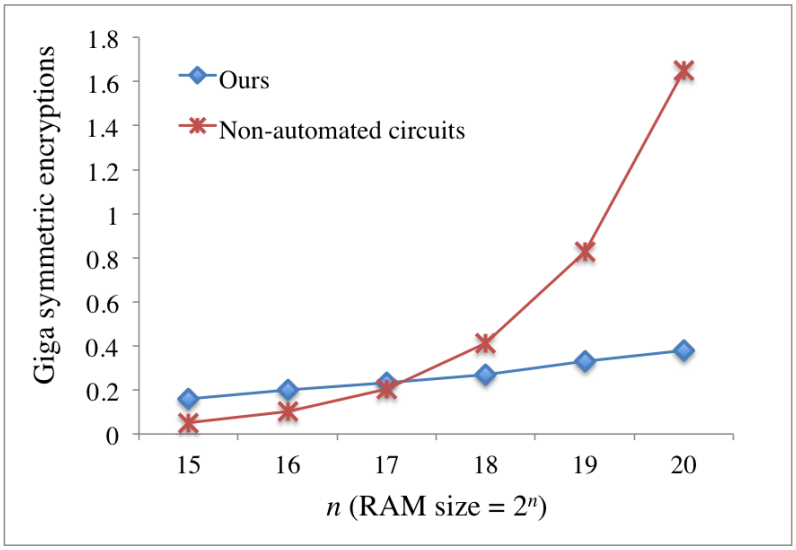

Figure 4.11: SCVM vs. hand-constructed linear scan circuit (Binary Search)

4.6.2.1 Repeated sublinear-time queries

In this scenario, ORAM initialization is a one-time operation whose cost can be amortized over multiple subsequent queries, achieving sublinear amortized cost per query.

Binary search. One example application we tested is binary search, where one party owns a confidential (sorted) array of size n, and the other party searches for (secret) values stored in that array.

In our experiments, we set the ORAM bucket size to 32. For binary search, we aligned our experimental settings with those of Gordon et al. [38], namely, assuming the size of each data item is 512 bits. We set the recursion factor to 8 (i.e., each block can store up to 8 indices for the data in the upper level recursion tree) and the recursion cut-off threshold to 1000 (namely no more recursion once fewer than 1000 units are to be stored). Comparing to a circuit-model implementation—which uses

a circuit of size $O(n \log n)$ that implements binary search—our approach is faster for all RAM sizes tested (see Figure 4.10). For $n = 2^{20}$, our approach achieves a 100× speedup.

Note it is also possible to use a smaller circuit of size $O(n)$ that just performs a linear scan over the data. However, such a circuit would have to be "hand-crafted," and would not be output by automated compilation of a binary-search program. Our approach runs faster for large n even when compared to such an implementation (see Figure 4.11). On data of size $n = 2^{20}$, our approach achieves a 5× speedup even when compared to this "hand-crafted" circuit-based solution.

Heap. Besides binary search, we also implemented an oblivious heap data structure (with 32-bit payload, i.e., size of each item). The costs of insertion and extraction respecting various heap sizes are given in Figure 4.12 and 4.13, respectively. The basic shapes of the performance curves are very similar to that for binary search (except that heap extraction is twice as slow as insertion because two comparisons are needed per level). We can observe an 18× speedup for both heap insertion and heap extraction when the heap size is 2^{20}.

The speedup of our heap implementation over automated circuits is even greater when the size of the payload is bigger. At 512-bit payload, we have an 100× speedup for data size 2^{20}. This is due to the extra work incurred from realizing the ORAM mechanism, which grows (in poly-logarithmic scale) with the size of the RAM but independent of the size of each data item.

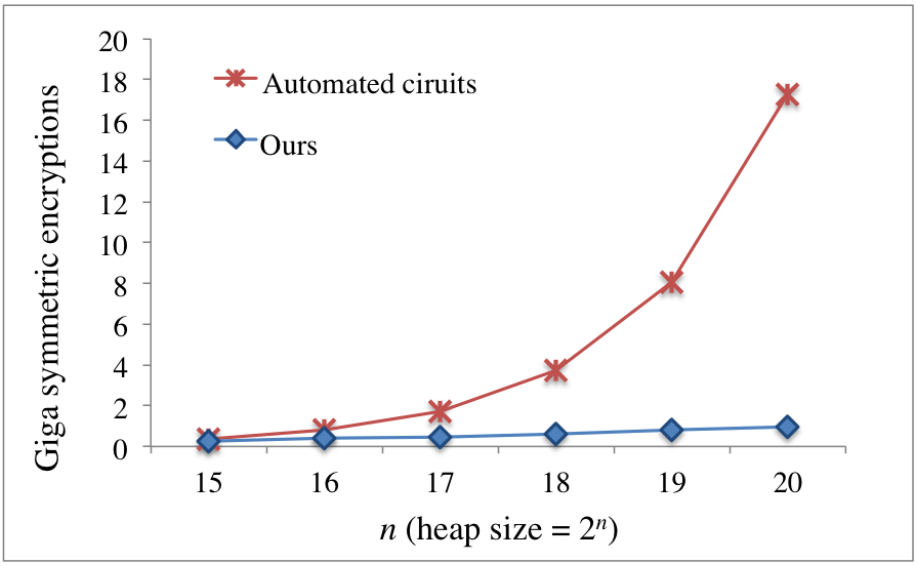

Figure 4.12: Heap insertion in SCVM

4.6.2.2 Faster one-time executions

We present two applications: the Knuth-Morris-Pratt string-matching algorithm (representative of linear-time RAM programs) and Dijkstra's shortest-path algorithm (representative of super-linear time RAM programs). We compare our approach with a naive program-to-circuit compiler which copies the entire array for every dynamic memory access.

The Knuth-Morris-Pratt algorithm. Alice has a secret string T (of length n) while Bob has a secret pattern P (of length m) and wants to scan through Alice's string looking for this pattern. The original KMP algorithm runs in $O(n + m)$ time when T and P are in plaintext. Our compiler compiles an implementation of KMP into a secure string matching protocol preserving its linear efficiency up to a polylogarithmic factor (due to the ORAM technique).

We assume the string T and the pattern P both consist of 16-bit characters.

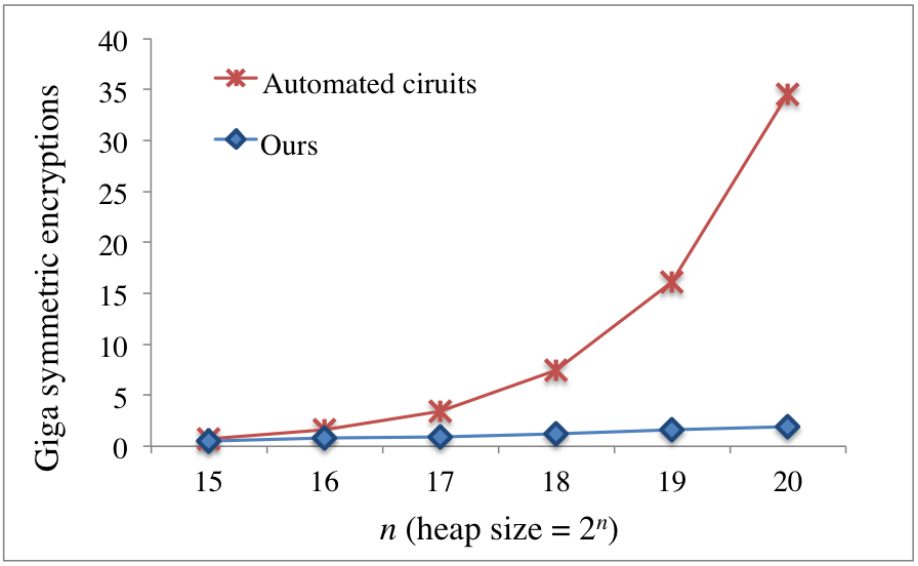

Figure 4.13: Heap extraction in SCVM

The recursion factor of the ORAM is set to 16. Figure 4.14 and 4.15 show our results compared to those when a circuit-model compiler is used. From Figure 4.14, we can observe that our approach is slower than the circuit-based approach on small datasets, since the overhead of the ORAM protocol dominates in such cases. However, the circuit-based approach's running time increases more quickly as the dataset's size increases. When $m = 50$ and $n = 2 \times 10^6$, our program runs 21× faster.

Dijkstra's algorithm. Here Alice has a secret graph while Bob has a secret source/destination pair and wishes to compute the shortest distance between them. Compiling from a standard Dijkstra shortest-path algorithm, we obtain an $O(n^2 \log^3 n)$-overhead RAM-model protocol.

In our experiment, Alice's graph is represented by an $n \times n$ adjacency matrix (of 32-bit integers) where n is the number of vertices in the graph. The distances associated with the edges are denoted by 32-bit integers. We set ORAM recursion

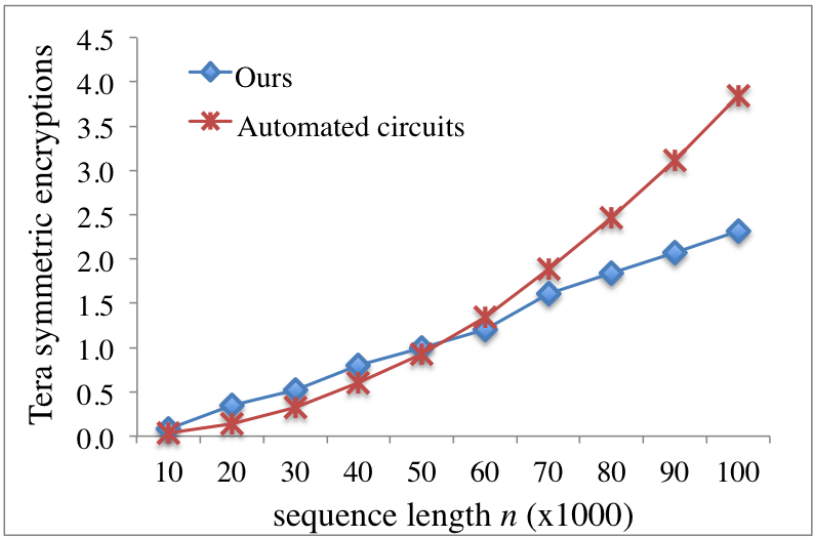

Figure 4.14: KMP string matching for median n (fixing $m = 50$) in SCVM

factor to 8. The results (Figure 4.16) show that our scheme runs faster for all sizes of graphs tested. As the performance of our protocol is barely noticeable in Figure 4.16, the performance gaps between the two protocols for various n is explicitly plotted in Figure 4.17. Note the shape of the speedup curve is roughly quadratic.

Aggregation over sliding windows. Alice has a key-value table, and Bob has a (size-n) array of keys. The secure computation task is the following: for every size-k window on the key array, look up k values corresponding to Bob's k keys within the window, and output the minimum value. Our compiler outputs a $O(n \log^3 n)$ protocol to accomplish the task. The optimized protocol performs significantly better, as shown in Figure 4.18 and 4.19 (we fixed the window size k to 1000 and set recursion factor to 8, while varying the dataset from 1 to 6 million pairs).

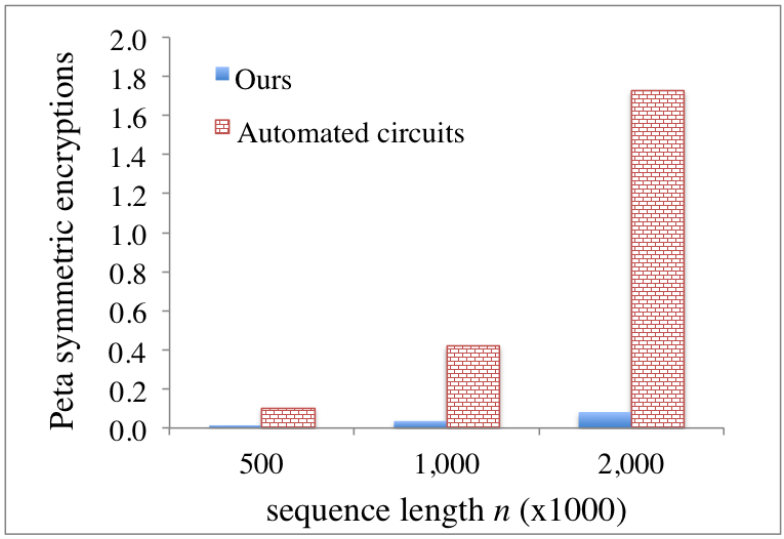

Figure 4.15: KMP string matching for large n (fixing $m = 50$) in SCVM

4.6.3 Comparison with RAM-SC Baselines

Benefits of instruction-trace obliviousness. The RAM-SC technique of Gordon et al. [38], described in Section 4.2, uses a universal next-instruction circuit to hide the program counter and the instructions executed. Each instruction involves ORAM operations for instruction and data fetches, and the next-instruction circuit must effectively execute all possible instructions and use an n-to-1 multiplexer to select the right outcome. Despite the lack of concrete implementation for their general approach, we show through back-of-the-envelope calculations that our approach should be orders-of-magnitude faster.

Consider the problem of binary search over a 1-million item dataset: in each iteration, there are roughly 10 instructions to run, hence 200 instructions in total to complete the search. To run every instruction, a universal-circuit-based implementation has to execute every possible instruction defined in its instruction set. Even

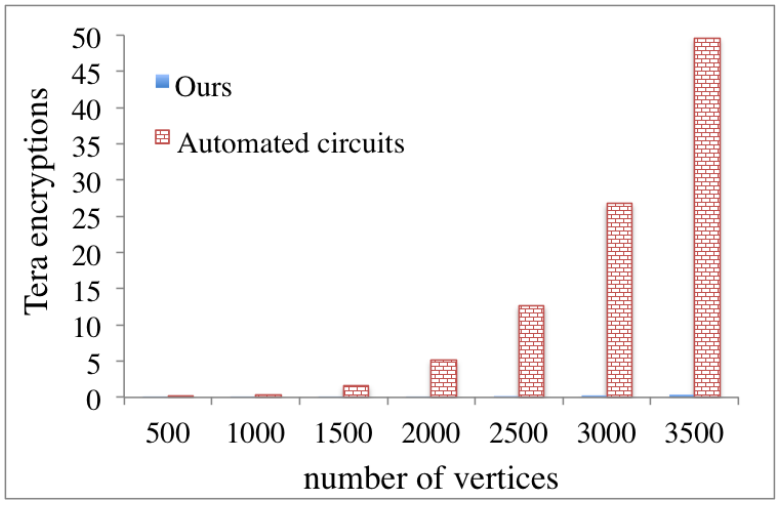

Figure 4.16: Dijkstra's shortest-path algorithm's performance in SCVM

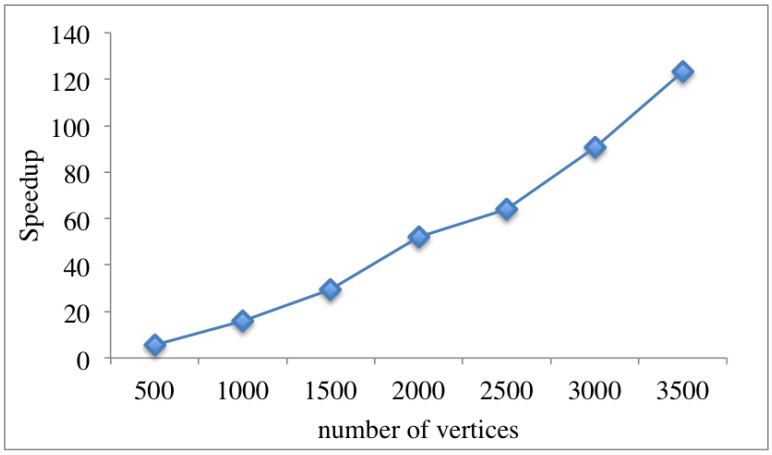

Figure 4.17: Dijkstra's shortest-path algorithm's speedup of SCVM

if we conservatively assume a RISC-style instruction set, we would require over 9 million (non-free) binary gates to execute just a memory read/write over a 512M bit RAM. Plus, an extra ORAM read is required to obliviously fetch every instruction. Thus, at least a total of 3600 million binary gates are needed, which is more than 20 times slower than our result exploiting instruction trace obliviousness. Furthermore, notice that binary search is merely a case where the program traces are very short (with only logarithmic length). Due to the overwhelming cost of ORAM read/write

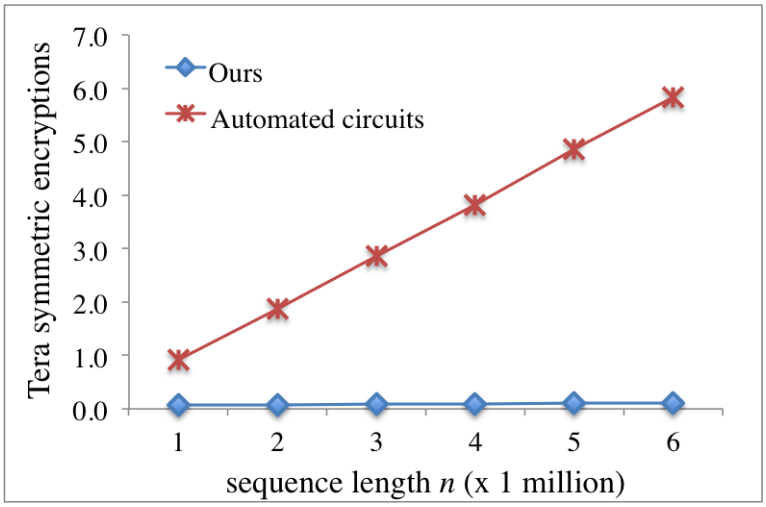

Figure 4.18: Aggregation over sliding windows's performance in SCVM

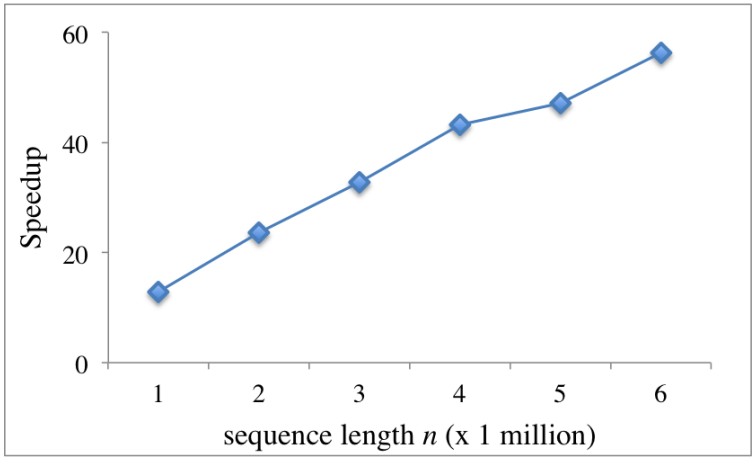

Figure 4.19: Aggregation over sliding windows's speedup in SCVM

instructions, we stress that the performance gap will be much greater with respect to programs that have relatively fewer memory read/write instructions (comparing to binary search, 1 out of 10 instructions is a memory read instruction).

Benefits of memory-trace obliviousness. In addition to avoiding the overhead of a next-instruction circuit, SCVM avoids the overhead of storing all arrays in a single, large ORAM. Instead, SCVM can store some arrays as non-oblivious secret shared memory, and others in separate ORAM banks, rather than one large

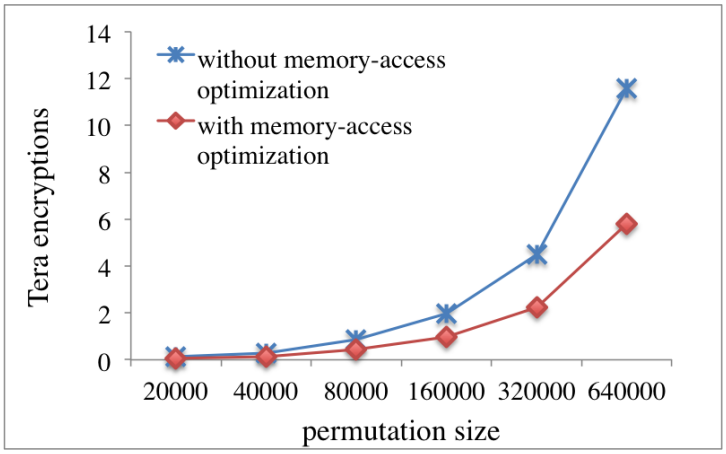

Figure 4.20: SCVM's Savings by memory-trace obliviousness optimization (inverse permutation). the non-linearity (around 60) of the curve is due to the increase of the ORAM recursion level at that point.

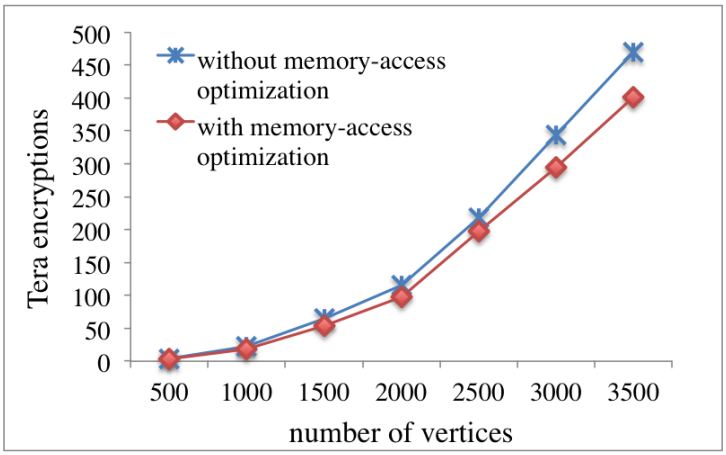

Figure 4.21: Savings by memory-trace obliviousness optimization (Dijkstra)

ORAM. Doing so does not compromise security because the type system ensures memory-trace obliviousness. Here we assess the advantages of these optimizations by comparing against SCVM programs compiled without the optimizations enabled. The results for two applications are given in Figure 4.20 and 4.21.

- **Inverse permutation.** Consider a permutation of size n, represented by an array `a` of n distinct numbers from 1 to n, i.e., the permutation maps the `i`-th object to the `a[i]`-th object. One common computation would be to compute

its inverse, e.g., to do an inverse table lookup using secret indices. The inverse permutation (with result stored in array `b`) can be computed with the loop:

```
while (i < n) { b[a[i]]=i; i=i+1;}
```

The memory-trace obliviousness optimization automatically identifies that the array `a` doesn't need to be put in ORAM though its content should remain secret (because the access pattern to `a` is entirely public known). This yields 50% savings, which is corroborated by our experiment results (Figure 4.20).

- **Dijkstra's shortest path.** We discussed the advantages of memory-trace obliviousness in Section 4.3 with respect to Dijkstra's algorithm. Our experiments show that we consistently save $15 \sim 20\%$ for all graph sizes. The savings rates for smaller graphs are in fact higher even though it is barely noticeable in the chart because of the fast (super-quadratic) growth of overall cost.

4.7 Conclusions

We describe the SCVM system as the first automated approach for RAM-model secure computation. In the next Chapter, we will extend SCVM to ObliVM with richer programming features to improve expressive power, easy programmability, and performance. Directions for future work include extending our framework to support malicious security; applying orthogonal techniques (e.g., [20]) to ensure correctness of the compiler; incorporating other cryptographic backends into our framework; and adding additional language features such as higher-dimensional arrays and structured data types.

Chapter 5: ObliVM: A Programming Framework for Secure Computation

In Chapter 4, we have seen that trace oblivious approach and SCVM system can improve the performance for RAM-model secure computation. In this Chapter, we extend SCVM to build a programming framework to make secure computation both easy-to-program and practically efficient.

Architecting a system framework for secure computation presents numerous challenges. First, the system must allow *non-specialist programmers* without security expertise to develop applications. Second, *efficiency* is a first-class concern in the design space, and scalability to big data is essential in many interesting real-life applications. Third, the framework must be reusable: *expert programmers* should be able to easily extend the system with rich, optimized libraries or customized cryptographic protocols, and make them available to non-specialist application developers.

We design and build ObliVM, a system framework for automated secure multiparty computation. ObliVM is designed to allow non-specialist programmers to write programs much as they do today, and our ObliVM compiler compiles the program to an efficient secure computation protocol. To this end, ObliVM offers a domain-specific language that is intended to address a fundamental representation gap,

namely, secure computation protocols (and other branches of modern cryptography) rely on *circuits* as an abstraction of computation, whereas real-life developers write *programs* instead. In architecting ObliVM, our main contribution is the design of programming support and compiler techniques that facilitate such program-to-circuit conversion while ensuring maximal efficiency. Presently, our framework assumes a semi-honest two-party protocol in the back end. To demonstrate an end-to-end system, we chose to implement an improved Garbled Circuit protocol as the back end, since it is among the most practical protocols to date. Our ObliVM framework, including source code and demo applications, will be open-sourced on our project website `http://www.oblivm.com`.

This chapter is based on a paper that I co-authored with Yan Huang, Kartik Nayak, Elaine Shi, and Xiao Shaun Wang [60]. I designed the novel programming language and implemented the programming frameworks. I developed the compiler to support the new language, which emits code that is runnable on a new secure computation backend designed and implemented by Yan Huang and Xiao Shaun Wang. I conducted experiments to show the compiler's effectiveness with the help of Xiao Shaun Wang.

5.1 ObliVM Overview and Contributions

In designing and building ObliVM, we make the following contributions.

Programming abstractions for oblivious algorithms. The most challenging part about ensuring a program's obliviousness is memory-trace obliviousness – there-

fore our discussions below will focus on memory-trace obliviousness. A straightforward approach (henceforth referred to as the generic ORAM baseline) is to provide an Oblivious RAM (ORAM) abstraction, and require that all arrays (whose access patterns depend on secret inputs) be stored and accessed via ORAM. This approach, which was effectively taken by SCVM [59], is generic, but does not necessarily yield the most efficient oblivious implementation for each specific program.

At the other end of the spectrum, a line of research has focused on customized oblivious algorithms for special tasks (sometimes also referred to as circuit structure design). For example, efficient oblivious algorithms have been demonstrated for graph algorithms [14, 37], machine learning algorithms [70, 71], and data structures [51, 66, 92]. The customized approach can outperform generic ORAM, but is extremely costly in terms of the amount of cryptographic expertise and time consumed.

ObliVM aims to achieve the best of both worlds by offering oblivious programming abstractions that are both user- and compiler friendly. These programming abstractions are high-level programming constructs that can be understood and employed by non-specialist programmers without security expertise. Behind the scenes, ObliVM translates programs written in these abstractions into efficient oblivious algorithms that outperform generic ORAM. When oblivious programming abstractions are not applicable, ObliVM falls back to employing ORAM to translate programs to efficient circuit representations. Presently, ObliVM offers the following oblivious programming abstractions: MapReduce abstractions, abstractions for oblivious data structures, and a new loop coalescing abstraction which enables novel

oblivious graph algorithms. We remark that this is by no means an exhaustive list of possible programming abstractions that facilitate obliviousness. It would be exciting future research to uncover new oblivious programming abstractions and incorporate them into our ObliVM framework.

An expressive programming language. ObliVM offers an expressive and versatile programming language called ObliVM-Lang. When designing ObliVM, we have the following goals.

- Non-specialist application developers find the language intuitive.
- Expert programmers should be able to extend our framework with new features. For example, an expert programmer should be able to introduce new, user-facing oblivious programming abstractions by embedding them as libraries in ObliVM (see Section 5.3.2 for an example).
- Expert programmers can implement even low-level circuit libraries directly atop ObliVM-Lang. Recall that unlike a programming language in the traditional sense, here the underlying cryptography fundamentally speaks only of AND and XOR gates. Even basic instructions such as addition, multiplication, and ORAM accesses must be developed from scratch by an expert programmer. In most previous frameworks, circuit libraries for these basic operations are developed in the back end. ObliVM, for the first time, allows the development of such circuit libraries in the source language, greatly reducing programming complexity. Section 5.4.1 demonstrates case studies for implementing basic arithmetic operations and Circuit ORAM atop our source

language ObliVM.

- Expert programmers can implement customized protocols in the back end (e.g., faster protocols for performing big integer operations or matrix operations), and export these customized protocols to the source language as native types and native functions.

To simultaneously realize these aforementioned goals, we need a much more powerful and expressive programming language than any existing language for secure computation [44, 54, 59, 75, 98]. Our ObliVM-Lang extends the SCVM language presented in Chapter 4 and offers new features such as phantom functions, generic constants, random types, as well as native types and functions. We will show why these language features are critical for implementing oblivious programming abstractions and low-level circuit libraries.

Additional architectural choices. ObliVM also allows expert programmers to develop customized cryptographic protocols (not necessarily based on Garbled Circuit) in the back end. These customized back end protocols can be exposed to the source language through native types and native function calls, making them immediately reusable by others. Section 5.5.1 describes an example where an expert programmer designs a customized protocol for `BigInteger` operations using additively-homomorphic encryption. The resulting `BigInteger` types and operations can then be exported into our source language ObliVM-Lang.

5.1.1 Applications and Evaluation

ObliVM's easy programmability allowed us to develop a suite of libraries and applications, including streaming algorithms, data structures, machine learning algorithms, and graph algorithms. These libraries and applications will be shipped with the ObliVM framework. Our application-driven evaluation suggests the following results:

Efficiency. We use ObliVM's user-facing programming abstractions to develop a suite of applications. We show that over a variety of benchmarking applications, the resulting circuits generated by ObliVM can be orders of magnitude smaller than the generic ORAM baseline (assuming that the state-of-the-art Circuit ORAM [90] is adopted for the baseline) under moderately large data sizes. We also compare our ObliVM-generated circuits with hand-crafted designs, and show that for a variety of applications, our auto-generated circuits are only **0.5%** to **2%** bigger in size than oblivious algorithms hand-crafted by human experts.

Development effort. We give case studies to show how ObliVM greatly reduces the development effort and expertise needed to create applications over secure computation.

New oblivious algorithms. We describe a few new oblivious algorithms that we uncover during this process of programming language and algorithms co-design. Specifically, we demonstrate new oblivious graph algorithms including oblivious Depth-First-Search for dense graphs, oblivious shortest path for sparse graphs, and an oblivious minimum spanning tree algorithm.

5.1.2 Threat Model, Deployment, and Scope

Deployment scenarios and threat model. ObliVM are designed for the same deployment scenarios under the same threat model as SCVM.

Scope. A subset of ObliVM's source language ObliVM-Lang has a security type system which, roughly speaking, ensures that the program's execution traces are independent of secret inputs [58, 59].

ObliVM-Lang's type system is further extended to support reasoning about the declassifications of *random numbers* to provide a principled guidance on how developers should use random numbers properly while enforcing security. This extended type system, however, does not guarantee the security on random number usages. We argue that this extended type system is still useful in capturing subtle bugs in the implementations like oblivious data structures. We leave developing a sound and complete type system to handle random numbers as a future work.

By designing a new language, ObliVM does not directly retrofit legacy code. Such a design choice maximizes opportunities for compile-time optimizations. We note, however, that in subsequent work joint with our collaborators [39], we have implemented a MIPS CPU in ObliVM, which can securely evaluate standard MIPS instructions in a way that leaks only the termination channel (i.e., total runtime of the program) – this secure MIPS CPU essentially provides backward compatibility atop ObliVM whenever needed.

5.2 Programming Language and Compiler

As mentioned earlier, we wish to design a powerful source language ObliVM-Lang such that an expert programmer can *i)* develop oblivious programming abstractions as libraries and offer them to non-specialist programmers; and *ii)* implement low-level circuit gadgets atop ObliVM-Lang.

ObliVM-Lang builds on top of the recent SCVM IR as described in Chapter 4 – the only known language to date that supports ORAM abstractions , and therefore offers scalability to big data. In this section, we will describe new features that ObliVM-Lang offers and explain intuitions behind our security type system.

As compelling applications of ObliVM-Lang, in Section 5.3, we give concrete case studies and show how to implement oblivious programming abstractions and low-level circuit libraries atop ObliVM-Lang.

5.2.1 Language features for expressiveness and efficiency

Security labels. Except for the new random type introduced in Section 5.2.2, all other variables and arrays are either of a `public` or `secure` type. `secure` variables are secret-shared between the two parties such that neither party sees the value. `public` variables are observable by both parties. Arrays can be publicly or secretly indexable. For example,

- `secure int10[public 1000] keys`: secret array contents but indices to the array must be public. This array will be secret shared but not placed in

ORAMs.

- `secure int10[secure 1000] keys`: This array will be placed in a secret-shared ORAM, and we allow secret indices into the array.

Standard features. ObliVM-Lang allows programmers to use C-style keyword `struct` to define *record types*. It also supports *generic types* similar to templates in C++. For example, a binary tree with public topological structure but secret per-node data can be defined without using pointers (assuming its capacity is 1000 nodes):

```
struct KeyValueTable<T> {
  secure int10[public 1000] keys;
  T[public 1000] values;
};
```

In the above, the type `int10` means that its value is a 10-bit signed integer. Each element in the array `values` has a generic type `T` similar to C++ templates. ObliVM-Lang assumes data of type `T` to be secret-shared. In the future, we will improve the compiler to support public generic types.

Generic constants. Besides general types, ObliVM-Lang also supports *generic constants* to further improve the reusability. Let us consider the following tree example:

```
struct TreeNode@m<T> {
  public int@m key;
  T value;
  public int@m left, right;
};
struct Tree@m<T> {
  TreeNode<T>[public (1<<m)-1] nodes;
  public int@m root;
};
```

This code defines a binary search tree implementation of a key-value store, where keys are m-bit integers. The *generic constant* `@m` is a variable whose value will be instantiated to a constant. It hints that `m` bits are enough to represent all the position references to the array. The type `int@m` refers to an integer type with `m` bits. Further, the capacity of array `nodes` can be determined by `m` as well (i.e. `(1<<m)-1`). Note that Zhang et al. [98] also allow specifying the length of an integer, but require this length to be a hard-coded constant – this necessitates modification and recompilation of the program for different inputs. ObliVM-Lang's generic constant approach eliminates this constraint, and thus improves reusability.

Functions. ObliVM-Lang allows programmers to define functions. For example, following the `Tree` defined as above, programmers can write a function to search the value associated with a given key in the tree as follows:

```
1  T Tree@m<T>.search(public int@m key) {
2    public int@m now = this.root, tk;
3    T ret;
4    while (now != -1) {
5      tk = this.nodes[now].key;
6      if (tk == key)
7        ret = this.nodes[now].value;
8      if (tk <= key)
9        now = this.nodes[now].right;
10     else
11       now = this.nodes[now].left;
12   }
13   return ret
14 };
```

This function is a method of a `Tree` object, and takes a `key` as input, and returns a

value of type `T`. The function body defines three local variables `now` and `tk` of type `public int@m`, and `ret` of type `T`. The definition of a local variable (e.g. `now`) can be accompanied with an optional initialization expression (e.g. `this.root`). When a variable (e.g. `ret` or `tk`) is not initialized explicitly, it is initialized to be a default value depending on its type.

The rest of the function is standard, C-like code, except that ObliVM-Lang requires exactly one return statement at the bottom of a function whose return type is not `void`. We highlight that ObliVM-Lang allows arbitrary looping on a public guard (e.g. line 4) without loop unrolling, which cannot be compiled in previous loop-elimination-based work [13, 43, 44, 55, 65, 98].

Function types. Programmers can define a variable to have function type, similar to function pointers in C. To avoid the complexity of handling arbitrary higher order functions, the input and return types of a function type must not be function types. Further, generic types cannot be instantiated with function types.

Native primitives. ObliVM-Lang supports native types and native functions. For example, ObliVM-Lang's default back end implementation is ObliVM-SC, which is implemented in Java. Suppose an alternative BigInteger implementation in ObliVM-SC (e.g., using additively homomorphic encryption) is available in a Java class called `BigInteger`. Programmers can define

```
typedef BigInt@m = native BigInteger;
```

Suppose that this class supports four operations: `add`, `multiply`, `fromInt` and `toInt`, where the first two operations are arithmetic operations and last two operations are used to convert between Garbled Circuit-based integers and HE-based

integers. We can expose these to the source language by declaring:

```
BigInt@m BigInt@m.add(BigInt@m x, BigInt@m y)
    = native BigInteger.add;
BigInt@m BigInt@m.multiply(BigInt@m x, BigInt@m y)
    = native BigInteger.multiply;
BigInt@m BigInt@m.fromInt(int@m y) = native BigInteger.fromInt;
int@m BigInt@m.toInt(BigInt@m y) = native BigInteger.toInt;
```

5.2.2 Language features for security

The key requirement of ObliVM-Lang is that a program's execution traces will not leak information. These execution traces include a memory trace, an instruction trace, a function stack trace, and a declassification trace. The trace definitions are similar to SCVM in Chapter 4, and we develop a security type system for ObliVM-Lang, which is similar to the one for SCVM to enforce trace obliviousness.

In addition, ObliVM-Lang provides an extended type system that imposes further constraints on how random numbers and functions should be used to achieve security. This type system extension does not enforce formal security, but it provides useful hints to capture subtle bugs, e.g., when implement oblivious data structures.

Random numbers and implicit declassifications. Many oblivious programs such as ORAM and oblivious data structures crucially rely on randomness. In particular, their obliviousness guarantee has the following nature: the joint *distribution* of memory traces is identical regardless of secret inputs (these algorithms typically have a cryptographically negligible probability of correctness failure). ObliVM-Lang

supports reasoning of such "distributional" trace-obliviousness by providing *random types* associated with an affine type system. For instance, `rnd32` is the type of a 32-bit random integer. A random number will always be secret-shared between the two parties.

To generate a random number, there is a built-in function `RND` with the following signature:

```
rnd@m RND(public int32 m)
```

This function takes a public 32-bit integer `m` as input, and returns `m` random bits. Note that `rnd@m` is a *dependent type*, whose type depends on values, i.e. `m`. To avoid the complexity of handling general dependent types, the ObliVM-Lang compiler restricts the usage of dependent types to only this built-in function, and handles it specially.

In our ObliVM framework, outputs of a computation can be explicitly declassified with special syntax. Random numbers are allowed *implicit declassification* – by assigning them to public variables. Here "implicitness" means that the declassification happens not because this is a specified outcome of the computation.

For security, we must ensure that each random number is implicitly declassified *at most once* for the following reason. When implicitly declassifying a random number, both parties observe the random number as part of the trace. Now consider the following example where `s` is a secret variable.

```
 1  rnd32 r1 = RND(32), r2= RND(32);
 2  public int32 z;
 3  if (s) z = r1; // implicit declass
 4  else z = r2; // implicit declass
    ......
XX  public int32 y = r2; // NOT OK
```

In this program, random variables `r1` and `r2` are initialized in Line 1 – these variables are assigned a fresh, random value upon initialization. Up to Line 4 random variables `r1` and `r2` are each declassified no more than once. Line XX, however, could potentially cause `r2` to be declassified more than once. Line XX clearly is not secure since in this case the observable public variable `y` and `z` could be correlated – depending on which secret branch was taken earlier.

Therefore, we use an *affine type* system to ensure that each random variable is implicitly declassified at most once. This way, each time a random variable is implicitly declassified, it will introduce a independently uniform variable to the observable trace. In our security proof, a simulator can just sample this random number to simulate the trace.

It turns out that the above example reflects the essence of what is needed to implement oblivious RAM and oblivious data structures in our source language. We refer the readers to Sections 5.3 and 5.4.2 for details.

Function calls and phantom functions. A straightforward idea to prevent stack behavior from leaking information is to enforce function calls in a public context. Then the requirement is that each function's body must satisfy memory- and instruction-trace obliviousness. Further, by defining native functions, ObliVM-Lang implicitly assumes that their implementations satisfy memory- and instruction-

trace obliviousness.

Beyond this basic idea, ObliVM-Lang makes a step forward to enabling *function calls within a secret if-statement* by introducing the notion of *phantom function*. The idea is that each function can be executed in dual modes, a *real* mode and and a *phantom* mode. In the real mode, all statements are executed normal with real computation and real side effects. In the phantom mode, the function execution merely simulates the memory traces of the real world; no side effects take place; and the phantom function call returns a secret-shared default value of the specified return type. This is similar to padding ideas used in several previous works [6, 78].

We will illustrate the use of phantom function with the following `prefixSum` example. The function `prefixSum(n)` accesses a global integer array `a`, and computes the prefix sum of the first `n` + 1 elements in `a`. After accessing each element (Line 3), the element in array `a` will be set to 0 (Line 4).

```
1  phantom secure int32 prefixSum
2      (public int32 n) {
3    secure int32 ret=a[n];
4    a[n]=0;
5    if (n != 0) ret = ret+prefixSum(n-1);
6    return ret;
7  }
```

The keyword `phantom` indicates that the function `prefixSum` is a phantom function.

Consider the following code to call the phantom functions:

```
if (s) then x = prefixSum(n);
```

To ensure security, `prefixSum` will always be called no matter `s` is true or

false. When `s` is false, however, it must be guaranteed that (1) elements in array `a` will not be assigned to be 0; and (2) the function generates traces with the same probability as when `s` is true. To this end, the compiler will generate target code with the following signature:

```
prefixSum(idx, indicator)
```

where `indicator` means whether the function will be called in the real or phantom mode. To achieve the first goal, the global variable will be modified only if `indicator` is false. The compiler will compile the code in line 4 into the following pseudo-code:

```
a[idx]=mux(0, a[idx], indicator);
```

It is easy to see, that all instructions will be executed, and thus the generated traces are identical regardless of the value of `indicator`. Note, that such a function is not implementable in any prior loop-unrolling based compiler, since n is provided at runtime only.

It is worth noticing that phantom function relaxed the restriction posed by previous memory trace oblivious type systems [58], which do not allow looping in the secure context (i.e. within a secret conditional). The main difficulty in previous systems was to quantify the numbers of loop iterations in the two branches of an if-statement, and to enforce the two numbers to be the same. Phantom functions remove the need of this analysis by executing both branches, with one branched really executed, and the other executed phantomly. As long as an adversary is unable to distinguish between a real execution from a phantom one, the secret

guard of the if-statement will not be leaked, even when loops are virtually present (i.e. in a phantom function).

5.3 User-Facing Oblivious Programming Abstractions

Programming abstractions such as MapReduce and GraphLab have been popularized in the parallel computing domain. In particular, programs written for a traditional sequential programming paradigm are difficult to parallelize automatically by an optimizing compiler. These new paradigms are not only easy for users to understand and program with, but also provide insights on the structure of the problem, and facilitate parallelization in an automated manner.

In this section, we would like to take a similar approach towards oblivious programming as well. The idea is to develop oblivious programming abstractions that can be easily understood and consumed by non-specialist programmers, and our compiler can compile programs into efficient oblivious algorithms. In comparison, if these programs were written in a traditional imperative-style programming language like C, compile-time optimizations would have been much more limited.

5.3.1 MapReduce Programming Abstractions

An interesting observation is that "parallelism facilitates obliviousness" [24, 36]. If a program (or part of a program) can be efficiently expressed in parallel programming paradigms such as MapReduce and GraphLab [2, 64] (with a few additional constraints), there is an efficient oblivious algorithm to compute this task.

We stress that in this paper, we consider MapReduce merely as a programming abstraction that facilitates obliviousness – in reality we compile MapReduce programs to *sequential* implementations that runs on a single thread. Parallelizing the algorithms is outside the scope of this work.

Background: Oblivious algorithms for streaming MapReduce. A *streaming* MapReduce program consists of two basic operations, `map` and `reduce`.

- The `map` operation: takes an array denoted $\{\alpha_i\}_{i\in[n]}$ where each $\alpha_i \in \mathcal{D}$ for some domain $\mathcal{D}$, and a function $\texttt{mapper} : \mathcal{D} \to \mathcal{K} \times \mathcal{V}$. Now `map` would apply $(k_i, v_i) := \texttt{mapper}(\alpha_i)$ to each α_i, and output an array of key-value pairs $\{(k_i, v_i)\}_{i\in[n]}$.

- The `reduce` operation: takes in an array of key-value pairs denoted $\{(k_i, v_i)\}_{i\in[n]}$ and a function $\texttt{reducer} : \mathcal{K} \times \mathcal{V}^2 \to \mathcal{V}$. For every unique key k value in this array, let $(k, v_{i_1}), (k, v_{i_2}), \ldots (k, v_{i_m})$ denote all occurrences with the key k. Now the `reduce` operation applies the following operation in a streaming fashion:

$$R_k := \texttt{reducer}(k, \ldots \texttt{reducer}(k, \texttt{reducer}(k, v_{i_1}, v_{i_2}), v_{i_3}), \ldots, v_{i_m})$$

  The result of the `reduce` operation is an array consisting of a pair (k, R_k) for every unique k value in the input array.

Goodrich and Mitzenmacher [36] observe that any program written in a streaming MapReduce abstraction can be converted to efficient oblivious algorithms, and they leverage this observation to aid the construction of an ORAM scheme.

```
1  Pair<K,V>[public n] MapReduce@m@n<I,K,V>
2      (I[public m] data,  Pair<K, V> map(I),
3       V reduce(K, V, V), V initialVal,
4       int2 cmp(K, K)) {
5    public int32 i;
6    Pair<K, V>[public m] d2;
7    for (i=0; i<m; i=i+1)
8      d2[i] = map(data[i]);
9    sort@m<K, V>(d2, 1, cmp);
10   K key = d2[0].k;
11   V val = initialVal;
12   Pair<int1, Pair<K, V>>[public m] res;
13   for (i=0; i+1<m; i=i+1) {
14     res[i].v.k = key;
15     res[i].v.v = val;
16     if (cmp(key, d2[i+1].k)==0) {
17       res[i].k.val = 1;
18     } else {
19       res[i].k.val = 0;
20       key = d2[i+1].k;
21       val = initialVal;
22     }
23     val = reduce(key, val, d2[i+1].v);
24   }
25   res[m-1].k.val = 0;
26   res[m-1].v.k = key;
27   res[m-1].v.v = val;
28   sort@m<int1, Pair<K, V>>
29     (res, 1, zeroOneCmp);
30   Pair<K, V>[public n] top;
31   for (i=0; i < n; i = i + 1)
32     top[i] = res[i].v;
33   return top;
34 }
```

Figure 5.1: **Streaming MapReduce in** ObliVM-Lang. See Section 5.3.1 for oblivious algorithms for the streaming MapReduce paradigm [36].

- The `map` operation is inherently oblivious, and can be done by making a linear scan over the input array.
- The `reduce` operation can be made oblivious through an oblivious sorting (denoted o-sort) primitive.
  - First, o-sort the input array in ascending order of the key, such that all pairs with the same key are grouped together.
  - Next, in a single linear scan, apply the `reducer` function: *i)* If this is the last key-value pair for some key k, write down the result of the aggregation (k, R_k). *ii)* Else, write down a dummy entry $\perp$.
  - Finally, o-sort all the resulting entries to move $\perp$ to the end.

Providing the streaming MapReduce abstraction in ObliVM. It is easy to implement the streaming MapReduce abstraction as a library in our source language ObliVM-Lang. The ObliVM-Lang implementation of streaming MapReduce paradigm is provided in Figure 5.1.

MapReduce has two generic constants, `m` and `n`, to represent the sizes of the input and output respectively. It also has three generic types, `I` for inputs' type, `K`, for output keys' type, and `V`, for output values' type. All of these three types are assumed to be secret.

It takes five inputs, `data` for the input data, `map` for the mapper, `reduce` for the reducer, `initialVal` for the initial value for the reducer, and `cmp` to compare two keys of type `K`.

Lines 6-10 are the mapper phase of the algorithm, then line 11 uses the function `sort` to sort the intermediate results based on their keys. After line 11, the intermediate results with the same key are grouped together, and line 12-29 produce the output of the reduce phase with some dummy outputs. Finally, lines 30-35 use oblivious sort again to eliminate those dummy outputs, and eventually line 36 returns the final results.

Notice that in these functions, there are three arrays, `data`, `d2`, and `res`. The program declares all of them to have only public access pattern, because they are accessed by either a sequential scan, or an oblivious sorting. In this case, the compiler will not place these arrays into ORAM banks.

Using MapReduce. Figure 5.1 needs to be written by an expert developer only once. From then on, an end user can make use of this programming abstraction.

We further illustrate how to use the above MapReduce program to implement a histogram. In SCVM (Chapter 4), a histogram program is as below.

```
for (public int i=0; i<n; ++i) c[i] = 0;
for (public int i=0; i<m; ++i) c[a[i]] ++;
```

This program counts the frequency of each values in $[0..\mathtt{n}-1]$ in the array `a` of size `m`. Since the program makes dynamic memory accesses, the SCVM compiler would decide to put the array `c` inside an ORAM.

An end user can write the same program using a simple MapReduce abstraction as follows. Our ObliVM-Lang compiler would generate target code that relies

on oblivious sorting primitives rather than generic ORAM, improving the performance by a logarithmic factor in comparison with the SCVM implementation. In Section 5.5, we show that the practical performance gain ranges from 10× to 400×.

```
int2 cmp(int32 x, int32 y) {
  int2 r = 0;
  if (x < y) r = -1;
  else if (x > y) r = 1;
  return r;
}
Pair<int32, int32> mapper(int32 x) {
  return Pair<int32, int32>(x, 1);
}
int32 reducer(int32 k, int32 v1, int32 v2) {
  return v1 + v2;
}
```

The top-level program can launch the computation using

```
c=MapReduce@m@n<int32, int32, int32>(a, mapper, reducer, cmp, 0);
```

5.3.2 Programming Abstractions for Data Structures

We now explain how to provide programming abstractions for a class of pointer-based oblivious data structures described by Wang et al. [92]. Figure 5.2 gives an example, where an expert programmer provides library support (Figure 5.3) for implementing a class of pointer-based data structures such that a non-specialist programmer can implement data structures which will be compiled to efficient oblivious algorithms that outperform generic ORAM. We stress that while we give a stack example for simplicity, this paradigm is also applicable to other pointer-based data structures, such as AVL tree, heap, and queue.

```
1  struct StackNode@m<T> {
2    Pointer@m next;
3    T data;
4  };
5  struct Stack@m<T> {
6    Pointer@m top;
7    SecStore@m store;
8  };
9  phantom void Stack@m<T>.push(T data) {
10   StackNode@m<T> node = StackNode@m<T> (
       top, data);
11   this.top = this.store.allocate();
12   store.add(top.(index, pos), node);
13 }
14 phantom T Stack@m<T>.pop() {
15   StackNode@m<T> res = store
       .readAndRemove(top.(index, pos));
16   top = res.next;
17   return res.data;
18 }
```

Figure 5.2: Oblivious stack by non-specialist programmers.

Implementing oblivious data structure abstractions in ObliVM. We assume that the reader is familiar with the oblivious data structure algorithmic techniques described by Wang et al. [92]. To support efficient data structure implementations, an expert programmer implements two important objects (see Figure 5.3):

- A `Pointer` object stores two important pieces of information: an `index` variable that stores the logical identifier of the memory block pointed to (each memory block has a globally unique `index`); and a `pos` variable that stores the random leaf label in the ORAM tree of the memory block.

- A `SecStore` object essentially implements an ORAM, and provides the following member functions to an end-user: The `SecStore.remove` function essen-

```
1  rnd@m RND(public int32 m) = native lib.rand;
2  struct Pointer@m {
3    int32 index;
4    rnd@m pos;
5  };
6  struct SecStore@m<T> {
7    CircuitORAM@m<T> oram;
8    int32 cnt;
9  };
10 phantom void SecStore@m<T>.add(int32 index,
     int@m pos, T data) {
11   oram.add(index, pos, data);
12 }
13 phantom T SecStore@m<T>
     .readAndRemove(int32 index, rnd@m pos) {
14   return oram.readAndRemove(index, pos);
15 }
16 phantom Pointer@m SecStore@m<T>.allocate() {
17   cnt = cnt + 1;
18   return Pointer@m(cnt, RND(m));
19 }
```

Figure 5.3: Code by expert programmers to help non-specialists implement oblivious stack.

tially is a syntactic sugar for the ORAM's `readAndRemove` interface [80, 90], and the `SecStore.add` function is a syntactic sugar for the ORAM's `Add` interface [80, 90]. Finally, the `SecStore.allocate` function returns a new `Pointer` object to the caller. This new `Pointer` object is assigned a globally unique logical identifier (using a counter `cnt` that is incremented each time), and a fresh random chosen leaf label `RND`(m).

Stack implementation by a non-specialist programmer. Given abstractions provided by the expert programmer, a non-specialist programmer can now implement a class of data structures such as stack, queue, heap, AVL Tree, etc. Figure 5.2 gives a stack example.

Role of affine type system. We use Figure 5.3 as an example to illustrate how our `rnd` types with their affine type system can ensure security. As mentioned earlier, `rnd` types have an affine type system. This means that each `rnd` can be declassified (i.e., made public) at most once. In Figure 5.3, the `oram.readAndRemove` call will declassify its argument `rnd@m pos` inside the implementation of the function body. From an algorithms perspective, this is because the leaf label `pos` will be revealed during the `readAndRemove` operation, incurring a memory trace where the value `rnd@m pos` will be observable by the adversary.

5.3.3 Loop Coalescing and New Oblivious Graph Algorithms

We introduce a new programming abstraction called loop coalescing, and show how this programming abstraction allowed us to design novel oblivious graph algorithms such as Dijkstra's shortest path and minimum spanning tree for sparse graphs. Loop coalescing is non-trivial to embed as a library in ObliVM-Lang. We therefore support this programming abstraction by introducing special syntax and modifications to our compiler. Specifically, we introduce a new syntax called *bounded-for* loop as shown in Figure 5.4. *For succinctness, in this section, we will present pseudo-code.*

In the program in Figure 5.4, the `bwhile(n)` and `bwhile(m)` syntax at Lines 1 and 3 indicate that the outer loop will be executed for a total of n times, whereas the inner loop will be executed for a total of m times – over all iterations of the outer loop.

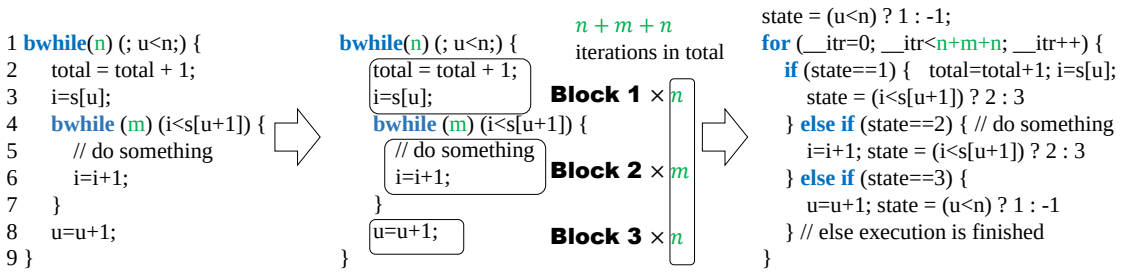

Figure 5.4: **Loop coalescing.** The outer loop will be executed at most n times in total, the inner loop will be executed at most m times in total – over all iterations of the outer loop. A naive approach compiler would pad the outer and inner loop to n and m respectively, incurring $O(nm)$ cost. Our loop coalescing technique achieves $O(n+m)$ cost instead.

Algorithms	Our Complexity	Best Known
Dijkstra's Algorithm (Sparse Graph)	$O((E+V)\log^2 V)$	$O((E+V)\log^3 V)$ (Generic ORAM baseline [90])
Prim's Algorithm (Sparse Graph)	$O((E+V)\log^2 V)$	$O(E\frac{\log^3 V}{\log\log V})$ for $E = O(V\log^\gamma V), \gamma \geq 0$ [37] $O(E\frac{\log^3 V}{\log^\delta V})$ for $E = O(V2^{\log^\delta V}), \delta \in (0,1)$ [37] $O(E\log^2 V)$ for $E = \Omega(V^{1+\epsilon}), \epsilon \in (0,1]$ [37]

Table 5.1: **Summary of algorithmic results.** All costs reported are in terms of circuit size. The asymptotic notation omits the bit-length of each word for simplicity. Our oblivious Dijkstra's algorithm and oblivious Prim's algorithm can be composed using our novel loop coalescing programming abstraction and oblivious data structures.

Algorithm 1 Dijkstra' algorithm with bounded for

Secret Input: `s`: the source node
Secret Input: `e`: concatenation of adjacency lists stored in a single ORAM array. Each vertex's neighbors are stored adjacent to each other.
Secret Input: `s[u]`: sum of out-degree over vertices from 1 to u.
Output: `dis`: the shortest distance from `source` to each node

```
1:  dis := [∞, ∞, ..., ∞]
2:  PQ.push(0, s)
3:  dis[s] := 0
4:  bwhile(V)(!PQ.empty())
5:      (dist, u) := PQ.deleteMin()
6:      if(dis[u] == dist) then
7:          dis[u] := −dis[u];
8:          bfor(E)(i := s[u]; i < s[u + 1]; i = i + 1)
9:              (u, v, w) := e[i];
10:             newDist := dist + w
11:             if (newDist < dis[v]) then
12:                 dis[v] := newDist
13:                 PQ.insert(newDist, u)
```

To deal with loop coalescing, the compiler partitions the code within an bounded-loop into code blocks, each of which will branch at the end. The number of execution times for each code block will be computed as the bound number for the inner most bounded-loop that contains the code block. Then the compiler will transform a bounded loop into a normal loop, whose body simulates a state machine that each state contains a code block, and the branching statement at the end of each code block will be translated into an assignment statement that moves the state machine into a next state. The total number of iterations of the emitted normal loop is the summation of the execution times for all code blocks. Figure 5.4 illustrates this compilation process.

We now show how this loop coalescing technique leads to new novel oblivious graph algorithms.

Algorithm 2 Oblivious Dijkstra' algorithm

Secret Input: `e, s`: same as Algorithm 1
Output: `dis`: the shortest distance from s to each node

```
1: dis := [∞, ∞, ..., ∞]; dis[source] = 0
2: PQ.push(0, s); innerLoop := false
3: for i := 0 → 2V + E do
4:     if not innerLoop then
5:         (dist, u) := PQ.deleteMin()
6:         if dis[u] == dist then
7:             dis[u] := −dis[u]; i := s[u]
8:             innerloop := true;
9:         end if
10:    else
11:        if i < s[u + 1] then
12:            (u, v, w):= e[i]
13:            newDist := dist + w
14:            if newDist < dis[u] then
15:                dis[u] := newDist
16:                PQ.insert(newDist, v′)
17:            end if
18:            i = i + 1
19:        else
20:            innerloop := false;
21:        end if
22:    end if
23: end for
```

Oblivious Dijkstra shortest path for sparse graphs. It is an open problem how to compute single source shortest path (SSSP) obliviously for *sparse graphs* more efficiently than generic ORAM approaches. Blanton et al. [12] designed a solution for a *dense* graph, but their technique cannot be applied when the graph is sparse.

Recall that the priority-queue-based Dijkstra's algorithm has to update the weight whenever a shorter path is found to any vertex. In an oblivious version of Dijkstra's, this operation dominates the overhead, as it is unclear how to realize it more efficiently than using generic ORAMs. Our solution to oblivious SSSP is

more efficient thanks to (1) avoiding this weight update operation, and (2) a *loop coalescing* technique.

Avoiding weights updating. This is accomplished by two changes to a standard priority-queue-based Dijkstra's algorithm, i.e., lines 6-7 and line 12 in Algorithm 1. The basic idea is, whenever a shorter distance `newDist` from s to a vertex u is found, instead of updating the existing weight of u in the heap, we insert a new pair $(\texttt{newDis}, u)$ into the priority queue. This change can result in multiple entries for the same vertex in the queue, leading to two concerns: (1) the size of the priority queue cannot be bounded by V; and (2) the same vertex might be popped and processed multiple times from the queue. Regarding the first concern, we note the size of the queue can be bounded by $E = O(V^2)$ (since $E = o(V^2)$ for sparse graphs). Hence, each priority queue `insert` and `deleteMin` operation can still be implemented obliviously in $O(\log^2 V)$ [92]. The second concern is resolved by the check in lines 6-7: every vertex will be processed at most once because $\texttt{dis}[v]$ will be set negative once vertex v is processed.

Loop coalescing. In Algorithm 1, the two nested loops (line 4 and line 8) use secret data as guards. In order not to leak the secret loop guards, a naive approach is to iterate each loop a maximal number of times (i.e., $V+E$, as V alone is considered secret).

Using our loop coalescing technique, we can derive an oblivious Dijkstra's algorithm that asymptotically outperforms a generic ORAM baseline for sparse graphs. The resulting oblivious algorithm is described in Algorithm 2. Note that at most V vertices and E edges will be visited, we coalesce the two loops into a single

one. The code uses a state variable `innerloop` to indicate whether a vertex or an edge is being processed. Each iteration deals with one of a vertex (lines 5-8), an edge (lines 15-18), or the end of a vertex's edges (line 13). So there are $2V + E$ iterations in total. Note the ObliVM-Lang compiler will pad the `if`-branches in Algorithm 2 to ensure obliviousness. Further, an oblivious priority queue is employed for `PQ`.

Cost analysis. In Algorithm 2, each iteration of the loop (lines 3-18) makes a constant number of ORAM accesses and two priority queue primitives (`insert` and `deleteMin`, both implemented in $O(\log^2 V)$ time). So, the total runtime is $O((V + E) \log^2 V)$.

Additional algorithmic results. Summarized in Table 5.1, our loop coalescing technique also immediately gives a new oblivious Minimum Spanning Tree (MST) algorithm whose full description is omitted.

5.4 Implementing Rich Circuit Libraries

5.4.1 Case Study: Basic Arithmetic Operations

The rich language features provided by ObliVM-Lang make it possible to implement complex arithmetic operations easily and efficiently. We give a case study to demonstrate how to use ObliVM-Lang to implement Karatsuba multiplication.

Implementing Karatsuba multiplication. Figure 5.5 contains the implementation of Karatsuba multiplication [50] in ObliVM-Lang. Karatsuba multiplication implements the following recursive algorithm to compute multiplication of two n bit numbers, `x` and `y`, taking $O(n^{\log_2 3})$ amount of time. As a quick overview, the

```
1  int@(2*n) karatsubaMult@n(
     int@n x, int@n y) {
2    int@2*n ret;
3    if (n < 18) {
4      ret = x*y;
5    } else {
6      int@(n-n/2) a = x$n/2~n$;
7      int@(n/2) b = x$0~n/2$;
8      int@(n-n/2) c = y$n/2~n$;
9      int@(n/2) d = y$0~n/2$;
10     int@(2*(n-n/2)) t1 =
         karatsubaMult@(n-n/2)(a, c);
11     int@(2*(n/2)) t2 =
         karatsubaMult@(n/2)(b, d);
12     int@(n-n/2+1) aPb = a + b;
13     int@(n-n/2+1) cPd = c + d;
14     int@(2*(n-n/2+1)) t3 =
         karatsubaMult@(n-n/2+1)(aPb, cPd);
15     int@(2*n) padt1 = t1;
16     int@(2*n) padt2 = t2;
17     int@(2*n) padt3 = t3;
18     ret = (padt1<<(n/2*2)) + padt2 +
       ((padt3 - padt1 - padt2)<<(n/2));
19   }
20   return ret;
21 }
```

Figure 5.5: **Karatsuba multiplication in** ObliVM-Lang**.**

algorithm works as follows. First, express the n-bit integers `x` and `y` as the concatenation of $\frac{n}{2}$-bit integers: `x = a*2`$^{n/2}$`+b, y = c*2`$^{n/2}$`+d`. Now, `x*y` can be calculated as follows:

```
t1 = a*c; t2 = b*d; t3 = (a+b)*(c+d);

x*y = t1<<n + t2 + (t3-t1-t2)<<(n/2);
```

where the multiplications `a*c` and `b*d` are implemented through a recursive call to the Karatsuba algorithm itself (until the bit-length is small enough).

To implement Karatsuba efficiently, we need to perform operations on a subset of bits. We hence introduce the following syntactic sugar in ObliVM-Lang: In lines

```
1  #define BUCSIZE 3
2  #define STASHSIZE 33
3  struct Block@n<T>{
4    int@n id, pos;
5    T  data;
6  };
7  struct CircuitOram@n<T>{
8    dummy Block@n<T>[public 1<<n+1]
       [public BUCSIZE] buckets;
9    dummy Block@n<T>[public STASHSIZE] stash;
10 };
```

Figure 5.6: **Part of our Circuit ORAM implementation (Type Definition) in** ObliVM-Lang**.**

6 to 9 of Figure 5.5, the syntax `num$i~j$` means extracting the part of integer `num` from i-th bit to j-th bit.

5.4.2 Case Study: Circuit ORAM

In Figure 5.7, we show part of the Circuit ORAM implementation using ObliVM-Lang. Line 3 to line 6 is the definition of a ORAM block containing two metadata fields of an index of type `int`, and a position label of type `rnd`, along with a data field of type `<T>`.

Circuit ORAM (line 7-10) is organized to contain an array of buckets (i.e. arrays of ORAM blocks), and a stash (i.e. an array of blocks). The `dummy` keyword in front of `Block@n<T>` indicates the value of this type can be `null`. In many cases, (e.g. Circuit ORAM implementation), using `dummy` keyword leads to a more efficient circuit generation.

Line 11-30 demonstrates how `readAndRemove` can be implemented. Taking the input of an secret integer index `id`, and a random position label `pos`, the label

```
11  phantom T  CircuitOram@n<T>
      .readAndRemove(int@n id, rnd@n pos) {
12    public int32 pubPos = pos;
13    public int32 i = (1 << n) + pubPos;
14    T  res;
15    for (public int32 k = n; k>=0; k=k-1) {
16      for (public int32 j=0;j<BUCSIZE;j=j+1)
17        if (buckets[i][j] != null &&
18              buckets[i][j].id == id){
19            res = buckets[i][j].data;
20            buckets[i][j] = null;
21        }
22      i=(i-1)/2;
23    }
24    for (public int32 i=0;i<STASHSIZE;i=i+1)
25      if (stash[i]!=null&&stash[i].id==id) {
26          res = stash[i].data;
27          stash[i] = null;
28      }
29    return res;
30  }
```

Figure 5.7: **Part of our Circuit ORAM implementation (ReadAndRemove) in** ObliVM-Lang**.**

`pos` is first declassified into public. Then affine type system allows declassifying `pos` once, i.e. `pos` is never used for the rest of the program. Further in a function calling `readAndRemove` with inputs `arg1` and `arg2`, `arg2` cannot be used either for the rest of the program. This is crucial to enforce that every position labels will use revealed only once after its generation, and, to our best knowledge, no prior work enables such an enforcement in a compiler.

This Circuit ORAM implementation can be type-checked by ObliVM-Lang's extended type checker, which gives users stronger confidence that the implementation does not leak information through its execution traces.

5.5 Evaluation

5.5.1 Back End Implementation

Our compiler emits code to a Java-based secure computation back end called ObliVM-SC. ObliVM-SC is designed to be extensible through a central notion called *computational environments*. Conceptually, our compiler emits circuit designs; whereas a computation environment decides how a circuit design, namely, how each AND and XOR gate will be executed. In other words, computational environments provide a separation between circuit designs and their executions, allowing circuit gadgets to be potentially reusable for multiple cryptographic protocols, such as Garbled Circuit [95] or GMW [34]. Currently, ObliVM provides a Garbled Circuit protocol with semi-honest security. However, adapting a circuit design to a different protocol such as GMW would simply require changing to an alternative computation environment, and does not involve modification of the compiler.

5.5.2 Metrics and Experiment Setup

Number of AND gates. In Garbled Circuit-based secure computation, functions are represented in boolean circuits consisting of XOR and AND gates. Due to well-known Free XOR techniques [8, 17, 53], the cost of evaluating XOR gates are insignificant in comparison with AND gates. Therefore, a primary performance metric is *the number of AND gates*. This metric is *platform independent*, i.e., independent of the artifacts of the underlying software implementation, or the hardware

configurations where the benchmark numbers are measured. This metric facilitates a fair comparison with existing works based on boolean circuits, and is one of the most popular metrics used in earlier works [44, 46, 54, 55, 59, 71, 72, 91, 92].

Wall-clock runtime. Unless noted otherwise, all wall-clock numbers are measured by executing the protocols between two Amazon EC2 machines of types c4.8xlarge and c3.8xlarge. This metric is platform and implementation dependent, and therefore we will explain how to best interpret wallclock runtimes, and how these runtimes will be affected by the underlying software and hardware configurations.

Compilation time. For all programs we ran, the compilation time is under 1 second. Therefore, we do not separately report the compilation time for each program.

5.5.3 Comparison with Previous Automated Approaches

The first general-purpose secure computation system, Fairplay, was built in 2004 [65]. Since then, several improved systems were built [13, 43, 44, 46, 54, 55, 98]. Except for our prior work SCVM, existing systems provide no support for ORAM – and therefore, each dynamic memory access would be compiled to a linear scan of memory.

We now evaluate the speedup ObliVM achieves relative to previous approaches. To illustrate the sources of the speedup, we consider the following sequence of progressive baselines. We start from Baseline 1 which is representative of a state-of-the-art automated secure computation system. We then add one feature at a time to the baseline, resulting in the next baseline, until we arrive at Baseline 5 which is

<table>
<tr><th></th><th>Oblivious programming abstractions and compiler optimizations demonstrated</th><th>Parameters for Figure 5.8</th><th>Parameters for Table 5.3 and Table 5.4</th></tr>
<tr><td>Dijkstra's Algorithm MST</td><td>Loop coalescing abstraction (see Section 5.3.3).</td><td>$V = 2^{14}, E = 3V$</td><td>$V = 2^{10}, E = 3V$</td></tr>
<tr><td>Heap Map/Set</td><td rowspan="2">Oblivious data structure abstraction (see Section 5.3.2).</td><td>$N = 2^{27}, K = 32, V = 480$</td><td rowspan="2">$N = 2^{23}, K = 32, V = 992$</td></tr>
<tr><td>Binary Search</td><td>$N = 2^{23}, K = 32, V = 992$</td></tr>
<tr><td>AMS Sketch</td><td rowspan="2">Compile-time optimizations: split data into separate ORAMs [59].</td><td>$\epsilon = 6 \times 10^{-5}, \delta = 2^{-20}$</td><td>$\epsilon = 2.4 \times 10^{-4}, \delta = 2^{-20}$</td></tr>
<tr><td>Count Min Sketch</td><td colspan="2">$\epsilon = 3 \times 10^{-6}, \delta = 2^{-20}$</td></tr>
<tr><td>K-Means</td><td>MapReduce abstraction (see Section 5.3.1).</td><td>$N = 2^{18}$</td><td>$N = 2^{16}$</td></tr>
</table>

Table 5.2: **List of applications used in Figures 5.8.** For graph algorithms, V, E stand for number of vertices and edges; for data structures, N, K, V stand for capacity, bit-length of key and bit-length of value; for streaming algorithms, ϵ, δ stand for relative error and failure probability; for K-Means, N stands for number of points.

essentially our ObliVM system.

- **Baseline 1: A state-of-the-art automated system with no ORAM support.** Baseline 1 is intended to characterize a state-of-the-art automated secure computation system with no ORAM support. We assume a compiler that can detect public memory accesses (whose addresses are statically inferrable), and directly make such memory accesses. For each each dynamic memory access (whose address depends on secret inputs), a linear scan of memory is employed. Baseline 1 is effectively a lower-bound estimate of the cost incurred by CMBC-GC [44], a state-of-the-art system in 2012.

- **Baseline 2: With GO-ORAM [35].** In Baseline 2, we implement the GO-ORAM scheme on top of Baseline 1. Dynamic memory accesses made by a program will be compiled to GO-ORAM accesses. We make no additional compile-time optimizations.

- **Baseline 3: With Circuit ORAM [90].** Baseline 3 is essentially the same as Baseline 2 except that we now replace the ORAM scheme with a state-of-the-art Circuit ORAM scheme [90].

- **Baseline 4: Language and compiler.** Baseline 4 assumes that the ObliVM language and compiler are additionally employed (on top of Baseline 3), resulting in additional savings due to our compile-time optimizations as well as our oblivious programming abstractions.

- **Baseline 5: Back end optimizations.** In Baseline 5, we employ additional

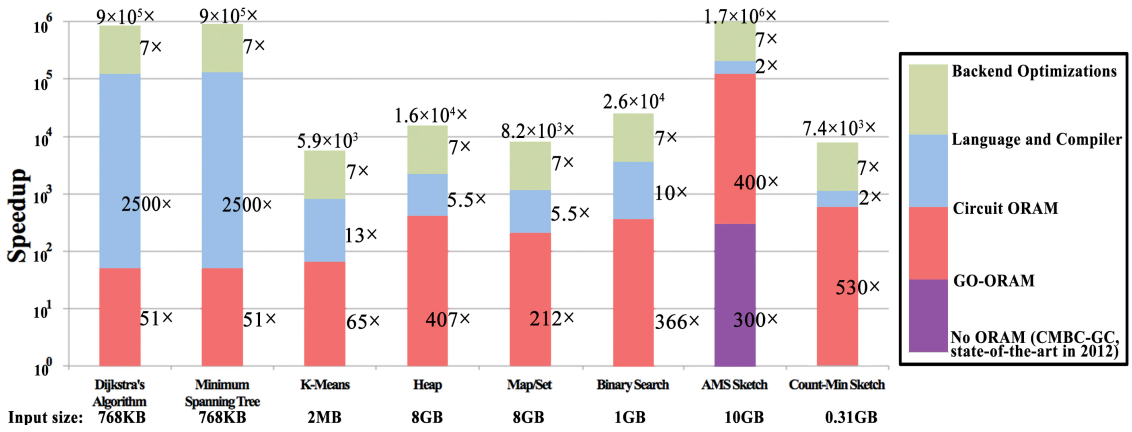

Figure 5.8: **Sources of speedup in comparison with state-of-the-art in 2012 [44]: an in-depth look.**

back end optimizations atop Baseline 4. Baseline 5 reflects the performance of the actual ObliVM system.

We consider a set of applications in our evaluation as described in Table 5.2. We select several applications to showcase our oblivious programming abstractions, including MapReduce, loop coalescing, and oblivious data structure abstractions. For all applications, we choose moderately large data sizes ranging from 768KB to 10GB. For data structures (e.g., Heap, Map/Set) and binary search, for Baseline 1, we assume that each operation (e.g., search, add, delete) is done with a single linear scan. For Baseline 2 and 3, we assume that a typical sub-linear implementation is adopted. For all other applications, we assume that Baseline 1 to 3 adopt the most straightforward implementation of the algorithm.

Results. Figure 5.8 shows the speedup we achieve relative to a state-of-the-art automated system that does not employ ORAM [44]. This speedup comes from the following sources:

No ORAM to GO-ORAM: For most of the cases, the data size considered was

not big enough for GO-ORAM to be competitive to a linear-scan ORAM. The only exception was AMS sketch, where we chose a large sketch size. In this case, using GO-ORAM would result in a 300× speedup in comparison with no ORAM (i.e., linear-scan for each dynamic memory access). This part of the speedup is reflected in purple in Figure 5.8. Here the speedup stems from a reduction in circuit size (as measured by the number of AND gates).

Circuit ORAM: The red parts in Figure 5.8 reflect the multiplicative speedup attained when we instead use Circuit ORAM (as opposed to no ORAM or GO-ORAM, whichever is faster). This way, we achieve an additional 51× to 530 performance gains – reflected by a reduction in the total circuit size.

Language and compiler: As reflected by the blue bars in Figure 5.8, our oblivious programming abstractions and compile-time optimizations bring an additional 2× to 2500× performance savings on top of a generic Circuit ORAM-based approach. This speedup is also measurable in terms of reduction in the circuit size.

Back end optimizations: Our ObliVM-SC is a better architected and more optimized version of its predecessor FastGC [46] which is employed by CMBC-GC [44]. FastGC [46] reported a garbling speed of 96K AND gates/sec, whereas ObliVM garbles at 670K AND gates/sec on a comparable machine. In total, we achieve an 7× overall speedup compared with FastGC [46].

We stress, however, that ObliVM's main contribution is not the back end implementation. In fact, it would be faster to hook up ObliVM's language and compiler with a JustGarble-like system that employs a C-based implementation and hardware

AES-NI. However, presently JustGarble does not provide a fully working end-to-end protocol. Therefore, it is an important direction of future work to extend JustGarble to a fully working protocol, and integrate it into ObliVM.

Comparison with SCVM. In comparison with SCVM, ObliVM's offers the following new features: 1) new oblivious programming abstractions; 2) Circuit ORAM implementation that is 20× to 30× times faster than SCVM's binary-tree ORAM implementation for 4MB to 4GB data sizes; and 3) ability to implement low-level gadgets including the ORAM algorithm itself in the source language.

Since the design of efficient ORAM algorithms is mainly the contribution of the Circuit ORAM paper [90], here we focus on evaluating the gains from programming abstractions. Therefore, instead of comparing with SCVM per se, we compare with SCVM + Circuit ORAM instead (i.e., SCVM with its ORAM implementation updated to the latest Circuit ORAM).

5.5.4 ObliVM vs. Hand-Crafted Solutions

We show that ObliVM achieves competitive performance relative to hand-crafted solutions for a wide class of common tasks. We also show that ObliVM significantly reduces development effort in comparison with previous secure computation frameworks.

Competitive performance. For a set of applications, including Heap, Map/Set, AMS Sketch, Count-Min Sketch, and K-Means, we compared implementations auto-generated by ObliVM with implementations hand-crafted by human experts. Here

the human experts are authors of this paper. We assume that the human experts have wisdom of employing the most efficient, state-of-the-art oblivious algorithms when designing circuits for these algorithms. For example, Histogram and K-Means algorithms are implemented with oblivious sorting protocols instead of generic ORAM. Heap and Map/Set employ state-of-the-art oblivious data structure techniques [92]. The graph algorithms including Dijkstra and MST employ novel oblivious algorithms proposed in this paper. In comparison, our ObliVM programs for the same applications do not require special security expertise to create. The programmer simply has to express these tasks in the programming abstractions we offer whenever possible. Over the suite of application benchmarks we consider, our ObliVM programs are competitive to hand-crafted implementations – and the performance difference is only $\mathbf{0.5}\% - \mathbf{2}\%$ throughout.

We remark that the hand-crafted circuits are not necessarily the optimal circuits for each computation task. However, they do represent asymptotically the best known algorithms (or new algorithms that are direct implications of this paper). It is conceivable that circuit optimization techniques such as those proposed in TinyGarble [82] can further reduce circuit sizes by a small constant factor (e.g., 50%). We leave this part as an interesting direction of future research.

Developer effort. We use two concrete case studies to demonstrate the significant reduction of developer effort enabled by ObliVM.

Case study: ridge regression. Ridge regression [41] takes as input a large number of data points and finds the best-fit linear curve for these points. The algorithm

is an important building block in various machine-learning tasks [72]. Previously, Nikolaenko et al. [72] developed a system to securely evaluate ridge regression, using the FastGC framework [46], which took them roughly **three weeks** [5]. In contrast, we spent **two student·hours** to accomplish the same task using ObliVM. In addition to the speedup gain from ObliVM-SC back end, our optimized libraries result in 3× smaller circuits with aligned parameters. We defer the detailed comparison to the online technical report [61].

Case study: oblivious data structures. Oblivious AVL tree (i.e, the Map/Set data structure) is an example algorithm that was previously too complex to program as circuits, but now becomes very easy with ObliVM. In an earlier work [92], we designed an oblivious AVL tree algorithm, but were unable to implement it due to high programming complexity. Now, with ObliVM, we implement an AVL tree with 311 lines of code in ObliVM-Lang, consuming under 10 student·hours (including the implementation as well as debugging).

We stress that it is not possible to implement oblivious AVL tree in previous languages for secure computation, including the state-of-the-art Wysteria [75].

5.5.5 End-to-End Application Performance

Currently in ObliVM-SC, we implemented a standard garbling scheme with Garbled Row Reduction [69] and FreeXOR [53]. We also implemented an OT extension protocol proposed by Ishai et al. [48] and a basic OT protocol by Naor and Pinkas [68].

Program	Input size	CMBC-GC (estimate)		ObliVM Framework			ObliVM + JustGarble (estimate)	
		#AND gates	Total time	#AND gates	Total time	Online time	Total time	Online time
Basic instructions								
Integer addition	1024 bits	2977	31ms	**1024**	**1.7ms**	**0.6ms**	0.12ms	0.05ms
Integer mult.	1024 bits	6.4M	66.4s	**572K**	**833ms**	**274ms**	69.4ms	28.9ms
Integer Comparison	16384 bits	32K	335.7ms	**16384**	**26ms**	**8.58ms**	1.96ms	0.82ms
Floating point addition	64 bits	10K	104ms	**3035**	**4.32ms**	**1.45ms**	0.36ms	0.15ms
Floating point mult.	64 bits	10K	104ms	**4312**	**6.29ms**	**2.02ms**	0.52ms	0.22ms
Hamming distance	1600 bits	30K	310ms	**3200**	**5.07ms**	**1.71ms**	0.39ms	0.16ms
Linear or super-linear algorithms								
K-Means	0.5MB	550B	66d	**2269M**	**62.1min**	**23.6min**	4.58min	1.9min
Dijkstra's Algorithm	48KB	755B	91d	**10B**	**12.6h**	**3.09h**	20.4min	8.5min
MST	48KB	755B	91d	**9.6B**	**12.4h**	**3h**	19.6min	8.2min
Histogram	0.25MB	137B	16.5d	**866M**	**21.5min**	**8.56min**	1.7min	42.5s
Sublinear-time algorithms								
Heap	1GB	32B	3.9d	**12.5M**	**59.3s**	**10.42s**	1.5s	625ms
Map/Set	1GB	32B	3.9d	**23.9M**	**117.2s**	**20.67s**	2.9s	1.2s
Binary Search	1GB	32B	3.9d	**1562K**	**7.36s**	**1.34s**	189ms	78.8ms
Count Min Sketch	0.31GB	9.9B	30.8h	**8088K**	**20.77s**	**6.4s**	0.98s	0.41s
AMS Sketch	1.25GB	40B	5.18d	**9949K**	**36.76s**	**9.95s**	1.21s	504ms

Table 5.3: **Application performance.** Actual measured numbers are in bold. The remainder are estimated numbers and should be interpreted with care. ObliVM numbers for basic instructions and sublinear-time algorithms are the mean of 20 runs. Since for all these applications, our measurements have small spread (all runs are within 6% from the mean), we use a single run for linear-time and super-linear algorithms (the same for Table 5.4).

Setup. For evaluation, here we consider a scenario where a client secret shares its data between two non-colluding cloud providers a priori. For cases where inputs are a large dataset (e.g., Heap, Map/Set, etc), depending on the application, the client may sometimes need to place the inputs in an ORAM, and secret-share the resulting ORAM among the two cloud providers. We do not measure this setup cost in our evaluation – this cost can depend highly on the available bandwidth between the client and the two cloud providers. Therefore, our evaluation begins assuming this one-time setup has completed.

End-to-end application performance. In Table 5.3, we consider three types of applications, basic instructions (e.g., addition, multiplication, and floating point operations); linear or super-linear algorithms (e.g., Dijkstra, K-Means, Minimum Spanning Tree, and Histogram); and sublinear-time algorithms (e.g., Heap, Map/Set, Binary Search, Count Min Sketch, AMS Sketch). We report the circuit size, online and total costs for a variety of applications at typical data sizes.

In Table 5.3, we also compare ObliVM with a state-of-the-art automated secure computation system CMBC-GC [44]. We note that the authors of CMBC-GC did not run all of these application benchmarks, so we project the performance of CMBC-GC using the following estimate: we first change our compiler to adopt a linear scan of memory upon dynamic memory accesses – this allows us to obtain an estimate of the circuit size CMBC-GC would have obtained for the same applications. For the set of application benchmarks (e.g., K-Means, MST, etc) CMBC-GC did report in their paper, we confirmed that our circuit size estimates are always a lower bound of what CMBC-GC reported. We then estimate the runtime of CMBC-

GC based on their reported 96K AND gates per sec – assuming that a network bandwidth of at least 2.8MBps is provisioned.

As mentioned earlier, the focus of this paper is our language and compiler, not the back end cryptographic implementation. It should be relatively easy to integrate our language and compiler with a JustGarble-like back end that employs hardware AES-NI. In Table 5.3, we also give an estimate of the performance we anticipate if we ran our ObliVM-generated circuits over a JustGarble-like back end. This is calculated using our circuit sizes and the 11M AND gates/sec performance number reported by JustGarble [11].

- *Online cost.* To measure online cost, we assume that all work that is independent of input data is performed offline, including garbling and input-independent OT preprocessing. Our present ObliVM implementation achieves an online speed of 1.8M gates/sec consuming roughly 54MBps network bandwidth.

- *Offline cost.* When no work is deferred to an offline phase, ObliVM achieves a garbling speed of $670K$ gates/sec consuming 19MBps network bandwidth.

Slowdown relative to a non-secure baseline. For completeness, we now describe ObliVM's slowdown in comparison with a non-secure baseline where computation is performed in cleartext. As shown in Table 5.4, our slowdown relative to a non-secure baseline is application dependent, and ranges from $45\times$ to $9.3 \times 10^6\times$. We also present the *estimated* slowdown if a JustGarble-like back end is used for ObliVM-generated circuits. These numbers are estimated based on our circuit sizes

Task	Cleartext	ObliVM		ObliVM+JustGB (estimate)	
	Time	Runtime	Slowdown	Runtime	Slowdown
K-Means (Online)	0.4ms	24min	3.6×10^6	1.9min	2.9×10^5
K-Means (Total)	0.4ms	62min	9.3×10^6	4.58min	6.9×10^5
Distributed GWAS (Online)	40ms	1.8s	45	0.14s	3.5
Distributed GWAS (Total)	40ms	5.2s	130	0.28s	7
Binary Search (Online)	10μs	1.3s	1.3×10^5	78.8ms	7.9×10^3
Binary Search (Total)	10μs	7.4s	7.4×10^5	189ms	1.9×10^4
AMS Sketch (Online)	80μs	9.5s	1.2×10^5	0.5s	6.3×10^3
AMS Sketch (Total)	80μs	36.8s	4.6×10^5	1.2s	1.5×10^4
Hamming (Online)	0.3μs	1.71ms	6×10^3	0.16ms	5.3×10^2
Hamming (Total)	0.3μs	5.07ms	1.7×10^4	0.39ms	1.3×10^3

Table 5.4: **Slowdown of secure computation compared with non-secure, cleartext computation.** Parameter choices are the same as Table 5.3. Online cost only includes operations that are input-dependent. All time measurements assume data are pre-loaded to the memory. ObliVM requires a bandwidth of 19MBps. Numbers for JustGarble are estimated using ObliVM-generated circuit sizes assuming 315MBps bandwidth.

as well as the reported 11M AND gates/sec performance metric reported by JustGarble [11].

In particular, we elaborate on the following interesting cases. First, the distributed genome-wide association study(GWAS) application is Task 1 in the iDash secure genomic analysis competition [1], with total data size 380KB. This task achieves a small slowdown, because part of the computation is done locally – specifically, Alice and Bob each performs some local preprocessing to obtain the alle frequencies of their own data, before engaging in a secure computation protocol to compute χ^2-statistics. For details, we refer the reader to our online short note on how we implemented the competition tasks. On the other hand, benchmarks with floating point operations such as K-Means incur a relatively larger slowdown because modern processors have special floating point instructions which makes it

favorable to the insecure baseline.

5.6 Conclusion

We design ObliVM, a programming framework for automated secure computation. Additional examples can be found at our project website http://www.oblivm.com, including popular streaming algorithms, graph algorithms, data structures, machine learning algorithms, secure genome analysis [1], etc.

Chapter 6: Conclusion Remarks and Future Directions

6.1 Summary

In this thesis, we investigate in a set of cloud-related security applications in which programs' execution traces may leak information. We propose principled approaches to achieve performant *trace oblivious program execution*. In particular, we exploit the intrinsic obliviousness within each program, so that expensive cryptographic ORAM constructions and their overheads can be saved. Security type systems are developed to enforce that our optimization does not violate privacy and security requirements in the application domains.

Based on these principled methods, we build GhostRider, a hardware-software co-designed system, as a hardware-based solution, and ObliVM, a RAM-model secure computation framework, as a cryptography-based solution to mitigate attacks from cloud's insiders and intruders. Both systems demonstrate superior improvements over previous ones by orders of magnitudes.

6.2 Future Direction

While this thesis greatly expanded the study in trace oblivious execution, several future directions are promising.

6.2.1 Verifying Hardware ORAM Implementation

While we have demonstrated that ObliVM's type system can help indentifying bugs in Circuit ORAM implementations, it remains an interesting question whether the obliviousness of ORAM algorithms can be verified automatically. An particularly interesting and important direction is to verify a hardware ORAM controller, such as in GhostRider, is implemented secure. Several challenges may arise during this investigation. First, no static analysis-based approach has been developed to verify the obliviousness of ORAM algorithms. The main challenge is to enforce random numbes are handled correctly. Second, there is a gap between the languages that we have been studying and popular hardware programming languages, such as Verilog. This gap may introduce more technical difficulties to design such a verifier.

6.2.2 Parallel Trace Oblivious Execution

So far, we have focused on sequential programs. While parallel programs become the new main fashion in applications such as big data and deep learning, it is interesting and important to study how to achieve trace obliviousness for parallel programs. This is not easy. In particular, the design of concurrent ORAM algorithms are still in its theoretical phase, and most existing concurrent ORAM

algorithms are not practically. Therefore, syntactic hints from the executed program to exploit its obliviousness are more promising than relying on parallel ORAM algorithms. In fact, many big-data programming frameworks, such as MapReduce [25], already force programmers to express their computations into mappers and reducers, which are both able to be executed in parallel without leaking any information through the execution traces. GraphSC [70] has made the first attempt to adopt this idea to present a set of programming interfaces so that programs using these interfaces can be turned into there parallel oblivious version automatically without incurring too much overhead.

6.2.3 Differentially Privately Oblivious Execution

In this thesis, we focus on programs that leak absolutely no information through their execution traces. Most practical programs, however, do not have a counterpart satisfying this property: it is very easy for the program to include a loop whose guard depends on some secret data. Therefore, it is interesting to seek for a weaker version of trace obliviousness.

Privacy researches in recent ten years advocate for weaker privacy notions such as *differential privacy* to be enforced in real applications. Therefore, it is interesting to study whether trace obliviousness has a differentially private counterpart. If related techniques can be developed, trace oblivious program execution will be more practical in such applications where absolute trace obliviousness is unnecessary but differential privacy is sufficient.

Appendix A: Proof of Theorem 1

A.1 Trace equivalence and lemmas

We shall further study some properties of trace equivalence. First of all, we define the length of a trace t, denoted as $|t|$ to be:

$$|t| = \begin{cases} 1 & \text{if } t = \mathbf{read}(x,n) \mid \mathbf{write}(x,n) \mid \mathbf{readarr}(x,n,n') \mid \\ & \quad \mathbf{writearr}(x,n,n') \\ 0 & \text{if } t = \epsilon \\ |t_1| + |t_2| & \text{if } t = t_1 @ t_2 \end{cases} \tag{A.1}$$

Lemma 1. *If $t_1 \equiv t_2$, then $|t_1| = |t_2|$.*

Proof. Let us prove by induction on how $t_1 \equiv t_2$ is derived. If $t_1 = t_2$, then the conclusion is obvious. If $t_1 = \epsilon @ t_2$, then $|t_1| = |\epsilon| + |t_2| = |t_2|$. Similarly, we can prove the conclusion when $t_1 = t_2 @ \epsilon$, $t_2 = \epsilon @ t_1$, $t_2 = t_1 @ \epsilon$, or $t_1 = \epsilon @ t$ and $t_2 = t @ \epsilon$.

If $t_1 = t_{11} @ t_{12}$, $t_2 = t_{21} @ t_{22}$, $t_{11} \equiv t_{21}$, and $t_{12} \equiv t_{22}$, then by induction, we have $|t_{11}| = |t_{21}|$, and $|t_{12}| = |t_{22}|$. Therefore $|t_1| = |t_{11}| + |t_{12}| = |t_{21}| + |t_{22}| = |t_2|$.

Finally, if $t_1 = (t'_1 @ t'_2) @ t'_3$ and $t_2 = t'_1 @ (t'_2 @ t'_3)$, then $|t_1| = |t'_1 @ t'_2| + |t'_3| = |t'_1| + |t'_2| + |t'_3| = |t'_1| + |t'_2 @ t'_3| = |t_2|$. □

Now, we define the i-th element in a trace, denoted $t[i]$, as follows:

$$t[i] = \begin{cases} \epsilon & \text{if } i \leq 0 \vee i > |t| \\ t & \text{if } i = 1 \wedge t = \mathbf{read}(x, n) \mid \mathbf{write}(x, n) \mid \\ & \quad \mathbf{readarr}(x, n, n') \mid \mathbf{writearr}(x, n, n') \\ t_1[i] & \text{if } t = t_1@t_2 \vee 1 \leq i \leq |t_1| \\ t_2[i - |t_1|] & \text{if } t = t_1@t_2 \vee |t_1| < i \leq |t| \end{cases}$$

It is easy to see that if $\forall i.t_1[i] = t_2[i]$ implies $|t_1| = |t_2|$ by the following lemma.

Lemma 2. *$t[i] \neq \epsilon$ for all i such that $1 \leq i \leq |t|$, and ϵ otherwise.*

Proof. The second part of the conclusion is trivial since it directly follows the definition. We prove the first part by induction on $|t|$. If $|t \models 0$, then the conclusion is trivial.

If $|t| = 1$, and $1 \leq i \leq |t|$, then i must be 1. Therefore, $t[i]$ is one of $\mathbf{read}(x, n$, $\mathbf{write}(x, n)$, $\mathbf{readarr}(x, n, n')$, and $\mathbf{writearr}(x, n, n')$, and therefore $t[i] \neq \epsilon$.

If $|t| > 1$, then t must be a concatenation of two subsequences, i.e. $t_1@t_2$. If $1 \leq i \leq |t_1|$, then $t[i] = t_1[i]$, and by induction, we know that $t[i] \neq \epsilon$. Otherwise, if $|t_1| < i \leq |t|$, then $0 < i - |t_1| \leq |t| - |t_1| = |t_2|$. For natural number n, $n > 0$ implies $n \geq 1$. Therefore $1 \leq i - |t_1| \leq |t_2|$, and by induction, we have $t[i] = t_2[i - |t_1|] \neq \epsilon$. □

Before we go to the next lemma, we shall define the canonical representation

of a trace. First, we define the number of blocks in a trace t, denoted by $\#(t)$, as

$$\#(t) = \begin{cases} \#(t_1) + \#(t_2) & \text{if } t = t_1@t_2 \\ 1 & \text{otherwise} \end{cases}$$

Then we define an order $\preceq_t$ between two traces t_1 and t_2 as follows: $t_1 \preceq_t t_2$ if and only if either of the following two conditions hold true: (i) $\#(t_1) < \#(t_2)$, or (ii) $\#(t_1) = \#(t_2) \geq 2$, $t_1 = t_1'@t_1''$, $t_2 = t_2'@t_2''$, and either of the following three sub-conditions holds true: (ii.a) $\#(t_1') > \#(t_2')$; (ii.b) $\#(t_1') = \#(t_2')$ and $t_1' \preceq_t t_2'$; or (ii.c) $t_1' = t_2'$ and $t_1'' \preceq t_2''$. It is easy to see that $\preceq_t$ is complete.

Definition 10 (canonical representation)**.** *The canonical representation of a trace t is the minimal element in the set $\{t' : t \equiv t'\}$ under order $\preceq_t$.*

Lemma 3. *The canonical representation of t is (i) ϵ is $|t \models 0$; or (ii) $can(t) = (...((t_1@t_2)@t_3)...@t_n)$, where $n = |t| > 0$, and $t_i = t[i]$.*

Proof. On the one hand, it is easy to see that $can(t)$ belongs to the set $\{t' : t \equiv t'\}$. In fact, we can prove by induction on $\#(t)$. If $\#(t) = 1$, then either $|t \models 1$, or $|t \models 0$. For the former case, t is one of **read**$(x, n$, **write**(x, n), **fetch**(p), **readarr**(x, n, n'), and **writearr**(x, n, n'), and thus $t = t[1] = can(t)$. For the later case, $t = \epsilon$.

Now suppose $\#(t) > 1$, and thus $t = t'@t''$. Suppose $|t' \models l_1$ and $|t'' \models l_2$. If $l_2 = 0$, by induction, $t'' = \epsilon$, and thus $t \equiv t'$. Furthermore, we have $|t| = |t'|$, and $\forall i.t[i] = t'[i]$ by definition. Therefore $t \equiv t' \equiv can(t') = can(t)$. Similarly, we can prove the conclusion is true when $l_1 = 0$. Now suppose $l_1 > 0$ and $l_2 > 0$, then $can(t') = (...((t_1@t_2)@t_3)...t_{l_1})$, and $can(t'') = (...((t_{l_1+1}@t_{l_1+2})@t_{l_1+3})...@t_{l_1+l_2})$.

Then $t \equiv can(t')@can(t'') \equiv can(t)$.

On the other hand, we shall show that $can(t)$ is the minimal one in $\{t' : t \equiv t'\}$. To show this point, we only need to show that for all $t' \equiv can(t)$, we have $can(t) \preceq_t t'$. We prove by induction on n. If $n = 1$, the conclusion is obvious. Suppose $n > 1$ and the conclusion holds true for all $n' < n$.

It is easy to see that $\#(t') > 1$, therefore we suppose $t' = t_l@t_r$. Then we prove that there exists k such that $t_l \equiv (...(t_1@t_2)...@t_k)$ and $t_r \equiv (...(t_{k+1}@t_{k+2})...@t_n)$. We prove by induction on n and how many steps of equivalent-transitive rule, i.e., $t_1 \equiv t_2 \wedge t_2 \equiv t_3 \Rightarrow t_1 \wedge t_3$, should be applied to derive $can(t) \equiv t'$. If we should apply 0 step, then we know one of the following situations holds: (i) $t' = t''@t_n$ where $t'' \equiv (...(t_1@t_2)...@t_{n-1})$; (ii) $t' = (...(t_1@t_2)...t_{n-2})@(t_{n-1}@t_n)$; (iii) $t' = t@\epsilon$; or (iv) $t' = \epsilon@t$. In any case, our conclusion holds true. Now suppose we need to apply $n > 0$ steps to derive t', where in the $n-1$ step, we derive that $can(t) \equiv t''$ and we can derive $t'' \equiv t'$ without applying the equivalent-transitive rule. Therefore by induction, we know that $t'' = t''_l@t''_r$, and there is k such that $t''_l \equiv (...(t_1@t_2)...@t_k)$ and $t''_r \equiv (...(t_{k+1}@t_{k+2})...@t_n)$. Since we can derive $t'' \equiv t'$ without applying equivalent-transitive rule, we know that one of the following situations holds:

1. $t''_l \equiv t_l$ and $t''_r \equiv t_r$;
2. $t' = \epsilon@t''$;
3. $t' = t''@\epsilon$;
4. $t'' = t'@\epsilon$;

5. $t'' = \epsilon @ t'$;

6. $\epsilon @ t' = t'' @ \epsilon$;

7. $t' @ \epsilon = \epsilon @ t''$;

8. $t''_l = (t_{l1} @ t_{l2})$, $t_l \equiv t_{l1}$, and $t_r \equiv t_{l2} @ t''_r$ (in this case, we have $(t_{l1} @ t_{l2}) @ t''_r \equiv t_{l1} @ (t_{l2} @ t''_r)$); and

9. $t''_r = (t_{r1} @ t_{r2})$, $t_l \equiv t''_l @ t_{r1}$, and $t_r \equiv t_{r2}$ (in this case, we have $t''_l @ (t_{r1} @ t_{r2}) \equiv (t''_l @ t_{r1}) @ t_{r2}$).

For the first 7 cases, the conclusion is trivial. For Case 8, by induction, we know there are some k' such that $t_{l1} \equiv (...(t_1 @ t_2)...@ t_{k'})$ and $t_{l2} \equiv (...(t_{k'+1} @ t_{k'+2})...@ t_k)$. Therefore $t_l \equiv (...(t_1 @ t_2)...@ t_{k'})$, and $t_r \equiv (...(t_{k'+1} @ t_{k'+2})...@ t_k) @ (...(t_{k+1} @ t_{k+2})...@ t_n)$ while the later is equivalent to $(...(t_{k'+1} @ t_{k'+2})...@ t_n)$. Similarly, we can prove under case 9, the conclusion is also true.

Next, we prove that $can(t) \preceq_t t_l @ t_r$. To show this point, by induction, we know $(...(t_1 @ t_2)...@ t_k) \preceq t_l$ and $(...(t_{k+1} @ t_{k+2})...@ t_n) \preceq t_r$. If either $\#(t_l) > k$ or $\#(t_r) > n - k$, we have $\#(t') > n$ and thus $can(t) \preceq t'$. Suppose $\#(t_l) = k$ and $\#(t_r) = n - k$. If $k < n - 1$, then by the definition of $\preceq$, we have $can(t) \preceq_t t'$. Next suppose $k = n - 1$, then by induction, we know $(...(t_1 @ t_2)...@ t_{n-1}) \preceq_t t_l$, and thus $can(t) \preceq_t t_l @ t_r = t'$. □

Next is the most important lemma about trace-equivalence.

Lemma 4. *$t_1 \equiv t_2$, if and only if $\forall i. t_1[i] = t_2[i]$.*

Proof. "⇒" Suppose $\forall i.t_1[i] = t_2[i]$. Then by Lemma 3, we know $can(t_1) = can(t_2)$, and thus $t_1 \equiv can(t_1) \equiv can(t_2) \equiv t_2$.

"⇐" Suppose $t_1 \equiv t_2$, then by Lemma 3, we have $can(t_1) \equiv can(t_2)$. Due to both $can(t_1)$ and $can(t_2)$ have the same form, we know they are identical. Therefore, we can conclude that $\forall i.t_1[i] = t_2[i]$. □

A.2 Lemmas on trace pattern equivalence

Trace pattern equivalence has similar properties as trace equivalence. If fact, we define the length of a trace pattern T, denoted as $|T|$, to be

$$|T| = \begin{cases} 1 & \text{if } T = \mathbf{Read}(x) \mid \mathbf{Fetch}(p) \\ 0 & \text{if } T = \epsilon \\ |T_1| + |T_2| & \text{if } T = T_1 @ T_2 \end{cases}$$

Similar to trace, we define the i-th element in a trace pattern T, denoted $T[i]$, as follows:

$$T[i] = \begin{cases} \epsilon & \text{if } i \leq 0 \vee i > |T| \\ T & \text{if } i = 1 \wedge T = \mathbf{Read}(x) \\ T_1[i] & \text{if } T = T_1 @ T_2 \vee 1 \leq i \leq |T_1| \\ T_2[i - |T_1|] & \text{if } T = T_1 @ T_2 \vee |T_1| < i \leq |T| \end{cases}$$

Using exactly the same technique, we can prove the following lemma:

Lemma 5. *$T_1 \sim_L T_2$, if and only if $\forall i.T_1[i] = T_2[i]$.*

To avoid verbosity, we do not provide the full proof here. It is quite similar to

the proof of Lemma 4

A.3 Proof of memory trace obliviousness

To prove Theorem 1, *memory trace obliviousness by typing*, we shall first prove the following lemma:

Lemma 6. *If* $\Gamma \vdash e : \mathit{Nat}\ L; T$*, then for any two* Γ*-valid low-equivalent memories* M_1*,* M_2*, if* $\langle M_1, e\rangle \Downarrow_{t_1} n_1$*,* $\langle M_2, e\rangle \Downarrow_{t_2} n_2$*, then* $t_1 = t_2$ *and* $n_1 = n_2$

Proof. We use structural induction on expression e to prove this lemma. If e is in form of x, then $\Gamma(x) = \text{Nat L}$, and thus $M_1(x) = M_2(x) = n$ according to the definition of low-equivalence and Γ-validity. Therefore $t_1 = \mathbf{read}(x, n) = t_2$, and $n_1 = n_2 = n$.

If e is in form of $e_1\ op\ e_2$, then $\Gamma \vdash e_1 : \text{Nat L}$ and $\Gamma \vdash e_2 : \text{Nat L}$. Suppose $\langle M_i, e_j\rangle \Downarrow_{t_{ij}} n'_{ij}$, for $i = 1, 2$, $j = 1, 2$. Then $t_{1j} = t_{2j}$ and $n_{1j} = n_{2j}$ for $j = 1, 2$. Therefore $t_1 = t_{11}@t_{12} = t_{21}@t_{22} = t_2$, and $n_1 = n_{11}\ op\ n_{12} = n_{21}\ op\ n_{22} = n_2$.

Next, we consider the expression in form of $x[e]$. We know that $\Gamma(x) = \text{Array L}$, which implies $\Gamma \vdash e : \text{Nat L}$. Suppose $\langle M_i, e\rangle \Downarrow_{t'_i} n'_i$, then by induction $t'_1 = t'_2$ and $n'_1 = n'_2$. Furthermore, since $M_1 \sim_L M_2$, we have $\forall i \in \mathbf{Nat}.M_1(x)(i) = M_2(x)(i)$. Therefore $t_1 = t'_1@\mathbf{readarr}(x, n'_1, M_1(x)(n'_1)) = t'_2@\mathbf{readarr}(x, n'_2, M_2(x)(n'_2)) = t_2$, and $n_1 = M_1(x)(n'_1) = M_2(x)(n'_2) = n_2$.

Finally, the conclusion is trivial for constant expression. □

For convenience, we define $lab : \mathbf{Type} \to \mathbf{SecLabels}$ as:

$$lab(\tau) = \begin{cases} l & \text{if } \tau = \text{Int } l \\ l & \text{if } \tau = \text{Array } l \end{cases}$$

Similar to Lemma 6, we can prove the following lemma:

Lemma 7. *If* $\Gamma \vdash e : Nat\ l; T$ *and* $l \in$ ***ORAMBanks****, then for any two* Γ*-valid low-equivalent memories* M_1*,* M_2*, if* $\langle M_1, e\rangle \Downarrow_{t_1} n_1$*,* $\langle M_2, e\rangle \Downarrow_{t_2} n_2$*, then* $t_1 = t_2$

Proof. If $l = \text{L}$, then the conclusion is obvious by Lemma **??**. We only consider $l \in$ **ORAMBanks**. We use structural induction to prove this lemma. If e is in form of x, then according to the definition of Γ-validity and $evt(\cdot)$, we have $t_1 = lab(\Gamma(x)) = t_2$.

If e is in form of $e_1\ op\ e_2$, then $\Gamma \vdash e_1 : \text{Nat } l_1$ and $\Gamma \vdash e_2 : \text{Nat } l_2$. Suppose $\langle M_i, e_j\rangle \Downarrow_{t_{ij}} n'_{ij}$, for $i = 1, 2$, $j = 1, 2$. Then $t_{1j} = t_{2j}$, for $j = 1, 2$ by induction. Therefore $t_1 = t_{11}@t_{12} = t_{21}@t_{22} = t_2$.

Finally, we consider the expression in form of $x[e]$. We know that $\Gamma \vdash e : \text{Nat } l'$. Suppose $\langle M_i, e\rangle \Downarrow_{t'_i} n'_i$. If $l' = L$, then $t'_1 = t'_2$ by Lemma **??**. Otherwise, $l \in$ **ORAMBanks**, and by induction assumption, we have $t'_1 = t'_2$. Since $l \in$ **ORAMBanks**, we know $l = lab(\Gamma(x))$, and thus $t_1 = t'_1@l = t'_2@l = t_2$. □

Now we shall study the property of trace pattern equivalence. First of all, we have the following lemma:

Lemma 8. *Suppose* s *and* S *are a statement and a labeled statement respectively. If* $\Gamma, l_0 \vdash S; T$*,* $l_0 \in$ ***ORAMBanks*** *and* $\langle M, S\rangle \Downarrow_t M'$*, then* $M \sim_L M'$*.*

Proof. We prove by induction on the statement S. Notice that the statement is impossible to be **while** statement. The conclusion is trivial for the statement **skip**.

If s is $x := e$, then $l_0 \subseteq lab(\Gamma(x))$, and thus $lab(\Gamma(x)) \in$ **ORAMBanks**. Therefore $M' = M[x \mapsto (n, l)]$ for some natural number n and some security label l, which implies $M' \sim_L M$. Similarly, if s is $x[e_1] := e_2$, then $lab(\Gamma(x)) \in$ **ORAMBanks**. Furthermore, $\langle M, x[e_1] := e_2\rangle \Downarrow_t M[x \mapsto (m, l)]$ for some mapping m, and some security label $l \in$ **ORAMBanks**. Therefore $M' = M[x \mapsto (m, l)]$, which implies for x such that $M(x) = (n, \mathrm{L})$, we know that $M'(x) = (n, \mathrm{L})$. Therefore $M' \sim_L M$.

Next, let us consider statement **if**(e, S_1, S_2). Then we know either of the two conditions holds true: (1) $\langle M, S_1\rangle \rightarrow M'$, and (2) $\langle M, S_2\rangle \rightarrow M'$. Since $\Gamma, l_0 \vdash$ **if**$(e, S_1, S_2); T$, we have $\Gamma, l' \vdash S_1; T_1$, and $\Gamma, l' \vdash S_2; T_2$, where $l_0 \sqsubseteq l'$. Therefore we know for either condition, we have $M \sim_L M'$.

Finally, for sequence of two statements $S_1; S_2$, suppose $\langle M, S_1\rangle \Downarrow_t M_1$, and $\langle M_1, S_2\rangle \Downarrow_{t'} M'$. Then $M \sim_L M_1 \sim_L M'$. □

According to definition of the trace pattern equivalence, it is obvious to see that, if $T \sim_L T'$, then T is a sequence, whose element each is in the form of **Fetch**(p), **Read**(x), ϵ, and o.

We shall define *a trace t belongs to a trace pattern T, under a memory M, denoted by $t \in T[M]$* as follows:

$$\epsilon \in \epsilon[M] \qquad o \in o[M] \qquad \frac{M(x) = (n, \mathrm{L}), n \in \mathbf{Nat}}{\mathbf{read}(x, n) \in \mathbf{Read}(x)[M]}$$

$$\frac{t_1 \in T_1[M] \quad t_2 \in T_2[M]}{t_1@t_2 \in T_1@T_2[M]} \qquad \frac{t \in T[M] \quad T \sim_L T'}{t \in T'[M]}$$

Now, we prove the most important lemma for $t \in T[M]$:

Lemma 9. *$t \in T[M]$ if and only if $|t| = |T|$ and $\forall i.t[i] \in (T[i])[M]$.*

Proof. "$\Rightarrow$" Suppose $|t| = |T|$ and $\forall i.t[i] \in (T[i])[M]$. We prove by induction on $\#(t)$. If $\#(t) = 1$, then the conclusion is trivial. Assume the conclusion holds for all $\#(t') < n$, now suppose $\#(t) = n > 1$. Then we know $t = t_1@t_2$. If $t_1 = \epsilon$, then we know $|t_2| = |t| = |T|$ and $\forall i.t_2[i] = t[i] \in (T[i])[M]$, by induction, we know $t_2 \in T[M]$. Furthermore, we have $t_1 = \epsilon \in \epsilon[M]$, therefore $t_1@t_2 \in \epsilon@T[M]$. Since $\epsilon@T \sim_L T$, we have $t = t_1@t_2 \in T[M]$. A similar argument shows that if $t_2 = \epsilon$, then we also have $t \in T[M]$.

Now let us consider when $|t_1 \models 0$. By induction, we have $t_1 \in \epsilon[M]$ and $t_2 \in T[M]$, and then again, we have $t \in T[M]$. Similarly, if $|t_2 \models 0$, we also have $t \in T[M]$.

Now assume $|t_1| > 0$ and $|t_2| > 0$, and suppose $T_1 = (...(T_1@T_2)...@T_{|t_1|})$ and $T_2 = (...(T_{|t_1|+1}@T_{|t_1|+2})...@T_{|T|})$. Then by induction, we know that $t_1 \in T_1[M]$ and $t_2 \in T_2[M]$, and thus $t_1@t_2 \in T_1@T_2[M]$. According to Lemma 5, we have $T_1@T_2 \sim_L T$, and thus $t = t_1@t_2 \in T[M]$.

"$\Leftarrow$" We prove by induction on how many steps to derive $t \in T[M]$. Suppose we need only 1 step, then one of the following four conditions is true: (i) $t = \epsilon = T$;

(ii) $t = o = T$; (iii) $t = \mathbf{read}(x, n)$, $T = \mathbf{Read}(x)$ and $M(x) = n$. In either case, the conclusion is trivial.

Then suppose we need n step, and the last step is derived from $t = t_1@t_2$, $T = T_1@T_2$, and $t_1 \in T_2[M]$ and $t_2 \in T_2[M]$. Then by induction we have $|t_1| = |T_1|$, $|t_2| = |T_2|$, $\forall i.t_1[i] \in (T_1[i])[M]$, and $\forall i.t_2[i] \in (T_2[i])[M]$. For $i < 1$ or $i > |T|$, then $t[i] = \epsilon = T[i]$, and thus $t[i] \in (T[i])[M]$. If $1 \leq i \leq |T_1|$, then $t[i] = t_1[i]$ and $T[i] = T_1[i]$, and by induction, we have $t[i] \in (T[i])[M]$; if $|T_1| < i \leq |T|$, then $t[i] = t_2[i - |t_1|]$ and $T[i] = T_2[i - |T_1|]$, and by induction, we have $t[i] \in (T[i])[M]$.

Finally, suppose we need n step, and the last step is derived from $t \in T'[M]$ and $T' \sim_L T$. Then according to Lemma 5, we know that $\forall i.T'[i] = T[i]$, which also implies that $|T'| = |T|$. By induction, we have $|t| = |T'|$ and $\forall i.t[i] \in (T'[i])[M]$, and therefore, we have $\forall i.t[i] \in (T[i])[M]$ and $|t| = |T|$. □

We have the following corollaries.

Corollary 1. *If $M_1 \sim_L M_2$, and $t \in T[M_1]$, then $t \in T[M_2]$.*

Proof. By Lemma 9, we only need to show that $\forall i.t[i] \in (T[i])[M_2]$.

Let us prove by structural induction on how $t \in T[M]$ is derived. If $t = \epsilon = T$, or $t = o = T$, or $t = t_1@t_2$ and $T = T_1@T_2$, then the conclusion is trivial. The only condition we need to prove is when $t = \mathbf{read}(x, n)$, and $T = \mathbf{Read}(x)$. If so, since $t \in T[M_1]$, therefore $M_1(x) = (n, \mathrm{L})$. Since $M_1 \sim_L M_2$, we know that $M_2(x) = (n, \mathrm{L})$. Therefore, we have $t = \mathbf{read}(x, n) \in \mathbf{Read}(x)[M_2] = T[M_2]$.

According to the definition of $T[i]$, we know it is in one of the following three forms: ϵ, o, or **Read**. If $T[i] = \epsilon$, then we know $i < 1$ or $i > |T| = |t_1|$. Therefore

$t[i] = \epsilon$, and thus $t[i] \in (T[i])[M_2]$. If $T[i] = o$, then we know $t[i] = o$. In both situations, we have $t[i] \in (T[i])[M_2]$. Finally, if $T[i] = \mathbf{Read}(x)$, then we know $t[i] = \mathbf{read}(x, n)$ where $n = M_1[x]$. Since $M_1 \sim_L M_2$, we have $M_2[x] = n$, and thus $t[i] \in (T[i])[M_2]$. □

Corollary 2. *If $t_1 \in T[M]$ and $t_2 \in T[M]$, then $t_1 \equiv t_2$.*

Proof. Assume $t_1 \in T[M]$, and $t_2 \in T[M]$, according to Lemma 9, we have $|t_1| = |T| = |t_2|$, $\forall i.t_1[i] \in (T[i])[M]$, and $\forall i.t_2[i] \in (T[i])[M]$. According to the definition of $T[i]$, we know it is in one of the following three forms: ϵ, o, or **Read**. If $T[i] = \epsilon$, then we know $i < 1$ or $i > |T| = |t_1| = |t_2|$. Therefore $t_1[i] = t_2[i] = \epsilon$. If $T[i] = o$, then we know $t_1[i] = t_2[i] = o$. Finally, if $T[i] = \mathbf{Read}(x)$, then we know $t_1[i] = \mathbf{read}(x, n_1)$, $n_1 = M[x]$, $t_2[i] = \mathbf{read}(x, n_2)$, and $n_2 = M[x]$. Therefore $n_1 = n_2$, and thus $t_1[i] = t_2[i]$. Therefore $\forall i.t_1[i] = t_2[i]$, and according to Lemma 4, we have $t_1 \equiv t_2$. □

Then we have the following lemmas:

Lemma 10. *Suppose $\Gamma \vdash e : \tau; T$, $T \sim_L T'$ for some T', and memory M is Γ-valid. If $\langle M, e\rangle \Downarrow_t n$, then $t \in T[M]$.*

Proof. We prove by structural induction on e. If e is n, then $T = \epsilon = t$.

If e is x, then $T = evt(lab(\Gamma(x)), \mathbf{Read}(x))$. If $lab(\Gamma(x)) = l \in \mathbf{ORAMBanks}$, then $t = l \in l[M]$. If $lab(\Gamma(x)) = \mathrm{L}$, then $T = \mathbf{Read}(x)$, and $t = \mathbf{read}(x, n)$, where $M(x) = (n, \mathrm{L})$. According to the definition, we know $t \in T[M]$.

If e is e_1 *op* e_2, then suppose $\langle M, e_i\rangle \Downarrow_{t_i} n_i$ and $\Gamma \vdash e_i : l_i; T_i$ for $i = 1, 2$.

Then according to the induction assumption, we have $t_i \in T_i[M]$ for $i = 1, 2$. Since $t = t_1@t_2$, and $T = T_1@T_2$, we know $t \in T[M]$.

Next we consider $x[e']$. Suppose $\Gamma \vdash e' : \text{Nat } l'; T'$, and $\langle M, e' \rangle \Downarrow_{t'} n'$, then $T = T'@evt(lab(\Gamma(x)), \mathbf{Readarr}(x))$, and $t = t'@evt(lab(\Gamma(x)), \mathbf{readarr}(x, n', n''))$ for some n''. Moreover, we have $t' \in T'[M]$ by induction. Since $T \sim_L T'$, we know $lab(\Gamma(x)) \in \mathbf{ORAMBanks}$. Therefore $t = t'@lab(\Gamma(x)) \in T'@lab(\Gamma(x))[M] = T[M]$. □

Lemma 11. *Assume $\Gamma, l_0 \vdash S; T$, $T \sim_L T'$ for some T', and $l_0 \in$ **ORAMBanks**, and M is a Γ-valid memory. If $\langle M, S \rangle \Downarrow_t M'$, then $t \in T[M]$.*

Proof. We prove by structural induction on the statement S. Since $l_0 \neq \text{L}$, therefore we know S cannot be a **while** statement. If S is **skip**, then $T = \epsilon = t$.

Let us consider when S is $x := e$. Then $\langle M, e \rangle \Downarrow_{t'} n'$, and $\Gamma \vdash e : \tau; T'$, and $T = T'@evt(lab(\Gamma(x)), \mathbf{Write}(x))$. Since $T \sim_L T$, T does not contain $\mathbf{Write}(x)$, and thus $lab(\Gamma(x)) \in \mathbf{ORAMBanks}$. Therefore $t = t'@lab(\Gamma(x)) \in T'@lab(\Gamma(x))[M]$ by Lemma 10.

Next, suppose S is $x[e_1] = e_2$. Suppose $\langle M, e_i \rangle \Downarrow_{t_i} n_i$, and $\Gamma \vdash e_i : \tau; T_i$ for $i = 1, 2$ by induction. Then $t_i \in T_i[M]$ for $i = 1, 2$. Similar to the discussion for $x := e$, we know $lab(\Gamma(x)) \in \mathbf{ORAMBanks}$, and thus $t = t_1@t_2@lab(\Gamma(x)) \in T_1@T_2@lab(\Gamma(x))[M]$.

Next, let us consider $(if)(e, S_1, S_2)$. Then $\Gamma, l_0 \vdash S_i; T_i$ for $i = 1, 2$, and $T_1 \sim_L T_2$. As well $\Gamma \vdash e : \tau; T_e$, $\langle M, e \rangle \Downarrow_{t_e} n_e$, $idx = ite(n_e, 1, 2)$, and $\langle M, S_{idx} \rangle \Downarrow_{t_{idx}} M'$. Then $T = T_e@T_1$, and $t_e \in T_e[M]$. If $idx = 1$, then $\langle M, S_1 \rangle_{t_1} M'$, and thus

$t_1 \in T_1[M]$. Therefore $t = t_e@t_1 \in T_e@T_1[M] = T[M]$. Similarly, if $idx = 2$, then $\langle M, S_2\rangle_{t_2} M'$, and thus $t_2 \in T_2[M]$. Therefore $t_2 \in T_1[M]$. As a conclusion $t = t_e@t_2 \in T_e@T_1[M] = T[M]$.

Finally, suppose S is $S_1; S_2$. Then we know $\Gamma, l_0 \vdash S_i; T_i$ for $i = 1, 2$, $\langle M, S_1\rangle \Downarrow_{t_1} M'$, and $\langle M', S_2\rangle \Downarrow_{t_2} M''$. Since $l_0 \in \mathbf{ORAMBanks}$, we know $M \sim_L M' \sim_L M''$. By induction assumption, we know $t_1 \in T_1[M]$, and $t_2 \in T_2[M']$. Since $M \sim_L M'$, according to Corollary 1, we know $t_2 \in T_2[M]$. Therefore $t = t_1@t_2 \in T_1@T_2[M] = T[M]$. □

Lemma 12. *Suppose* $\Gamma, l_0 \vdash S_i; T_i$, *for* $i = 1, 2$, *where* $l_0 \in \boldsymbol{ORAMBanks}$, *and* $T_1 \sim_L T_2$. *Given two* Γ*-valid low-equivalent memories* M_1, M_2, *if* $\langle M_1, S_1\rangle \Downarrow_{t_1} M_1'$, *and* $\langle M_2, S_2\rangle \Downarrow_{t_2} M_2'$, *then* $M_1' \sim_L M_2'$, *and* $t_1 \equiv t_2$.

Proof. According to Lemma 11, we know that $t_i \in T_i[M_i]$ for $i = 1, 2$. According to Lemma 8, we know that $M_1' \sim_L M_1$ and $M_2' \sim_L M_2$. Since $M_1 \sim_L M_2$, we know that $M_1' \sim_L M_1 \sim_L M_2 \sim_L M_2'$. Because $t_1 \in T_1[M_1]$, and $M_1 \sim_L M_2$, therefore $t_1 \in T_1[M_2]$. Furthermore, since $T_1 \sim_L T_2$, we have $t_1 \in T_2[M_2]$. Finally, since $t_2 \in T_2[M_2]$, and according to Corollary 2 we have $t_1 \equiv t_2$. □

Now we are ready to prove Theorem 1.

Proof of Theorem 1. We extend this conclusion by considering both normal statement and labeled statement, and shall prove by induction on the statement s. For notational convention, we suppose $\langle M_1, s\rangle \Downarrow_{t_1} M_1'$, and $\langle M_2, s\rangle \Downarrow_{t_2} M_2'$, and thus $\Gamma, l_0 \vdash S; T$ with $M_1 \sim_L M_2$ and both are Γ-valid. Our goal is prove $t_1 \equiv t_2$, and $M_1 \sim_L M_2$.

If s is **skip**, it is obvious.

Suppose s is $x := e$, then $\Gamma \vdash e : \text{Nat } l; T$. Suppose $\langle M_i, e\rangle \Downarrow_{t_i'} n_i$, for $i = 1, 2$. According to Lemma 6 and Lemma 7, we know $t_1' = t_2'$. If $lab(\Gamma(x)) \in$ **ORAMBanks**, then $M_1' = M_1[x \mapsto (n_1, l_1)] \sim_L M_1 \sim_L M_2 \sim_L M_2[x \mapsto (n_2, l_2)] = M_2'$, and $t_1 = t_1'@lab(\Gamma(x)) = t_2'@lab(\Gamma(x)) = t_2$, which implies $t_1 \equiv t_2$.

If $\Gamma(x) = \text{Nat L}$, then we know $\Gamma \vdash e : \text{Nat L}; T$, and according to Lemma 6, we have $n_1 = n_2$. Then we also have $M_1' = M_1[x \to n_1] \sim_L M_2[x \to n_2] = M_2'$, and $t_1 = t_1'@\mathbf{read}(x, n_1) \equiv t_2'@\mathbf{read}(x, n_2) = t_2$.

Next, suppose s is $x[e_1] := e_2$. Suppose $\langle M_i, e_j\rangle \Downarrow_{t_{ij}} n_{ij}$ for $i = 1, 2$, $j = 1, 2$. If $lab(\Gamma(x)) = \text{L}$, then we know $\Gamma \vdash e_j : \text{Nat L}$, for $j = 1, 2$. Then by Lemma 6, we have $t_{1j} = t_{2j}$, which implies $t_{1j} \equiv t_{2j}$, $n_{1j} = n_{2j}$, and according to the definition of Γ-validity and low-equivalence, $\forall i. M_1(x)(i) = M_2(x)(i)$. Therefore $t_1 = t_{11}@t_{21}@\mathbf{writearr}(x, n_{11}, n_{12}) \equiv t_{21}@t_{22}@\mathbf{writearr}(x, n_{21}, n_{22}) = t_2$, and $M_1' = M_1[x \to M_1(x)[n_{11} \to n_{12}]] \sim_L M_2[x \to M_2(x)[n_{21} \to n_{22}]] = M_2'$.

Otherwise, if $\Gamma(x) \in$ **ORAMBanks**, suppose $\Gamma \vdash e_i : \mathbf{Nat}\ l_i; T_i$, for $i = 1, 2$. Then we know $l_0 \sqcup l_1 \sqcup l_2 \sqsubseteq lab(\Gamma(x))$. Therefore, by Lemma 7, based on the same reasoning as above for Nat l case, we have $t_1 = t_{11}@t_{21}@\Gamma(x) \equiv t_{21}@t_{22}@\Gamma(x) = t_2$. Furthermore, $M_1' = M_1[x \to m_1] \sim_L M_1 \sim_L M_2 \sim_L M_2[x \to m_2] = M_2'$ for some two mappings, m_1 and m_2.

Then suppose the statement is $\mathbf{if}(e, S_1, S_2)$. There are two situations. If $\Gamma \vdash e : \text{Nat } l_e; T_e$, where $l_e \sqcup l_0 \in$ **ORAMBanks**, then according to Lemma 12, we know $M_1' \sim_L M_2'$, and $t_1 \equiv t_2$. Otherwise, we have $l_e = \text{L}$ and $l_0 = \text{L}$. Suppose $\langle M_i, e\rangle \Downarrow_{t_i'} n_i$, for $i = 1, 2$, then according to Lemma 6, we know $t_1' = t_2'$, which

implies $t_1' \equiv t_2'$, and $n_1 = n_2$. If $ite(n_1, 1, 2) = 1$, then we know $\langle M_1, S_1\rangle \Downarrow_{t_1''} M_1'$ and $\langle M_2, S_1\rangle \Downarrow_{t_2''} M_2'$. Therefore $t_1 = t_1'@t_1'' \equiv t_1'@t_2'' = t_2$, and $M_1' \sim_L M_2'$ by induction. We can show the conclusion for $ite(n_1, 1, 2) = 2$ similarly.

Next, let us consider the statement $\mathbf{while}(e, S)$. We know $\Gamma \vdash e : \text{Nat L}; T$, therefore there exists a constant n, and a trace t, such that $\langle M_i, e\rangle \Downarrow_t, n$ for both $i = 1, 2$, by Lemma 6.

We prove by induction on how many steps applying the S-WhileT rule and S-WhileF rule (WHILE rules for short) to derive $\langle \mathbf{while}(e, S), M_1\rangle \Downarrow_{t_1}\rangle$. If we only apply one time, then we must apply S-WhileF rule, and thus $n = 0$. Then we have $t_1 = t = t_2$, and $M_1' = M_1 \sim_L M_2 = M_2'$. Suppose the conclusion is true when we need to apply $n - 1$ steps of WHILE rules, now let us consider when we need to apply $n > 0$ steps. Then we know $n \neq 0$. Suppose $\langle M_i, S\rangle \Downarrow_{t_{i1}} M_{i1}$, and $\langle M_{i1}, \mathbf{while}(e, S)\rangle \Downarrow_{t_{i2}} M_i'$, for $i = 1, 2$. Then we know that we need to apply $n - 1$ steps of WHILE rules to derive $\langle M_{11}, \mathbf{while}(e, S)\rangle \Downarrow_{t_{12}} M_1'$. By induction, we have $t_{11} = t_{21}$, $t_{12} = t_{22}$, and $M_{11} \sim_L M_{21}$. Therefore $M_1' \sim_L M_2'$, and $t_1 = t_{11}@t_{12} = t_{21}@t_{22} = t_2$.

Finally, let us consider $S_1; S_2$. Suppose $\Gamma, l_0 \vdash S_1; T_1, \Gamma, l_0 \vdash S_2; T_2$, $\langle M_i, S_1\rangle \Downarrow_{t_{i1}} M_{i1}$, and $\langle M_{i1}, S_2\rangle \Downarrow_{t_{21}} M_i'$. Then by induction assumption, we have $t_{11} = t_{21}$, $t_{12} = t_{22}$, $M_{11} \sim_L M_{12}$, and thus $M_1' \sim_L M_2'$. Therefore $t_1 = t_{11}@t_{12} = t_{21}@t_{22} = t_2$. □

Appendix B: Proof of Theorem 2

Our proof proceeds in two steps. First, we prove a terminating version (Theorem 6) of Theorem 2. By having the assumption that the program will finally terminate, i.e. its execution does not end in an infinite loop, we will show how our type system enforces the MTO property. Then by using Theorem 6, we will show the MTO property holds for non-terminating programs, (i.e. Theorem 7), which implies Theorem 2 as an obvious corollary.

We start with the terminating case. We first prove some useful lemmas for terminating programs.

Lemma 13. *For all* I, ℓ, Υ, Sym, Υ', Sym', T, *such that*

$$\ell \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T$$

if there is some i *such that* $I(i) = \textbf{\textit{jmp}}\ n$, *then* $0 \leq i + n \leq |I|$.

Proof. We prove by induction on

$$\ell \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T$$

It is clearly that rule T-LOAD, T-STORE, T-LOADW, T-STOREW, T-IDB, T-

BOP, T-ASSIGN, T-NOP cannot derive I.

If this judgement is derived using rule T-SEQ, then we know $I = I_1; I_2$, and we have

$$\ell \vdash I_1 : \langle \Upsilon, Sym \rangle \to \langle \Upsilon'', Sym'' \rangle; T_1$$

$$\ell \vdash I_2 : \langle \Upsilon'', Sym'' \rangle \to \langle \Upsilon', Sym' \rangle; T_2$$

Then either $i < |I_1|$, or $|I_1| \leq i < |I_1| + |I_2| = |I|$. For the first case, we have $I_1(i) = \textbf{jmp}\ n$, and thus by induction, we have $0 \leq i + n \leq |I_1| < |I|$. For the second case, we have $I_2(i - |I_1|) = \textbf{jmp}\ n$, and thus by induction assumption, we have $0 \leq i - |I_1| \leq |I_2|$. Therefore, we have $0 < |I_1| \leq i \leq |I_1| + |I_2| = |I|$.

If this judgement is derived by rule T-IF, then we know $I = \iota_1; I_t; \iota_2; I_f$, and

$$\iota_1 = \textbf{br}\ r_1\ rop\ r_2 \hookrightarrow n_1$$

$$\iota_2 = \textbf{jmp}\ n_2$$

$$n_1 - 2 = |I_t|$$

$$n_2 = |I_f| + 1$$

Then, there are three possible scenarios:

1. $1 \leq i \leq 1 + |I_t|$. In this case, we know $I_t(i-1) = \textbf{jmp}\ n$, and $0 \leq i - 1 + n \leq |I_t|$. Therefore $0 < 1 \leq i + n \leq |I_t| + 1 < |I|$;

2. $i = 1 + |I_t|$. In this case, $I(i) = \iota_2$. Therefore, we have $0 < i + n_2 =$

$2 + |I_t| + |I_f| = |I|$;

3. $2 + |I_t| \leq i < 2 + |I_t| + |I_f| = |I|$. In this case, we can prove the conclusion similarly to Case 1.

If this judgement is derived by rule T-WHILE, then we can prove the conclusion similarly to the T-IF case.

Finally, if the judgement is derive by rule T-SUB, then the result follows by induction. □

Lemma 14. *For all* I, ℓ, Υ, Sym, Υ', Sym', T, *such that*

$$\ell \vdash I : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T$$

if there is some i *such that* $I(i) = \textbf{\textit{br}}\ r_1\ rop\ r_2 \hookrightarrow n$, *then* $0 \leq i + n \leq |I|$.

Proof (sketch). The proof is similar to the proof for Lemma 13. □

Lemma 15. *Given* $I = I_1; I_2; I_3$, R_i, S_i, M_i, pc_i *for* $i = 1, ..., k+1$, *and* t_i *for* $i = 1, ..., k$,

$$|I_1| \leq pc_i < |I_1| + |I_2| \quad \forall i \in \{1, ..., k\}$$

$$|I_1| \leq pc_{k+1} \leq |I_1| + |I_2|$$

and

$$|t_i| = 1 \quad \forall i \in \{1, ..., k\}.$$

Then

$$I \vdash (R_i, S_i, M_i, pc_i) \to_{t_i} (R_{i+1}, S_{i+1}, M_{i+1}, pc_{i+1})$$

holds true for $i = 1, ..., k$*, if and only if*

$$I_2 \vdash (R_i, S_i, M_i, pc_i - |I_1|) \rightarrow_{t_i}$$
$$(R_{i+1}, S_{i+1}, M_{i+1}, pc_{i+1} - |I_1|)$$

holds true for $i = 1, ..., k$*.*

Proof. We only prove the only-if-direction, and the if-direction is similar. We prove by induction on k. We first prove for $k = 1$. Consider each possibility for $I(pc_1)$.

Case ldb $k \leftarrow l[r]$**.** By rule LOAD, we know

$$n = |\, R(r) \textbf{ mod } size(l) \,|$$

$$b = M(l, n)$$

$$S_2 = S_1[k \mapsto (b, (l, n))]$$

$$t_1 = select(l, \texttt{read}(n, b), \texttt{eread}(n), l)$$

$$R_2 = R_1 \quad M_2 = M_1 \quad pc_2 = pc_1 + 1$$

Clearly, we have $pc_2 - |I_1| = pc_1 - |I_1| + 1$, and $I_2(pc_1 - |I_1|) = \textbf{ldb } k \leftarrow l[r]$, and thus

$$I_2 \vdash (R_1, S_1, M_1, pc_1 - |I_1|) \rightarrow_{t_1} (R_2, S_2, M_2, pc_2 - |I_1|)$$

Cases **stb** k, $k \leftarrow$ **idb** r, **ldw** $r_1 \leftarrow k[r_2]$, **stw** $r_1 \rightarrow k[r_2]$, $r_1 \leftarrow r_2$ *op* r_3, $r_1 \leftarrow n$, or **nop** are similar.

Case jmp n'. We have

$$R_2 = R_1 \quad S_2 = S_1 \quad M_2 = M_1 \quad pc_2 = pc_1 + n'$$

Since $pc_2 - |I_1 \models pc_1 - |I_1| + n'$, then the conclusion is obvious. We can prove similarly for $I(pc_1) = \textbf{br}\ r_1\ rop\ r_2 \hookrightarrow n'$.

Now, we assume the conclusion holds true for $k \leq k'$. For $k = k' + 1$, We know

$$I_2 \vdash (R_i, S_i, M_i, pc_i - |I_1|) \rightarrow_{t_i} (R_{i+1}, S_{i+1}, M_{i+1}, pc_{i+1} - |I_1|)$$

holds true for $i = 1, ..., k'$ by the induction assumption. Further, since the assumption says the conclusion holds for $k = 1$, thus

$$I_2 \vdash (R_{k'}, S_{k'}, M_{k'}, pc_{k'} - |I_1|) \rightarrow_{t_{k'}} (R_{k'+1}, S_{k'+1}, M_{k'+1}, pc_{k'+1} - |I_1|)$$

Therefore, the conclusion holds true.

□

Using Lemma 13, 14, 15, we can prove the following important lemma.

Lemma 16. *Given $I = I_a; I'; I_b$, where any of I_a and I_b can be empty, ℓ, ℓ', $\Upsilon_1, \Upsilon_1', \Upsilon_2, \Upsilon_2'$, Sym_1, Sym_1', Sym_2, Sym_2', T, T', such that*

$$\ell \vdash I : \langle \Upsilon_1, Sym_1 \rangle \rightarrow \langle \Upsilon_2, Sym_2 \rangle; T$$

and

$$\ell' \vdash I' : \langle \Upsilon'_1, Sym'_1 \rangle \to \langle \Upsilon'_2, Sym'_2 \rangle; T'$$

If for $pc = |I_a|$ *and* pc'*, where* $pc' < |I_a|$ *or* $pc' \geq |I_a| + |I'|$*, such that*

$$I \vdash (R, S, M, pc) \to_t (R', S', M', pc'),$$

then there exists $R'', S'', M'', pc'', t', t'$*, such that* $t = t'@t''$ *(where* t'' *can be* ϵ*), and*

$$I \vdash (R, S, M, pc) \to_{t'} (R'', S'', M'', pc'')$$

$$I \vdash (R'', S'', M'', pc'') \to_{t''} (R', S', M', pc')$$

$$pc'' = |I_a| + |I'| \quad t \equiv t'@t'' \quad |t'| > 0 \quad |t''| \geq 0$$

We provide an intuitive explanation of this lemma as follows. It says suppose a type-checked program I contains a type-checked segment I', and if the the program runs from the start of the segment I' (i.e. $pc = |I_a|$), and ends outside the segment (i.e. $pc' < |I_a|$ or $pc' \geq |I_a| + |I'|$), then it must stop at the end of the segment (i.e. $pc'' = |I_a| + |I'|$) first.

Notice the correctness of this lemma does not depend on whether I_a or I_b can type-check. Further, none or either or even both of I_a and I_b can be empty. In the last case, the conclusion is trivial.

Now we prove this lemma.

Proof. We suppose for $k = |t| - 1$,

$$I \vdash (R_i, S_i, M_i, pc_i) \rightarrow_{t_i} (R_{i+1}, S_{i+1}, M_{i+1}, pc_{i+1})$$

for $i = 0, ..., k$, where

$$|t_i| = 1 \quad \forall i \in \{0, ..., k\}$$

$$R^0 = R \quad S^0 = S \quad M^0 = M \quad pc_0 = pc$$

$$R^k = R' \quad S^k = S' \quad M^k = M' \quad pc_k = pc'$$

Suppose $i^\star$ is the smallest one such that $pc_{i^\star} < |I_a|$ or $pc_{i^\star} \geq |I_a| + |I'|$. Then, by Lemma 15, we know

$$I' \vdash (R_i, S_i, M_i, \hat{pc}_i) \rightarrow_{t_i} (R_{i+1}, S_{i+1}, M_{i+1}, \hat{pc}_{i+1})$$

for $i = 0, ..., i^\star - 2$, where $\hat{pc}_i = pc_i - |I|$. Clearly, we know $0 \leq \hat{pc}_{i^\star - 1} < |I'|$ by the definition of $i^\star$. Now, we consider the instruction $I'(\hat{pc}_{i^\star - 1})$. If it is not **jmp** n' or **br** r_1 *rop* $r_2 \hookrightarrow n'$, then we know

$$pc_{i^\star} = pc_{i^\star - 1} + 1 > |I_a|.$$

Therefore

$$pc_{i^\star} \geq |I_a| + |I'|$$

Further, since $pc_{i^\star-1} - |I_a| \models \hat{pc}_{i^\star-1} < |I'|$, we have

$$pc_{i^\star} \leq |I_a| + |I'|$$

Therefore, we have $pc_{i^\star} = |I_a| + |I'|$, and thus $R'' = R_{i^\star}$, $S'' = S_{i^\star}$, $M'' = M_{i^\star}$, $pc'' = pc_{i^\star}$, $t' = t_0@...@t_{i^\star-1}$, and $t'' = t_{i^\star}@...t_k$ satisfy the property.

If $I'(\hat{pc}_{i^\star} - 1) = \textbf{jmp}\ n'$, then by Lemma 13, we have

$$0 \leq \hat{pc}_{i^\star-1} + n' \leq |I'|$$

Therefore, we have

$$pc_{i^\star} = pc_{i^\star-1} + n' = \hat{pc}_{i^\star-1} + |I_a| + n' \leq |I_a| + |I'|$$

and

$$pc_{i^\star} = pc_{i^\star-1} + n' = \hat{pc}_{i^\star-1} + |I_a| + n' \geq |I_a|$$

Therefore, by the same argument as above, we know $pc_{i^\star} = |I_a| + |I'|$, and the conclusion is true.

Finally, if $I'(\hat{pc}_{i^\star} - 1) = \textbf{br}\ r_1\ rop\ r_2 \hookrightarrow n'$, we can prove the conclusion similarly using Lemma 14. □

Now we can state and prove Theorem 6. Some judgments mentioned in the theorem are defined in Figure B.1.

Theorem 6. *Given a program I in $\mathcal{L}_T$, such that $\ell \vdash I : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T$,*

$$\begin{array}{rcl} mem((b,(l,n))) & = & l \\ idx((b,(l,n))) & = & n \\ block((b,(l,n))) & = & b \end{array}$$

$$\frac{\begin{array}{c}\forall k \in \textbf{BlockIDs}. \\ \Upsilon(k) = mem(S_1(k)) = mem(S_2(k)) \\ \Upsilon(k) = D \Rightarrow S_1(k) = S_2(k) \\ \Upsilon(k) = E \Rightarrow idx(S_1(k)) = idx(S_2(k))\end{array}}{\Upsilon \vdash S_1 \sim S_2}$$

$$\frac{\forall r \in \textbf{Registers}. \Upsilon(r) = \mathtt{L} \Rightarrow R_1(r) = R_2(r)}{\Upsilon \vdash R_1 \sim R_2}$$

$$\boxed{(S, sv) \Downarrow v}$$

$$\frac{\begin{array}{c} mem(S(k)) = l = D \\ (S, sv) \Downarrow v_2 \\ v = block(S(k))(v_2) \end{array}}{(S, M_l[k, sv]) \Downarrow v} \qquad (S, n) \Downarrow n \qquad \frac{\begin{array}{c} (S, sv_1) \Downarrow v_1 \\ (S, sv_2) \Downarrow v_2 \\ v = v_1 \; aop \; v_2 \end{array}}{(S, sv_1 \; aop \; sv_2) \Downarrow v}$$

Figure B.1: Well formedness judgments for proof of Memory-Trace Obliviousness of $\mathcal{L}_{\text{GhostRider}}$

two memories M_1, M_2, two register mapping R_1, R_2, and two scratchpad mapping S_1, S_2, if the following assumptions are satisfied:

1. $M_1 \sim_L M_2$;
2. $\Upsilon \vdash R_1 \sim R_2$;
3. $\Upsilon \vdash S_1 \sim S_2$;
4. $\forall r \in \textbf{\textit{Registers}}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym(r)) \wedge \vdash_{ok} Sym(r) \Rightarrow (S_i, Sym(r)) \Downarrow R_i(r)$;
5. $\forall k \in \textbf{\textit{BlockIDs}}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym(r)) \wedge \vdash_{ok} Sym(k) \Rightarrow \exists n.(S_i, Sym(k)) \Downarrow n \wedge |\, n \; \textbf{\textit{mod}} \; size(\Upsilon(k)) \,| = idx(S_i(k))$.

and

$$I \vdash (R_1, S_1, M_1, 0) \rightarrow_{t_1} (R_1', S_1', M_1', pc')$$

$$I \vdash (R_2, S_2, M_2, 0) \rightarrow_{t_2} (R_2', S_2', M_2', pc'')$$

where $pc' = pc'' = |I|$*, then we have the following conclusions:*

1. $M_1' \sim_L M_2'$;
2. $\Upsilon' \vdash R_1' \sim R_2'$;
3. $\Upsilon' \vdash S_1' \sim S_2'$;
4. $\forall r \in \textbf{\textit{Registers}}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym(r)) \wedge \vdash_{safe} Sym'(r) \Rightarrow (S_i', Sym'(r)) \Downarrow R_i'(r)$;
5. $\forall k \in \textbf{\textit{BlockIDs}}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym(k)) \wedge \vdash_{safe} Sym'(k) \Rightarrow \exists n.(S_i', Sym'(k)) \Downarrow n \wedge |\, n \ \textbf{\textit{mod}}\ size(\Upsilon'(k))\,| = idx(S_i'(k)).$;
6. $t_1 \equiv t_2$

We first provide an intuition of this theorem.

Proof of Theorem 6. We prove Theorem 6 by induction on the length of derivation to derive

$$\mathtt{L} \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T$$

Case T-SEQ. Then $I = I_1; I_2$, and by inversion, we have

$$\mathtt{L} \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym \rangle; T$$

$$\mathtt{L} \vdash I_1 : \langle \Upsilon, Sym \rangle \to \langle \Upsilon_1, Sym_1 \rangle; T_1$$

$$\mathtt{L} \vdash I_2 : \langle \Upsilon_1, Sym_1 \rangle \to \langle \Upsilon', Sym' \rangle; T_2$$

By Lemma 15 and Lemma 16, we know

$$I_1 \vdash (R_1, S_1, M_1, 0) \to_{t_1^1} (R_1'', S_1'', M_1'', |I_1|)$$

$$I_2 \vdash (R_1'', S_1'', M_1'', 0) \to_{t_1^2} (R_1', S_1', M_1', |I_2|)$$

Similarly, we have

$$I_1 \vdash (R_2, S_2, M_2, 0) \to_{t_2^1} (R_2'', S_2'', M_2'', |I_1|)$$

$$I_2 \vdash (R_2'', S_2'', M_2'', 0) \to_{t_2^2} (R_2', S_2', M_2', |I_2|)$$

By induction assumption, we have

1. $M_1'' \sim_L M_2''$;
2. $\Upsilon \vdash R_1'' \sim R_2''$;
3. $\Upsilon \vdash S_1'' \sim S_2''$;
4. $\forall r \in \textbf{Registers}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym''(r)) \wedge \vdash_{safe} Sym''(r) \Rightarrow (S_i'', Sym''(r)) \Downarrow R_i'(r)$;
5. $\forall k \in \textbf{BlockIDs}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym''(k)) \wedge \wedge \vdash_{safe} Sym''(k) \Rightarrow \exists n.(S_i'', Sym''(k)) \Downarrow n \wedge |\, n \textbf{ mod } size(\Upsilon''(k)) \,| = idx(S_i''(k))$.

6. $t_1^1 \equiv t_2^1$

By induction assumption again, we have conclusions 1-5, and $t_1^2 \equiv t_2^2$. Therefore, we know $t_1 = t_1^1 @ t_1^2 \equiv t_2^1 @ t_2^2 = t_2$, which is conclusion 6.

Case T-LOAD. Then $I = \textbf{ldb}\ k \leftarrow l[r]$, and $pc' = pc'' = 1$. Further, by inversion, we know

$$n_i = |R_i(r)\ \textbf{mod}\ size(l)|$$

$$b_i = M_i(l, n_i) \quad S_i' = S_i[k \mapsto (b_i, (l, n_i))]$$

$$t_i = select(l, \texttt{read}(n_i, b_i), \texttt{eread}(n_i), l) \quad i = 1, 2$$

We prove conclusions 1-6 hold true. 1 holds true trivially, since the memories are not changed, i.e. $M_1' = M_1 \sim_L M_2 = M_2'$, and 2 and 4 hold true for the same reason.

We prove conclusion 3 as follows. First, we know $\Upsilon'(k) = l = mem(S_1'(k)) = mem(S_2'(k))$. If $\Upsilon'(k) = D$ or $\Upsilon'(k) = E$, then we know $l \notin \textbf{ORAMbanks}$, and thus $\Upsilon'(r) = \texttt{L}$, which implies that $idx(S_1'(k)) = n_1 = n_2 = idx(S_2'(k))$. Further, if $\Upsilon'(k) = l = D$, then we know $b_1 = M_1(D, n_1) = M_2(D, n_2) = b_2$ (due to $M_1 \sim_L M_2$), which implies $S_1'(k) = S_2'(k)$. Therefore, we know $\Upsilon \vdash S_1' \sim S_2'$.

For conclusion 5, since for all $k' \neq k$, $Sym'(k') = Sym(k')$ and $S_i'(k') = S_i(k')$ $(i = 1, 2)$, therefore conclusion 7 holds true. For k, if $\vdash_{safe} Sym'(k)$ does not hold, then the conclusion is trivial. Now we assume $\vdash_{safe} Sym'(k)$, if and only if $\vdash_{safe} Sym(r)$. By assumption 6, we know $(S_i, Sym(r)) \Downarrow R_i(r)$. Since $|\, R_i(r)\ \textbf{mod}\ size(\Upsilon'(k))\,| = |\, R_i(r)\ \textbf{mod}\ size(l)\,| = n_i$, we know conclusion 7 holds true.

For conclusion 6, if $l = D$, then following the discussion for conclusion 3, we know $n_1 = n_2$ and $b_1 = b_2$, and thus $t_1 = \texttt{read}(n_1, b_1) = \texttt{read}(n_2, b_2) = t_2$. If $l = E$, then similarly we know $n_1 = n_2$, and thus $t_1 = \texttt{eread}(n_1) = \texttt{eread}(n_2) = t_2$. If $l \in \textbf{ORAMBanks}$, then we know $t_1 = l = t_2$. Thus conclusion 6 holds true.

Case T-STORE. If $I = \textbf{stb}\ k$, and I is typed using rule T-STORE, then we have $pc' = pc'' = 1$. Further, we know

$$(b_i, a_i) = S_i(k) \quad a_i = (l_i, n_i) \quad M_i' = M_i[a_i \mapsto b_i]$$

$$t_i = \mathit{select}(l_i, \texttt{write}(n_i, b_i), \texttt{ewrite}(n_i), l_i) \quad i = 1, 2$$

Conclusions 2-5 are trivial, since registers and scratchpads are not changed. We first prove conclusion 6. By assumption 3, we know $l_1 = l_2$. If $l_1 = D$, then we know $t_1 = \texttt{write}(n_1, b_1)$, and $t_2 = \texttt{write}(n_2, b_2)$. By assumption 4, we know $n_1 = n_2$ and $b_1 = b_2$, therefore $t_1 = t_2$. If $l_1 = E$, then by assumption 5, we know $n_1 = n_2$. Therefore $t_1 = \texttt{ewrite}(n_1) = \texttt{ewrite}(n_2) = t_2$. Finally, if $l_1 \in \textbf{ORAMBanks}$, then $t_1 = l_1 = l_2 = t_2$.

Now we prove conclusion 1. The only difference between M_i' and M_i is the value for $a_i = (l_i, n_i)$. To show that $M_1' \sim_L M_2'$, we only need to show that if $l_1 = l_2 = D$, and $b_1 = b_2$. This point is induced by assumption 4. Therefore, conclusion 1 holds.

Case T-LOADW. If $I = \textbf{ldw}\ r_x \leftarrow k[r_y]$, and I is typed using rule T-

LOADW, then we have $pc' = pc'' = 1$. Further, we know

$$(b_i, a_i) = S_i(k) \quad n_i = \mid R(r_y) \textbf{ mod } size(b_i) \mid$$

$$R'_i = R_i[r_x \mapsto b_i(n_i)] \quad i = 1, 2$$

Since scratchpad and memories are not changed, conclusions 1, 3, and 5, trivially hold true. Further, since $t_1 = \mathbf{f} = t_2$, conclusion 6 is also true. We only need to prove for conclusions 2 and 4. For conclusion 2, if $\Upsilon'(r_x) = H$, the conclusion is trivial. If $\Upsilon'(r_x) = L$, then we know $l = D$, and by rule T-LOADW, we know $\Upsilon(r_y) = L$, which implies, by assumption 2, $n_1 = n_2$. Further, since $l = D$, by assumption 3, we know $b_1 = b_2$. Therefore $R'_1(r_x) = b_1(n_1) = b_2(n_2) = R'_2(r_x)$.

For conclusion 4, all we need to show is that for $i = 1, 2$, either $(\ell = H \vee \vdash_{const} M_l[k, Sym(r_y)]) \wedge \vdash_{safe} M_l[k, Sym(r_y)]$ is not true, $\ell = L$ or

$$(S_i, M_l[k, Sym(r_2) \textbf{ mod } size(b_i)]) \Downarrow b_i(n_i)$$

holds true. First of all, $\vdash_{const} M_l[k, Sym(r_y)]$ is not true. W.L.O.G. we suppose $\vdash_{safe} M_l[k, Sym(r_y)]$ and $\ell = H$, then we know $l = D$ and $\vdash_{safe} Sym(r_y)$. Therefore we have $(S_i, Sym(r_y) \textbf{ mod } size(b_i)) \Downarrow R(r_y) \textbf{ mod } size(b_i) = n_i$. Further, by conclusion 3, we know $\Upsilon'(k) = mem(S_i(k)) = D$. Combining with $b_i = block(S_i(k))$, we have

$$(S_i, M_l[k, Sym(r_2) \textbf{ mod } size(b_i)]) \Downarrow b_i(n_i)$$

Case T-IDB. If $I = k \leftarrow \textbf{idb } r$, and I is typed using rule T-IDB, then we

have $pc' = pc'' = 1$. Further, we know

$$(b_i, (l_i, n_i)) = S_i(k) \quad R_i' = R_i[r \mapsto n_i] \quad i = 1, 2$$

Conclusions 1, 3, 5, and 6 hold trivially. Conclusions 2 and 4 are implied by assumptions 3 and 5 respectively.

Case T-STOREW. If $I = \textbf{stw}\ r_x \rightarrow k[r_y]$, and I is typed using rule T-STOREW, then we have $pc' = pc'' = 1$. Further, we know

$$(b_i, a_i) = S_i(k) \quad n_i = |\, R_i(r_y)\ \textbf{mod}\ size(b)\,|$$
$$S_i' = S_i[k \mapsto (b_i[n_i \mapsto R_i(r_x)], a_i) \quad i = 1, 2$$

Since registers and memories are not changed, conclusions 1, 2, and 4 hold true. Further, since $t_1 = \mathbf{f} = t_2$, we know conclusion 6 is true. We now prove conclusions 3 and 5. Clearly, we only need to prove for $\Upsilon'(k)$ and $Sym(k)$. Suppose $a_i = (l_i, idx_i)$, then we know $\Upsilon'(k) = \Upsilon(k) = l_1 = l_2$. Further, if $\Upsilon'(k) = \Upsilon(k) = D$, then we know $\Upsilon(r_x) = \Upsilon(r_y) = L$, which, by assumption 2, implies $R_1(r_x) = R_2(r_x)$ and $R_1(r_y) = R_2(r_y)$. Therefore $n_1 = n_2$. Since $\Upsilon(k) = D$, by assumption 4, we have $S_1(k) = S_2(k)$, and thus $b_1 = b_2$. Therefore we have $S_1'(k) = (b_1[n_1 \mapsto R_1(r_x)], a_1) = (b_2[n_2 \mapsto R_2(r_x)], a_2) = S_2'(k)$. Finally, if $\Upsilon'(k) = \Upsilon(k) = E$, then by assumption 5, we know $idx_1 = idx(S_1(k)) = idx(S_2(k)) = idx_2$. Then we have $idx(S_1'(k)) = idx_1 = idx_2 = idx(S_2'(k))$. Therefore, conclusion 3 is true.

For conclusion 5, first we suppose $\ell = H$ and $\vdash_{safe} Sym'(k)$. Since $Sym'(k) = Sym(k)$, we know $\vdash_{safe} Sym(k)$, and thus by assumption 5, we know for some

n_1, n_2, $(S_i, Sym'(k)) \Downarrow n$ and $|\, n \,| \Upsilon(k) = idx_i$. Further, since $\ell = H$, we know $slab(\Upsilon(k)) = H$, and thus $mem(S_i(k)) = l_i \neq D$ for $i = 1, 2$. Now we prove that if $(S_1, sv) \Downarrow v$, then $(S'_1, sv) \Downarrow v$ by induction on sv. If $sv = M_l[k', sv_1]$, then we know $mem(S_1(k')) = l = D$, and $(S_1, sv) \Downarrow v_2$. Since $mem(S_1(k)) \neq D$, we know $k \neq k'$. Therefore we know $S'_1(k) = S_1(k)$, and thus by induction assumption, we have $(S'_1, sv) \Downarrow v$. Next, if $sv = n$, then the conclusion holds trivially. If $sv = sv_1\ aop\ sv_2$, then we know $(S_1, sv_1) \Downarrow v_1$, $(S_1, sv_2) \Downarrow v_2$, and $v = v_1\ aop\ v_2$. Therefore we know $(S'_1, sv) \Downarrow v$ by induction. Similarly, we can prove that if $(S_2, sv) \Downarrow v$, then $(S'_2, sv) \Downarrow v$. Therefore, we know $(S'_i, Sym'(k)) \Downarrow n_i$, and $|\, n_i \ \mathbf{mod}\ size(\Upsilon'(k)) \,| = idx_i = idx(S'_i(k))$, which means conclusion 5 holds true. Similarly, if $\vdash_{const} Sym'(k)$ and $\vdash_{safe} Sym'(k)$, we can also prove conclusion 5 easily.

Case T-BOP. If $I = r_1 \leftarrow r_2\ aop\ r_3$, and I is typed using rule T-BOP, then we have $pc' = pc'' = 1$. Further, we know

$$n_i = R_i(r_2)\ aop R_i(r_3) \quad R'_i = R_i[r_1 \mapsto n_i] \quad i = 1, 2$$

Conclusions 1, 3, 5, and 6 are trivial. For conclusion 2, we only need to prove if $\Upsilon'(r_1) = l' = \Upsilon(r_2) \sqcup \Upsilon(r_3) = L$, then $R_1(r_1) = n_1 = R_2(r_1) = n_2$. To see this, since $\Upsilon(r_2) \sqcup \Upsilon(r_3) = L$, we know $\Upsilon(r_2) = \Upsilon(r_3) = L$. Therefore by assumption 2, we know $R_1(r_2) = R_2(r_2)$ and $R_1(r_3) = R_2(r_3)$. Therefore, we have

$$n_1 = R_1(r_2)\ aop\ R_1(r_3) = R_2(r_2)\ aop\ R_2(r_3) = n_2$$

For conclusion 4, we only need to consider $Sym'(r_1)$. If

$$\vdash_{safe} Sym'(r_1) = Sym(r_2)\ aop\ Sym(r_3)$$

then we know $\vdash_{safe} Sym(r_2)$ and $\vdash_{safe} Sym(r_3)$. Further, we know $\ell = \mathtt{H}$ or $\vdash_{const} Sym(r_1)$ and $\vdash_{const} Sym(r_2)$. Therefore, by assumption 6, we know

$$(S_i, Sym(r_2)) \Downarrow R_i(r_2), (S_i, Sym(r_3)) \Downarrow R_i(r_3) \text{ for } i = 1, 2$$

Therefore, we have

$$(S_i, Sym'(r_1)) \Downarrow R_i(r_2)\ aop\ R_i(r_3) = n_i = R'_i(r_1) \quad i = 1, 2$$

Therefore conclusion 4 holds true.

If $I = r \leftarrow n$, and I is typed using rule T-ASSIGN, then we have $pc = 0$ and $pc' = pc'' = 1$. Further, we know

$$R'_i = R_i[r_1 \mapsto n] \quad i = 1, 2$$

Similar to the T-BOP rule, conclusions 1, 3, 5 and 6 are trivial. For conclusion 2, we know $R'_1(r_1) = R'_2(r_1) = n$. For conclusion 4, we have $(S_i, n) \Downarrow n$ for $i = 1, 2$.

Case T-NOP. If $I =$ **nop**, and I is typed using T-NOP, then this is trivial, because $M_i = M'_i$, $R_i = R'_i$, $S_i = S'_i$ for $i = 1, 2$, and $\Upsilon = \Upsilon'$ and $Sym = Sym'$.

Case T-SUB. If I is typed using T-SUB, then we know

$$\ell \vdash \iota : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon'', Sym'' \rangle; T$$

where $\Upsilon'' \preceq \Upsilon'$ and $Sym'' \preceq Sym'$. W.L.O.G, we assume $\ell \vdash \iota : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon'', Sym'' \rangle; T$ is not derived by rule T-SUB (i.e. $\preceq$ is transitive). By induction, we have the following results

1. $M'_1 \sim_L M'_2$;
2. $\Upsilon \vdash R'_1 \sim R'_2$;
3. $\Upsilon \vdash S'_1 \sim S'_2$;
4. $\forall r \in \textbf{Registers}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym''(r)) \wedge \vdash_{safe} Sym''(r) \Rightarrow (S'_i, Sym''(r)) \Downarrow R'_i(r)$;
5. $\forall k \in \textbf{BlockIDs}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym''(r)) \wedge \vdash_{safe} Sym''(k) \Rightarrow \exists n.(S'_i, Sym'(k)) \Downarrow n \wedge |\, n \textbf{ mod } size(\Upsilon'(k)) \,| = idx(S'_i(k))$.
6. $t_1 \equiv t_2$

Therefore, conclusions 1 and 6 follow results 1 and 6. For conclusion 2, if $\Upsilon'(r) = \mathtt{L}$, then since $\Upsilon''(r) \sqsubseteq \Upsilon'(r)$, we know $\Upsilon''(r) = \mathtt{L}$, and thus $R'_1(r) = R'_2(r)$.

Conclusion 3 naturally holds true, since $\Upsilon''(k) = \Upsilon'(k)$. For conclusion 4, since $Sym'' \preceq Sym'$, we know $Sym'(r) =?$ or $Sym'(r) = Sym''(r)$. In the former case, $\vdash_{safe} Sym'(r)$ does not hold, and thus conclusion 4 is vacuously true. In the later

case, conclusion 4 directly follows results 4. Similarly, we can prove conclusion 5 as well.

Case T-LOOP. If I is typed using rule T-LOOP, then we know $I = I_c; \iota_1; I_b; \iota_2$, and

$$\ell \vdash I_c : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle$$

$$\ell \vdash I_b : \langle \Upsilon', Sym' \rangle \to \langle \Upsilon, Sym \rangle$$

We prove the conclusion by induction on number of times that instruction ι_1 is executed. First, assume ι_1 is executed once. Therefore, by Lemma 16, we know that

$$I_c \vdash (R_1, S_1, M_1, 0) \to_{t_1^1} (R_1'', S_1'', M_1'', |I_c|)$$

$$I \vdash (R_1'', S_1'', M_1'', |I_c|) \to_{t_1^2} (R_1', S_1', M_1', pc')$$

$$I_c \vdash (R_2, S_2, M_2, 0) \to_{t_2^1} (R_2'', S_2'', M_2'', |I_c|)$$

$$I \vdash (R_2'', S_2'', M_2'', |I_c|) \to_{t_2^2} (R_2', S_2', M_2', pc'')$$

$$t_1 = t_1^1 @ t_1^2 \quad t_2 = t_2^1 @ t_2^2$$

By induction assumption, we have

1. $M_1'' \sim_L M_2''$;
2. $\Upsilon \vdash R_1'' \sim R_2''$;
3. $\Upsilon \vdash S_1'' \sim S_2''$;

4. $\forall r \in \textbf{Registers}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym''(r)) \wedge \vdash_{safe} Sym''(r) \Rightarrow (S_i'', Sym''(r)) \Downarrow R_i''(r)$;

5. $\forall k \in \textbf{BlockIDs}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym''(r)) \wedge \vdash_{safe} Sym''(k) \Rightarrow \exists n.(S_i'', Sym'(k)) \Downarrow n \wedge \mid n \textbf{ mod } size(\Upsilon''(k)) \models idx(S_i''(k))$.

6. $t_1^1 \equiv t_2^1$

Further, we know $I(|I_c|) = \iota_1 = \textbf{br } r_1 \; rop \; r_2 \hookrightarrow n_1$, where $\Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathtt{L}$, which implies $\Upsilon(r_1) = \Upsilon(r_2) = \mathtt{L}$. Therefore, we know $R_1''(r_1) = R_2''(r_1)$ and $R_1''(r_2) = R_2''(r_2)$, which implies if

$$I \vdash (R_i'', S_i'', M_i'', |I_c|) \rightarrow_{\mathbf{f}} (R_i^\star, S_i^\star, M_i^\star, pc_i^\star)$$

then $pc_1^\star = pc_2^\star$, which is either $|I_c| + 1$ or $|I_c| + n_1 = |I|$. If the first case is true, then we can show

$$I_c \vdash (R_1, S_1, M_1, 0) \rightarrow_{t_1^1} (R_1'', S_1'', M_1'', |I_c|)$$
$$I \vdash (R_1'', S_1'', M_1'', |I_c|) \rightarrow_{t_1^2} (R_1'', S_1'', M_1'', |I_c| + 1)$$
the branch instruction ι_1 will not be taken.
$$I \vdash (R_1'', S_1'', M_1'', |I_c| + 1) \rightarrow_{t_1^3} (R_1''', S_1''', M_1''', |I| - 1)$$
$$I \vdash (R_1''', S_1''', M_1''', |I| - 1) \rightarrow_{t_1^4} (R_1''', S_1''', M_1''', 0)$$
$$I \vdash (R_1''', S_1''', M_1''', 0) \rightarrow_{t_1^5} (R_1', S_1', M_1', |I|)$$

Then by the same analysis, we know during $I \vdash (R_1''', S_1''', M_1''', 0) \rightarrow_{t_1^5} (R_1', S_1', M_1', |I|)$, the instruction ι_1 will be executed at least once, and thus it will be executed at least

twice in total, which contradicts our assumption that ι_1 is executed only once. Therefore, we know $pc_1^\star = pc_2^\star = |I|$. Therefore, we know $t_1 = t_1^1@\mathbf{f} \equiv t_2^2@\mathbf{f} = t_2$, and $R_i'' = R_i$, $S_i'' = S_i$, $M_i'' = M_i$ $(i = 1, 2)$. In this case, conclusions 1-6 all hold true.

Next, we assume the conclusions hold true for the number of the times that ι_1 is executed less than $u > 1$, and we consider the case when $|t_1| = u$. By the same analysis as above, we know

$$I_c \vdash (R_1, S_1, M_1, 0) \rightarrow_{t_1^1} (R_1'', S_1'', M_1'', |I_c|)$$

$$I \vdash (R_1'', S_1'', M_1'', |I_c|) \rightarrow_{\mathbf{f}} (R_1'', S_1'', M_1'', |I_c| + 1)$$

the branch instruction ι_1 will not be taken.

$$I \vdash (R_1'', S_1'', M_1'', |I_c| + 1) \rightarrow_{t_1^2} (R_1''', S_1''', M_1''', |I| - 1)$$

$$I \vdash (R_1''', S_1''', M_1''', |I| - 1) \rightarrow_{\mathbf{f}} (R_1''', S_1''', M_1''', 0)$$

$$I \vdash (R_1''', S_1''', M_1''', 0) \rightarrow_{t_1^3} (R_1', S_1', M_1', |I|)$$

$$I_c \vdash (R_2, S_2, M_2, 0) \rightarrow_{t_2^1} (R_2'', S_2'', M_2'', |I_c|)$$

$$I \vdash (R_2'', S_2'', M_2'', |I_c|) \rightarrow_{\mathbf{f}} (R_2'', S_2'', M_2'', |I_c| + 1)$$

the branch instruction ι_1 will not be taken.

$$I \vdash (R_2'', S_2'', M_2'', |I_c| + 1) \rightarrow_{t_2^2} (R_2''', S_2''', M_2''', |I| - 1)$$

$$I \vdash (R_2''', S_2''', M_2''', |I| - 1) \rightarrow_{\mathbf{f}} (R_2''', S_2''', M_2''', 0)$$

$$I \vdash (R_2''', S_2''', M_2''', 0) \rightarrow_{t_2^3} (R_2', S_2', M_2', |I|)$$

$$t_1 = t_1^1@\mathbf{f}@t_1^2@\mathbf{f}@t_1^3 \qquad t_2 = t_2^1@\mathbf{f}@t_2^2@\mathbf{f}@t_2^3$$

By induction assumption, we know

1. $M_1'' \sim_L M_2''$, $M_1''' \sim_L M_2'''$, and $M_1' \sim_L M_2'$;

2. $\Upsilon \vdash R_1'' \sim R_2''$, $\Upsilon \vdash R_1''' \sim R_2'''$, and $\Upsilon \vdash R_1' \sim R_2'$;

3. $\Upsilon \vdash S_1'' \sim S_2''$, $\Upsilon \vdash S_1''' \sim S_2'''$, and $\Upsilon \vdash S_1' \sim S_2'$;

4. $\forall r \in \textbf{Registers}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym''(r)) \wedge \vdash_{safe} Sym''(r) \Rightarrow (S_i'', Sym''(r)) \Downarrow R_i''(r)$;

5. $\forall r \in \textbf{Registers}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym'''(r)) \wedge \vdash_{safe} Sym'''(r) \Rightarrow (S_i''', Sym'''(r)) \Downarrow R_i'''(r)$;

6. $\forall r \in \textbf{Registers}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym'(r)) \wedge \vdash_{safe} Sym'(r) \Rightarrow (S_i', Sym'(r)) \Downarrow R_i'(r)$;

7. $\forall k \in \textbf{BlockIDs}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym''(r)) \wedge \vdash_{safe} Sym''(k) \Rightarrow$
   $\exists n.(S_i'', Sym''(k)) \Downarrow n$
   $\wedge | \, n \textbf{ mod } size(\Upsilon''(k)) \models idx(S_i''(k))$.

8. $\forall k \in \textbf{BlockIDs}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym'''(r)) \wedge \vdash_{safe} Sym'''(k) \Rightarrow$
   $\exists n.(S_i''', Sym'''(k)) \Downarrow n$
   $\wedge | \, n \textbf{ mod } size(\Upsilon'''(k)) \models idx(S_i'''(k))$.

9. $\forall k \in \textbf{BlockIDs}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym'(r)) \wedge \vdash_{safe} Sym'(k) \Rightarrow$
   $\exists n.(S_i', Sym'(k)) \Downarrow n$
   $\wedge | \, n \textbf{ mod } size(\Upsilon'(k)) \models idx(S_i'(k))$.

10. $t_1^1 \equiv t_2^1$, $t_1^2 \equiv t_2^2$, and $t_1^3 \equiv t_2^3$

Then conclusions 1-5 are true by the above results, and for conclusion 6, we have $t_1 = t_1^1@\mathbf{f}@t_1^2@\mathbf{f}@t_1^3 \equiv t_2^1@\mathbf{f}@t_2^2@\mathbf{f}@t_2^3 = t_2$.

Case T-IF. Thus $I = \iota_1; I_t; \iota_2; I_f$, where $\iota_1 = \mathbf{br}\ r_1\ rop\ r_2 \hookrightarrow n_1$. Consider ℓ'. If $\ell' = \mathtt{L}$, then we know $\Upsilon(r_1) = \Upsilon(r_2) = \mathtt{L}$. In this case, by assumption 2, we know $R_1(r_1) = R_2(r_1)$ and $R_1(r_2) = R_2(r_2)$. Therefore the program counter goes to the same value in both cases. Formally speaking, we have

$$I \vdash \langle R_i, S_i, M_i, 0\rangle \rightarrow_{\mathbf{f}} \langle R_i, S_i, M_i, pc_i\rangle \quad i = 1, 2$$

where $pc_1 = pc_2$, which is either 1, or $n_1 = |I_t| + 2$. If $pc_1 = 1$, then by Lemma 16, we know

$$I \vdash \langle R_i, S_i, M_i, 1\rangle \rightarrow_{t_i'} \langle R_i'', S_i'', M_i'', |I_t| + 1\rangle \quad i = 1, 2$$

Further, since $I(|I_t| + 1) = \iota_2 = \mathbf{jmp}\ n_2$, we know

$$I \vdash \langle R_i'', S_i'', M_i'', |I_t| + 1\rangle \rightarrow_{\mathbf{f}} \langle R_i'', S_i'', M_i'', |I|\rangle$$

for $i = 1, 2$, which implies $R_i'' = R_i'$, $S_i'' = S_i'$, $M_i'' = M_i'$, and $t_i = \mathbf{f}@t_i'@\mathbf{f}$. It is easy to prove, by induction assumption, that conclusions 1-5 hold true, and for 6, we have

$$t_1 = \mathbf{f}@t_1'@\mathbf{f} = \mathbf{f}@t_2'@\mathbf{f} = t_2$$

Similarly, if $pc_1 = pc_2 = n_1$, we can also prove the conclusions. Intuitively, this means if $\ell = \mathtt{L}$, then the branching statement in this if-block will go to the same

$$\frac{T = \mathbf{read}(l,k,sv) \qquad l = D \qquad (S,sv) \Downarrow n \qquad b = M(l,n) \qquad S' = S[k \mapsto (b,(l,n))] \qquad t = \mathbf{read}(n,b)}{\langle S,M,T\rangle \rightarrow_t \langle S',M\rangle}$$

$$\frac{T = \mathbf{read}(l,k,sv) \quad l = E \quad (S,sv) \Downarrow n \quad t = \mathbf{eread}(n) \quad S' = S[k \mapsto (\star,(l,n))]}{\langle S,M,T\rangle \rightarrow_t \langle S',M\rangle}$$

$$\frac{T = \mathbf{write}(l,k,sv) \qquad l = D \quad S(k) = (b,(l,n)) \quad t = \mathbf{write}(n,b) \quad M' = M[(l,n) \mapsto b]}{\langle S,M,T\rangle \rightarrow_t \langle S,M'\rangle}$$

$$\frac{T = \mathbf{write}(l,k,sv) \qquad l = E \quad S(k) = (\star,(l,n)) \quad t = \mathbf{write}(n,b) \quad M' = M[(l,n) \mapsto b]}{\langle S,M,T\rangle \rightarrow_t \langle S,M\rangle}$$

$$\frac{T = o \qquad t = o}{\langle S,M,T\rangle \rightarrow_t \langle S,M\rangle}$$

$$\frac{T = \mathbf{F} \qquad t = \mathbf{f}}{\langle S,M,T\rangle \rightarrow_t \langle S,M\rangle}$$

$$\frac{T = T_1 @ T_2 \quad \langle S,M,T_1\rangle \rightarrow_{t_1} \langle S',M'\rangle \quad \langle S',M',T_2\rangle \rightarrow_{t_2} \langle S'',M''\rangle}{\langle S,M,T_1@T_2\rangle \rightarrow_{t_1@t_2} \langle S'',M''\rangle}$$

$$\frac{t_1 \equiv t_2 \quad \langle S,M,T\rangle \rightarrow_{t_1} \langle S',M'\rangle}{\langle S,M,T\rangle \rightarrow_{t_2} \langle S',M'\rangle}$$

Figure B.2: Symbolic Execution in $\mathcal{L}_{\text{GhostRider}}$

branch, and we can prove the theorem.

Next, we consider when $\ell' = \texttt{H}$. If the branching statement goes to the same pc, then based on the above discussion, we know the conclusions hold true. With out loss of generality, we can consider when the branching instruction goes to different pc. We first study the relationship between a trace and a trace pattern. We first prove conclusion 6, i.e. $t_1 = t_2$. To prove this, we first define a new notion $\langle S,M,T\rangle \rightarrow_t \langle S',M'\rangle$ as in Figure B.2.

Here, we use a special value $\star$ to indicate a special block, that we do not care about its content. By doing so, we can use only the DRAM part of a memory M to

make this definition. It is easy to see that evaluating a symbolic value requires only scratchpad and memory corresponding to DRAM. We defined the *DRAM projection* of scratchpad and memory as follows:

$$S_D(k) = \begin{cases} (b,(D,n)) & S(k) = (b,(D,n)) \\ (\star,(E,n)) & S(k) = (b,(E,n)) \\ \text{undefined} & \textbf{otherwise} \end{cases}$$

Actually, the DRAM projection of scratchpad does not contain only DRAM, but also contain the index information about ERAM. But all these information are assumed to be public.

$$M_D(l,n) = \begin{cases} M(l,n) & l = D \\ \text{undefined} & \textbf{otherwise} \end{cases}$$

It is easy to see, if $\langle S, M, T\rangle \rightarrow_t \langle S', M'\rangle$, then we also have

$$\langle S_D, M_D, T\rangle \rightarrow_t \langle S'_D, M'_D\rangle$$

It is worth mentioning that given two low-equivalent memories $M^1 \sim_L M^2$ if and only if $M^1_D = M^2_D$.

Based on these rules, we can prove the following lemmas.

Lemma 17. *Given* $S, M, S', M', S'', M'', T_1, T_2$, *if* $T_1 \equiv T_2$, *and*

$$\langle S, M, T_1\rangle \rightarrow_{t_1} \langle S', M'\rangle$$

$$\langle S, M, T_2\rangle \rightarrow_{t_2} \langle S'', M''\rangle$$

then we have

$$t_1 \equiv t_2$$

$$S' = S''$$

$$M' = M''$$

Proof. We consider how $T_1 \equiv T_2$ is derived. **Case 1.** $T_1 = \mathbf{read}(l, k, sv_1)$ and $T_2 = \mathbf{read}(l, k, sv_2)$. Then we know $\vdash_{safe} sv_1$ and $sv_1 = sv_2$. Therefore we know if $(S, sv_1) \Downarrow n_1$ and $(S, sv_2) \Downarrow n_2$, then $n_1 = n_2$. If $l = D$, then we know $b = M(l, n_1) = M(l, n_2)$, and thus $t_1 = \mathbf{read}(n_1, b) = \mathbf{read}(n_2, b) = t_2$. Further $S' = S[k \rightarrow (b, (l, n_1))] = S[k \rightarrow (b, (l, n_2))] = S''$, and $M' = M = M''$. If $l = E$, then $t_1 = \mathbf{eread}(n_1) = \mathbf{eread}(n_2) = t_2$. $S' = S[k \rightarrow (\star, (l, n_1))] = S[k \rightarrow (\star, (l, n_2))] = S''$ and $M' = M''$ is trivial. It is impossible for l to be an ORAM bank.

Case 2. $T_1 = \mathbf{write}(l, k, sv_1)$ and $T_2 = \mathbf{write}(l, k, sv_2)$. If $l = D$ or $l = E$, then $S = (b, (l, n))$ (where $b = \star$ if $l = E$), and $t_1 = \mathbf{write}(n, b) = t_2$. Further $M' = M[(l, n) \mapsto b] = M''$, and $S' = S = S''$. It is impossible for l to be an ORAM bank.

Case 3. $T_1 = T_2 = o$ or $T_1 = T_2 = \mathbf{F}$, then the conclusion is trivial.

Case 5. $T_1 = T_x@(T_y@T_z)$ and $T_2 = (T_x@T_y)@T_z$. In this case, we know

$$\langle S, M, T_x\rangle \rightarrow_{t_x} \langle S^2, M^2\rangle$$

$$\langle S^2, M^2, T_y @ T_z \rangle \rightarrow_{t_{yz}} \langle S', M' \rangle$$

Further, we have

$$\langle S^2, M^2, T_y \rangle \rightarrow_{t_y} \langle S^3, M^3 \rangle$$

$$\langle S^2, M^2, T_z \rangle \rightarrow_{t_z} \langle S', M' \rangle$$

Therefore we have $t_1 = t_x @ (t_y @ t_z)$. Further, we know

$$\langle S, M, T_x \rangle \rightarrow_{t'_x} \langle S^\star, M^\star \rangle$$

$$\langle S^\star, M^\star, T_y \rangle \rightarrow_{t'_y} \langle S^{\star\star}, M^{\star\star} \rangle$$

$$\langle S^{\star\star}, M^{\star\star}, T_z \rangle \rightarrow_{t'_z} \langle S'', M'' \rangle$$

By induction assumption, we know $S^\star = S^2$, $M^\star = M^2$, $t_x \equiv t'_x$. Further, by induction assumption again, we have $S^{\star\star} = S^3$, $M^{\star\star} = M^3$, $t_y \equiv t'_y$. Finally, by applying induction assumption once more, we finally have $S' = S''$, $M' = M''$, and $t_z \equiv t'_z$. Therefore we know

$$t_1 = t_x @ (t_y @ t_z) \equiv (t_x @ t_y) @ t_z \equiv (t'_x @ t'_y) @ t'_z \equiv t_2$$

Case 6. $T_1 = T_x @ T_y$ and $T_2 = T'_x @ T'_y$. Then we know

$$\langle S, M, T_x \rangle \rightarrow_{t_x} \langle S^1, M^1 \rangle$$

$$\langle S^1, M^1, T_y\rangle \to_{t_y} \langle S', M'\rangle$$

and

$$\langle S, M, T'_x\rangle \to_{t'_x} \langle S^2, M^2\rangle$$

$$\langle S^2, M^2, T'_y\rangle \to_{t'_y} \langle S'', M''\rangle$$

Since $T_x \equiv T'_x$, we know by induction assumption, $t_x \equiv t'_x$, $S^1 = S^2$, and $M^1 = M^2$. Then applying induction assumption again, we have $t_y \equiv t'_y$, $S' = S''$, and $M' = M''$. □

Lemma 18. *Given a program I, register mappings R, R', scratchpads S, S', and memories M, M', if*

$$\ell \vdash I : \langle \Upsilon, Sym\rangle \to \langle \Upsilon', Sym'\rangle; T$$

where $\ell = \boldsymbol{H}$, $T \equiv T'$ (for some T'), R, R', S, S', M, M' satisfy assumptions 1-7 in Theorem 6 and

$$I \vdash \langle R, S, M, 0\rangle \to_t \langle R', S', M', |I|\rangle$$

then we have

$$\langle S_D, M_D, T\rangle \to_t \langle S'_D, M'_D\rangle$$

Proof. We prove by induction on how I is typed.

Case T-SEQ. We know $I = I_1; I_2$, and

$$\mathtt{H} \vdash I_1 : \langle \Upsilon, Sym\rangle \to \langle \Upsilon'', Sym''\rangle; T_1$$

$$\mathtt{H} \vdash I_2 : \langle \Upsilon'', Sym'' \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T_2$$

where $T = T_1@T_2$ Based on the discussion above, we know

$$I_1 \vdash \langle R, S, M, 0 \rangle \rightarrow_{t_1} \langle R'', S'', M'', |I_1| \rangle$$

$$I_2 \vdash \langle R'', S'', M', 0 \rangle \rightarrow_{t_2} \langle R', S', M', |I_2| \rangle$$

where $t = t_1@t_2$. Therefore, by induction assumption, we know

$$\langle S_D, M_D, T_1 \rangle \rightarrow_{t_1} \langle S''_D, M''_D \rangle$$

$$\langle S''_D, M''_D, T_2 \rangle \rightarrow_{t_1} \langle S'_D, M'_D \rangle$$

Therefore, we have

$$\langle S_D, M_D, T_1@T_2 \rangle \rightarrow_{t_1@t_2} \langle S'_D, M'_D \rangle$$

Case T-LOAD. $I = \textbf{ldb}\ k \leftarrow l[r]$. If $l = D$, then $T = \textbf{read}(l, k, sv)$, where $sv = Sym(r)$. Suppose $b = M(l, n)$, where $n = R(r)$. By assumption 2, we know $(S, sv) \Downarrow n$, therefore we know $(S_D, sv) \Downarrow n$ hold true as well. Further, $S' = S[k \mapsto (b, (l, n))]$, and thus $S'_D = S_D[k \mapsto (b, (l, n))]$. We also know $M'_D = M_D$, and $t = \textbf{read}(n, b)$. In conclusion, we have

$$\langle S_D, M_D, T \rangle \rightarrow_t \langle S'_D, M'_D \rangle$$

If $l = E$, then we know $T = \textbf{read}(l, k, sv)$ as well, where $sv = Sym(r)$. By

assumption 2, we have $(S_D, sv) \Downarrow n$, where $n = R(r)$. Combining with $t = \mathbf{eread}(n)$, $M'_D = M_D$, and $S'_D = S_D[k \mapsto (\star, (l, n))]$, we get our conclusion.

If $l \in \mathbf{ORAMBanks}$, then we know $T = l$, and $t = l$. Combining with $M'_D = M_D$, and $S'_D = S_D$, , the conclusion is trivial.

Case T-STORE. $I = \mathbf{stb}\ k$. Suppose $S(k) = (b, (l, n))$. If $l = D$ or $l = E$, then $S_D(k) = (b, (l, n))$ (where $b = \star$ if $l = E$), $T = \mathbf{write}(l, k, sv)$, $t = \mathbf{write}(n, b)$, and $M'_D = M_D[(l, n) \mapsto b]$. Therefore, we can get our conclusion.

If $l \in \mathbf{ORAMBanks}$, similar to the T-LOAD case, we can get our conclusion.

Case T-STOREW. $I = \mathbf{stw}\ r_1 \to k[r_2]$. Since $\ell = \mathtt{H}$, we know $slab(k) = \mathtt{H}$, which implies $k = E$ or $k \in \mathbf{ORAMBanks}$. Therefore $S'_D = S_D$ and $M'_D = M_D$. Further, since $T = \mathbf{F}$ and $t = \mathbf{f}$, we know

$$\langle S_D, M_D, T\rangle \to_t \langle S'_D, M'_D\rangle$$

Case T-LOADW, T-IDB, T-BOP, T-ASSIGN and T-NOP. In all these rules, $T = \mathbf{F}$, and $t = \mathbf{f}$. Further, it is easy to see that $S' = S$ and $M' = M$ in all these rules. Therefore, the concusion

$$\langle S_D, M_D, T\rangle \to_t \langle S'_D, M'_D, T\rangle$$

holds true.

Case T-UP. The conclusion is trivial by induction assumption.

Case T-LOOP. This is impossible, since T-LOOP requires $\ell = \mathtt{L}$.

Case T-IF. $I = \iota_1; I_t; \iota_2; I_f$, where $\iota_1 = \mathbf{br}\ r_1\ rop\ r_2 \hookrightarrow n_1$ and $\iota_2 = \mathbf{jmp}\ n_2$. Depending on the value of $R(r_1)$ and $R(r_2)$, it may jumps to one of the two branches. If the true branch is taken, then we know $t = \mathbf{f}@t_1@\mathbf{f}$, where

$$I_t \vdash \langle R, S, M, 0\rangle \rightarrow_{t_1} R', S', M', |I_t|\rangle$$

Since I_t is typable, we know, by induction assumption,

$$\langle S_D, M_D, T_t\rangle \rightarrow_{t_1} \langle S'_D, M'_D\rangle$$

Since

$$\langle S_D, M_D, \mathbf{F}\rangle \rightarrow_{\mathbf{f}} \langle S_D, M_D\rangle$$

$$\langle S'_D, M'_D, \mathbf{F}\rangle \rightarrow_{\mathbf{f}} \langle S'_D, M'_D\rangle$$

and $T = \mathbf{F}@t_1@\mathbf{F}$, we can derive our conclusion.

If the false branch is taken, then we know $t = \mathbf{f}@t_2$, where

$$I_f \vdash (R, S, M, 0) \rightarrow_{t_1} (R', S', M', |I_f|)$$

Therefore, we know

$$\langle S_D, M_D, T_f\rangle \rightarrow_{t_2} \langle S'_D, M'_D\rangle$$

Further, since $T_1@\mathbf{F} \equiv T_2$, by Lemma 17, we have $t_1@\mathbf{f} \equiv t_2$, which implies

$$\mathbf{f}@t_1@\mathbf{f} \equiv \mathbf{f}@t_2$$

Therefore, we have

$$\langle S_D, M_D, T\rangle \rightarrow_{\mathbf{f}@t_2} \langle S'_D, M'_D\rangle$$

□

Now, we get back to prove Theorem 2 for T-IF rule. Let us remind that $I = \iota_1; I_t; \iota_2; I_f$.

W.L.O.G, we suppose

$$I \vdash (R_1, S_1, M_1, 0) \rightarrow_{\mathbf{f}} (R_1, S_1, M_1, 1)$$

$$I_t \vdash (R_1, S_1, M_1, 0) \rightarrow_{t_t} (R'_1, S'_1, M'_1, |I_t|)$$

$$I \vdash (R'_1, S'_1, M'_1, |I_t| + 1) \rightarrow_{\mathbf{f}} (R'_1, S'_1, M'_1, |I|)$$

and

$$I \vdash (R_2, S_2, M_2, 0) \rightarrow_{\mathbf{f}} (R_2, S_2, M_2, |I_t| + 2)$$

$$I_f \vdash (R_2, S_2, M_2, 0) \rightarrow_{t_f} (R'_2, S'_2, M'_2, |I_f|)$$

By Lemma 18, we know

$$\langle S_{1D}, M_{1D}, T_1\rangle \rightarrow_{t_t} \langle S'_{1D}, M'_{1D}\rangle$$

$$\langle S_{2D}, M_{2D}, T_2\rangle \rightarrow_{t_f} \langle S'_{2D}, M'_{1D}\rangle$$

Since $\langle S'_{1D}, M'_{1D}, \mathbf{F}\rangle \rightarrow_{\mathbf{f}} \langle S'_{1D}, M'_{1D}\rangle$, we have

$$\langle S_{1D}, M_{1D}, T_1@\mathbf{F}\rangle \rightarrow_{t_t@\mathbf{f}} \langle S'_{1D}, M'_{1D}\rangle$$

Further, by assumption 1 and 2, we know $S_{1D} = S_{2D}$ and $M_{1D} = M_{2D}$. By Lemma 17, we know

$$t_1@\mathbf{f} \equiv t_2$$

$$S'_{1D} = S'_{2D}$$

$$M'_{1D} = M'_{2D}$$

Therefore, we have $t_1 = \mathbf{f}@t_t@\mathbf{f} \equiv \mathbf{f}@t_f = t_2$, which is conclusion 6. Further, the last two assertions show that conclusions 1 and 3 are true. So we only need to prove conclusions 2, 4, and 5.

The first step is to prove conclusion 4. If $\vdash_{safe} Sym(r)$ is not true, then conclusion 4 is vacuum. Otherwise, since $\ell' = H$, we know either $\ell = \mathtt{H}$ or $\vdash_{const} Sym(r)$. In either case, assumption 4, we know $(S_i, Sym(r)) \Downarrow R_i(r)$ for $i = 1, 2$. Then, by induction, we conclude that conclusion 4 holds true: if $(\ell' = H\vee \vdash_{const} Sym'(r))\wedge \vdash_{safe} Sym'(r)$, then $(S'_i, Sym'(r)) \Downarrow R'_i(r)$ $(i = 1, 2)$. Since $(\ell = \mathtt{H}\vee \vdash_{const} Sym'(r))\wedge \vdash_{safe} Sym'(r)$ implies $(\ell' = \mathtt{H}\vee \vdash_{const} Sym'(r))\wedge \vdash_{safe} Sym'(r)$, we know conclusion 4 is true. We can prove conclusion 5 similarly.

The next step is to prove conclusion 2, suppose $\Upsilon(r) = L$, then by T-IF,

(since $\ell' = H$) we know $\vdash_{safe} Sym(r)$ and $\vdash_{const} Sym(k)$. Therefore, by conclusion 4, we know $(S'_{1D}, Sym(r)) \Downarrow R_1(r)$, and $(S'_{2D}, Sym(r)) \Downarrow R_2(r)$. However, since $S'_{1D} = S'_{2D}$, we know $R_1(r) = R_2(r)$. This means conclusion 4 implies conclusion 2.

□

Theorem 7. *Given a program I in $\mathcal{L}_T$, such that $\ell \vdash I : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T$, two memories M_1, M_2, two register mapping R_1, R_2, and two scratchpad mapping S_1, S_2, if the following assumptions are satisfied:*

1. $M_1 \sim_L M_2$
2. $\Upsilon \vdash R_1 \sim R_2$
3. $\Upsilon \vdash S_1 \sim S_2$
4. $\forall r \in \textbf{\textit{Registers}}, i \in \{1, 2\}.(\ell = H \vee \vdash_{const} Sym(r)) \wedge \vdash_{ok} Sym(r) \Rightarrow (S_i, Sym(r)) \Downarrow R_i(r)$;
5. $\forall k \in \textbf{\textit{BlockIDs}}, i \in \{1, 2\}.(\ell = H \vee \vdash_{const} Sym(r)) \wedge \vdash_{ok} Sym(k) \Rightarrow \exists n.(S_i, Sym(k)) \Downarrow n \wedge |\, n \ \textbf{\textit{mod}}\ size(\Upsilon(k))\,| = idx(S_i(k))$.

and

$$I \vdash (R_1, S_1, M_1, 0) \to_{t_1} (R'_1, S'_1, M'_1, pc')$$

$$I \vdash (R_2, S_2, M_2, 0) \to_{t_2} (R'_2, S'_2, M'_2, pc'')$$

If $|t_1| = |t_2|$, then we have the following conclusions:

1. $M'_1 \sim_L M'_2$

2. $\Upsilon' \vdash R'_1 \sim R'_2$

3. $\Upsilon' \vdash S'_1 \sim S'_2$

4. $\forall r \in \textbf{\textit{Registers}}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym(r)) \wedge \vdash_{safe} Sym'(r) \Rightarrow (S'_i, Sym'(r)) \Downarrow R'_i(r)$;

5. $\forall k \in \textbf{\textit{BlockIDs}}, i \in \{1,2\}.(\ell = H \vee \vdash_{const} Sym(r)) \wedge \vdash_{safe} Sym'(k) \Rightarrow \exists n.(S'_i, Sym'(k)) \Downarrow n \wedge |\, n \ \textbf{\textit{mod}}\ size(\Upsilon'(k))\,| = idx(S'_i(k))$.

6. $t_1 \equiv t_2$

Proof. Next, we prove the non-terminating case. Again, we suppose

$$\ell \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T$$

and

$$I \vdash \langle R_1, S_1, M_1, 0 \rangle \rightarrow_{t_1} \langle R'_1, S'_1, M'_1, pc_1 \rangle$$

$$I \vdash \langle R_2, S_2, M_2, 0 \rangle \rightarrow_{t_2} \langle R'_2, S'_2, M'_2, pc_2 \rangle$$

where $|t_1| = |t_2|$. Then we prove by induction on how to derive $\mathtt{L} \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T$. If it is derived by applying one of rules T-LOAD, T-STORE, T-LOADW, T-STOREW, T-BOP, T-ASSIGN, T-NOP, then we know $|t_1| = |t_2| = 1$, and the conclusion follows directly from Theorem 6. If it is derived by applying rule T-UP, then the conclusion is trivial. So we only need to consider rule T-IF, T-LOOP, and T-SEQ.

Case T-SEQ. Suppose $I = I_1; I_2$. We prove by contradiction. With out loss of generality, we suppose $|t_1| = |t_2|$ is minimal such that $t_1 \not\equiv t_2$ or $M_1' \not\sim_L M_2'$. There are two sub cases:

$$I_1 \vdash (R_1, S_1, M_1, 0) \rightarrow_{t_1'} (R_1'', S_1'', M_1'', |I_1|)$$

$$I_2 \vdash (R_1'', S_1'', M_1'', 0) \rightarrow_{t_1''} (R_1', S_1', M_1', pc_1 - |I_1|)$$

where $pc_1 > |I_1|$ and $t_1 = t_1'@t_1''$, Or

$$I_1 \vdash (R_1, S_1, M_1, 0) \rightarrow_{t_1} (R_1'', S_1'', M_1'', pc_1)$$

In the first case, by assuming $|t_1|$ is minimal, we know

$$I_1 \vdash (R_2, S_2, M_2, 0) \rightarrow_{t_2'} (R_2', S_2', M_2'', |I_2|)$$

$$I_2 \vdash (R_2', S_2', M_2'', 0) \rightarrow_{t_2''} (R_2, S_2, M_2', pc_2 - |I_1|)$$

where $pc_2 > |I_1|$. Then by induction assumption, we can prove that $t_1' \equiv t_2'$ and $t_1'' \equiv t_2''$, and thus $t_1 \equiv t_2$. In the second case, by induction assumption, we directly prove that $t_1 \equiv t_2$.

Case T-LOOP. Suppose $I = I_c; \iota_1; I_b; \iota_2$. Since I_c is typable, by Lemma 16, we know that By Lemma 16, we know either one of the following two cases happens:

$$I_c \vdash (R_1, S_1, M_1, 0) \rightarrow_{t_1^1} (R_1'', S_1'', M_1'', |I_c|)$$

$$I \vdash (R_1'', S_1'', M_1'', |I_c|) \rightarrow_{t_1^2} (R_1', S_1', M_1', pc')$$

or

$$I_c \vdash (R_1, S_1, M_1, 0) \rightarrow_{t_1} (R_1, S_1, M_1', pc')$$

where $pc' \leq |I_c|$.

In the latter case, the conclusion follows by induction assumption. For the former case, we know

$$I_c \vdash (R_1, S_1, M_1, 0) \rightarrow_{t_1^1} (R_1'', S_1'', M_1'', |I_c|)$$

$$I \vdash (R_1'', S_1'', M_1'', |I_c|) \rightarrow_{t_1^2} (R_1', S_1', M_1', pc')$$

Similarly, we have

$$I_c \vdash (R_2, S_2, M_2, 0) \rightarrow_{t_2^1} (R_2'', S_2'', M_2'', |I_c|)$$

$$I \vdash (R_2'', S_2'', M_2'', |I_c|) \rightarrow_{t_2^2} (R_2', S_2', M_2', pc'')$$

By Theorem 6, we have

1. $M_1'' \sim_L M_2''$;
2. $\Upsilon \vdash R_1 \sim R_2$;
3. $\Upsilon \vdash S_1 \sim S_2$;
4. $\forall r \in \textbf{Registers}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym'(r)) \wedge \vdash_{safe} Sym'(r) \Rightarrow$

$(S_i'', Sym''(r)) \Downarrow R_i'(r)$;

5. $\forall k \in \mathbf{BlockIDs}, i \in \{1,2\}.(\ell = H \wedge \vdash_{const} Sym'(r)) \wedge \vdash_{safe} Sym'(k) \Rightarrow \exists n.(S_i'', Sym'(k)) \Downarrow n \wedge \mid n\Upsilon''(k) \models idx(S_i''(k))$.

6. $t_1^1 \equiv t_2^1$

Further, we know $I(|I_c|) = \iota_1 = \mathbf{br}\ r_1\ rop\ r_2 \hookrightarrow n_1$, where $\Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathtt{L}$, which implies $\Upsilon(r_1) = \Upsilon(r_2) = \mathtt{L}$. Therefore, we know $R_1''(r_1) = R_2''(r_1)$ and $R_1''(r_2) = R_2''(r_2)$, which implies if

$$I \vdash (R_i'', S_i'', M_i'', |I_c|) \rightarrow_{\mathbf{f}} (R_i^\star, S_i^\star, M_i^\star, pc_i^\star)$$

then $pc_1^\star = pc_2^\star$, which is either $|I_c| + 1$ or $|I_c| + n_1 = |I|$. If later, then we know $t_1 = t_1^1@\mathbf{f} \equiv t_2^2@\mathbf{f} = t_2$, and $R_i'' = R_i$, $S_i'' = S_i$, $M_i'' = M_i$ $(i = 1, 2)$. In this case, conclusions 1-6 all hold true.

In the former case, there are still two sub cases: **(1)**

$$I_b \vdash (R_1'', S_1'', M_1'', 0) \rightarrow_{t_1^2} (R_1^3, S_1^3, M_1^3, pc_1)$$

$$I_b \vdash (R_2'', S_2'', M_2'', 0) \rightarrow_{t_2^2} (R_2^3, S_2^3, M_2^3, pc_2)$$

Then by induction, we know $t_1^2 \equiv t_2^2$, and therefore $t_1 \equiv t_2$, and all conclusions 1-5 are true.

(2)

$$I_b \vdash (R_1'', S_1'', M_1'', 0) \rightarrow_{t_1^2} (R_1^3, S_1^3, M_1^3, |I_b|)$$

$$I_b \vdash (R_2'', S_2'', M_2'', 0) \rightarrow_{t_2^2} (R_2^3, S_2^3, M_2^3, |I_b|)$$

$$I \vdash (R_1^3, S_1^3, M_1^3, 0) \rightarrow_{t_1'} (R_1', S_1', M_1', |I_b|)$$

$$I \vdash (R_2^3, S_2^3, M_2^3, 0) \rightarrow_{t_2'} (R_2', S_2', M_2', |I_b|)$$

where $t_i = t_i^1@\mathbf{f}@t_i^2@\mathbf{f}@t_i'$ $(i = 1, 2)$. Similar to Theorem 6, we can show the conclusions 1-6 hold true for this case.

Case T-IF. Suppose $I = \iota_1; I_t; \iota_2; I_f$, where $\iota_1 = \mathbf{br}\ r_1\ rop\ r_2 \hookrightarrow n_1$. We need the following lemma.

Lemma 19. *If* $H \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle : T$, *and*

$$I \vdash (R, S, M, pc) \rightarrow_t (R', S', M', pc')$$

Then $pc' > pc$.

Proof (sketch). Since while-statement cannot be typed in high security context, and while-statement is the only place allowing jumping or branching back, in high security context, the program counter will only increase. □

If $\Upsilon(r_1) = \Upsilon(r_2) = \mathtt{L}$, then we know

$$I \vdash (R_i, S_i, M_i, 0) \rightarrow_{\mathbf{f}} (R_i, S_i, M_i, pc)$$

$$I \vdash (R_i, S_i, M_i, pc) \rightarrow_{t_i'} (R_i', S_i', M_i', pc_i)$$

where $t_i = \mathbf{f}@t_i'$ for $i = 1, 2$. In this case, we can prove the conclusions by induction

assumption.

If $\ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) = \mathtt{H}$, then by Lemma 19, we know

$$I \vdash (R_i, S_i, M_i, 0) \rightarrow_{t_i'} (R_i, S_i, M_i, |I|)$$

for $i = 1, 2$, where t_1 and t_2 are prefixes of t_1' and t_2'. By Theorem 6, we know $t_1' \equiv t_2'$. Therefore, we know $t_1 \equiv t_2$. Conclusion 1-5 can be proven similar to Theorem 6.

□

Proof of Theorem 2. Suppose I is a program, Υ and Sym satisfy: (1) $\forall r.Sym(r) = ? \wedge \Upsilon(r) = L$; and (2) $\forall k.Sym(k) = ? \wedge \Upsilon(k) = D$. There are Υ' and Sym', such that

$$\ell \vdash I : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T$$

where $\ell = \mathtt{L}$, and

$$I \vdash (R_0, S_0, M_1, 0) \rightarrow_{t_1} (R_1, S_1, M_1', pc_1)$$

$$I \vdash (R_0, S_0, M_2, 0) \rightarrow_{t_2} (R_2, S_2, M_2', pc_2)$$

where $M_1 \sim_L M_2$, $\forall r.R_0(r) = 0$, $\forall k.S_0(k) = (b_0, (D, 0))$, and $|t_1| = |t_2|$. Clearly, we have

1. $M_1 \sim_L M_2$
2. $\Upsilon \vdash R_0 \sim R_0$;
3. $\Upsilon \vdash S_0 \sim S_0$;

4. $\forall r \in \textbf{Registers}, k \in \textbf{Blocks}. \not\forall_{safe}\ Sym(r) \wedge \not\forall_{safe}\ Sym(k)$;

Therefore, by applying Theorem 7, we have $t_1 \equiv t_2$, and $M_1' \sim_L M_2'$. □

Appendix C: Proof of Theorem 3

We begin by discussing how to construct sim_A; the simulator sim_B is constructed similarly.

Since Alice does not have the view of Bob's local data, and those data secret-shared between them two, we define a special notion $\bullet$ as the values not observable to Alice. We define the operations on top of $\bullet$ as follows:

$$\bullet \; op \; v = \bullet \quad v \; op \; \bullet = \bullet \quad \bullet \, (v) = \bullet \quad m(\bullet) = \bullet$$

We define the following auxiliary functions accordingly:

$$\begin{array}{rcl}
(select_A(l,t,t'), select_B(l,t,t')) & := & select(l,t,t') \\
read_A(l,v) & := & \begin{cases} v & l \sqsubseteq \mathtt{A} \\ \bullet & \text{otherwise} \end{cases} \\
val(v,l) & := & v \\
val(m,l) & := & m \\
lab(v,l) & := & l \\
lab(m,l) & := & l
\end{array}$$

We then define Alice's snapshot of a memory M, denoted as $M \downarrow A$, in the following:

Definition 11. *Given a memory M, Alice's snapshot of M, denoted as $M \downarrow A$, is defined as a memory such that*

$$M{\downarrow}A(x) = \begin{cases} M(x) & \text{if } M(x) = (v, l) \text{ where } l \sqsubseteq \mathtt{A} \\ \bullet & \text{otherwise} \end{cases}$$

We further define the Alice-similarity property of two memories as follows:

Definition 12. *We say two memories M_1 and M_2 are Alice-similar, denoted as $M_1 \sim_A M_2$, if and only if $M_1{\downarrow}A = M_2{\downarrow}A$.*

Figure C.1 defines how sim_A evaluate an expression. The judgement in the form of $l \vdash_A \langle M, e\rangle \Downarrow_t v$ says that given a memory M, the simulator sim_A evaluates an expression e to value v, producing memory trace t.

Figure C.2 and Figure C.3 defines how sim_A simulates the instruction- and memory-traces until the next declassification. The judgement $\langle M_i, S_i\rangle \xrightarrow{(i,t)}_A \langle M_i', S_i'\rangle$ says that given a statement S_i and a memory M, sim_A evaluates the program S_i over memory M_i and reduces to program S_i' and memory M_i' emitting Alice's instruction trace i and memory trace t.

The judgement $\langle M_i, S_i\rangle \xrightarrow{(i,t)}{}^{\star}_A \langle M_i', S_i'\rangle$ is similar to $\langle M_i, S_i\rangle \xrightarrow{(i,t)}_A \langle M_i', S_i'\rangle$, but requires the last statement evaluated must be a declassification statement. We emphasize that our rules enforce that the memory over which the program is evaluated must be Γ-compatible.

$$\boxed{l \vdash_A \langle M, e\rangle \Downarrow_{t_a} v}$$

$$\text{Sim-E-Const}\quad l \vdash_A \langle M, n\rangle \Downarrow_{\epsilon} n$$

$$\text{Sim-E-Var}\frac{\begin{array}{c} lab(M(x)) = l' \qquad v = read_A(l, val(M(x))) \\ t = select_A(l, \mathbf{read}(x, v), x)\end{array}}{l \vdash_A \langle M, x\rangle \Downarrow_t v}$$

$$\text{Sim-E-Op}\frac{\begin{array}{c} lab(M(x_i)) = l_i \qquad l \vdash_A \langle M, x_i\rangle \Downarrow_{t_i} v_i \\ t = t_1@t_2 \qquad v = v_1\ op\ v_2 \qquad i = 1, 2\end{array}}{l \vdash_A \langle M, x_1\ op\ x_2\rangle \Downarrow_t v}$$

$$\text{Sim-E-Array}\frac{\begin{array}{c} lab(M(y)) = l' \qquad l \vdash_A \langle M, y\rangle \Downarrow_{t_1} v \\ lab(M(x)) = l'' \qquad v_1 = val(M(x)) \\ t_2 = select_A(l, \mathbf{readarr}(x, v_1, v), x) \\ t = t_1@t_2 \qquad v_2 = read_A(l, val(M(x))(v))\end{array}}{l \vdash_A \langle M, x[y]\rangle \Downarrow_t v_2}$$

$$\text{Sim-E-Mux}\frac{\begin{array}{cc} lab(M(x_i)) = \mathbf{Nat}\ l_i \qquad l \vdash_A \langle M, x_i\rangle \Downarrow_{t_i} v_i \qquad i = 1, 2, 3 \\ v_1 = 0 \Rightarrow v = v_2 & v_1 \neq 0 \Rightarrow v = v_3 \\ v_1 = \bullet \Rightarrow v = \bullet & t = t_1@t_2@t_3\end{array}}{l \vdash_A \langle M, \mathbf{mux}(x_1, x_2, x_3)\rangle \Downarrow_t v}$$

Figure C.1: Operational semantics for sim_A

The simulator $sim_A(M, S, D_1, ..., D_n)$ runs as follows. Initially set M_1 to be M and S_1 to be S. For each $i = 1, ..., n$, sim_A evaluates $\langle M_i, S_i\rangle \xrightarrow{(i,t)}{}^{\star}_A \langle M_i', S_i'\rangle$. If $D_i = \epsilon$, then set M_{i+1} to be M_i'; otherwise, $D_i = (x, v)$, set M_{i+1} to be $M_i'[x \mapsto v]$. Finally, sim_A evaluates $\langle M_n, S_i\rangle \xrightarrow{(i,t)}{}^{\star}_A \langle M', S'\rangle$, and returns (i, t).

The following lemma shows that the semantics for sim_A generates the same memory trace as the semantics for SCVM.

Lemma 20. *If $l \vdash \langle M, e\rangle \Downarrow_{(t_a,t_b)} v$ and $\Gamma \vdash e : \textbf{\textit{Nat}}\ l'$ and $l \vdash_A \langle M', e\rangle \Downarrow_t v'$ and $M \sim_A M'$, and $l' \sqsubseteq l$, and M and M' are Γ-compatible, then $t_a \equiv t$ and if $l \sqsubseteq \mathtt{A}$, then $v = v'$. Otherwise $v' = \bullet$.*

Proof. Prove by structural induction on e. If $e = x$, then $\Gamma(x) = \mathbf{Nat}\ l'$. If $l \sqsubseteq \mathtt{A}$,

$$\boxed{\langle M, S\rangle \xrightarrow{(i,t)}{}^{\star}_{A} \langle M', S'\rangle}$$

$$\text{Sim-Declass}\frac{t = y \qquad i = \mathbf{declass}(x, y)}{\langle M, \mathtt{O} : x := \mathbf{declass}_l(y)\rangle \xrightarrow{(i,t)}{}^{\star}_{A} \langle M, \mathtt{O} : \mathbf{skip}\rangle}$$

$$\text{Sim-Seq}\frac{\langle M, S_1\rangle \xrightarrow{(i,t)}{}^{\star}_{A} \langle M', S_1'\rangle}{\langle M, S_1; S_2\rangle \xrightarrow{(i,t)}{}^{\star}_{A} \langle M', S_1'; S_2\rangle} \qquad \text{Sim-Concat}\frac{\begin{array}{c}\langle M, S, \epsilon\rangle \xrightarrow{(i,t)}_{A} \langle M', S', \epsilon\rangle \\ \langle M', S'\rangle \xrightarrow{(i',t')}{}^{\star}_{A} \langle M'', S''\rangle\end{array}}{\langle M, S\rangle \xrightarrow{(i@i',t@t')}{}^{\star}_{A} \langle M'', S''\rangle}$$

Figure C.2: Operational semantics for statements in sim_A (part 1)

then $v' = val(M'(x)) = val(M(x)) = v$, therefore $v = v'$. Further $t = \mathbf{read}(x, v') = \mathbf{read}(x, v) = t_a$ if $l \sqsubseteq \mathtt{A}$. If $l = \mathtt{B}$, then $v' = \bullet$, and $t = \epsilon = t_a$. If $l = \mathtt{O}$, then $v' = \bullet$, and $t = x = t_a$.

If $e = n$, then $t = \epsilon = t_a$, and $v' = n = v$, and $l = \mathtt{P} \sqsubseteq \mathtt{A}$.

If $e = x_1\ op\ x_2$. Then we know $l \vdash_A \langle M', x_i\rangle \Downarrow_{t_i} v_i'$, and $\langle M, x_i\rangle \Downarrow_{(t_a^i, t_b^i)} v_i$ for $i = 1, 2$. By induction assumption, we know $t_i \equiv t_a^i$, and thus $t = t_1@t_2 \equiv t_a^1@t_a^2 = t_a$. For its value, suppose $\Gamma(x_i) = \mathbf{Nat}\ l_i$, $i = 1, 2$, if $l \sqsubseteq \mathtt{A}$, then $l_i \sqsubseteq \mathtt{A}$ holds true, and by induction assumption, we know $v_i = v_i'$ for $i = 1, 2$, and thus $v = v_1\ op\ v_2 = v_1'\ op\ v_2 = v'$. Otherwise, either or both v_1 and v_2 are $\bullet$, and thus we know $v' = \bullet$.

If $e = x[y]$. We first reason about the value. If $l \sqsubseteq \mathtt{A}$, then suppose $\Gamma(y) = \mathbf{Nat}\ l''$, then $l'' \sqsubseteq l' \sqsubseteq l \sqsubseteq \mathtt{A}$ according to $\Gamma \vdash x[y] : \mathbf{Nat}\ l'$. Then we know $v_1' = val(M'(y)) = val(M(y)) = v_1$. Further, we know $(m', l) = M'(x) = M(x) = (m, l)$, and thus $v' = get(m', v_1') = get(m, v_1) = v$. If $l \not\sqsubseteq \mathtt{A}$, then $v = \bullet$.

Then we reason about the trace. If $l \sqsubseteq \mathtt{A}$, then

$$t = \mathbf{read}(y, v_1)@\mathbf{readarr}(x, v_1, v) \equiv \mathbf{read}(y, v_1')@\mathbf{readarr}(x, v_1, v') = t_a$$

If $l = \mathtt{B}$, we have $t \equiv \epsilon \equiv t_a$. If $l = \mathtt{O}$, we have $t \equiv y@x \equiv t_a$.

For $e = \mathbf{mux}(x_1, x_2, x_3)$, based on a very similar argument as for x_1 *op* x_2, we can get the conclusion. □

The following lemma further claims that if an expression has a type $\mathtt{B}$, then simulating it will generate no observable instruction traces and memory traces to Alice.

Lemma 21. *If* $\Gamma \vdash e : \textbf{\textit{Nat}}\ l'$, *and* $\mathtt{B} \vdash \langle M, e\rangle \Downarrow_t v$, *and* M *is* Γ*-compatible then* $t \equiv \epsilon$.

Proof. Prove by structure induction on e. If $e = x$, then $t = \epsilon$ by rule Sim-E-Var.

If $e = x_1$ *op* x_2. Suppose $\Gamma \vdash x_i : \mathbf{Nat}\ l_i$ for $i = 1, 2$, then we know $l_i \sqsubseteq \mathtt{B}$. Therefore $t_i \equiv \epsilon$, and thus $t \equiv \epsilon$.

If $e = x[y]$, the conclusion follows the fact that $\mathtt{B} \vdash \langle M, y\rangle \Downarrow_\epsilon v$, and $select_A(\mathtt{B}, \mathbf{readarr}(x, v_1, v), x) = \epsilon$.

If $e = \mathbf{mux}(x_1, x_2, x_3)$, similar to binary operation, we know $t \equiv \epsilon$. □

Lemma 22. *If* $\mathtt{B} \vdash S$, *and* $\langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M', S'\rangle$, *where* M *is* Γ*-compatible, then* $i_a \equiv \epsilon$, $t_a \equiv \epsilon$, *and* $M \sim_A M'$

Proof. We prove by induction on the structure of S. If $S = l : \mathbf{skip}$, then the conclusion is trivial.

If $S = l : x := e$, then we know $l = \mathtt{B}$ and $\Gamma(x) = \mathbf{Nat}\ \mathtt{B}$. Then $i_a = \epsilon$ and $M' \sim_A M$ follow trivially. According to Lemma 21, we can prove $t_a \equiv \epsilon$.

If $S = \mathtt{O} : x := \mathbf{oram}(y)$ or $S = \mathtt{O} : x := \mathbf{declass}_l(y)$, then pc is required to be $\mathtt{P}$, so that the conclusion is vacuous.

If $S = l : y[x_1] := x_2$, then $l = \mathtt{B}$, and thus $i_a = \epsilon$. Therefore $t_a = t_{1a}@t_{2a}@t'_a$, where $\mathtt{B} \vdash \langle M, x_i\rangle \Downarrow_{(t_{ia},t_{ib})} v_i$ for $i = 1, 2$, and $t'_a = select_A(\mathtt{B}, \mathbf{writearr}(y, v_1, v_2), y) = \epsilon$. Therefore $t_{1a} \equiv t_{2a} \equiv \epsilon$ according to Lemma 21. In conclusion, we have $t_a = t_{1a}@t_{2a}@t'_a \equiv \epsilon$. Finally, for memory, $M' = M[y \mapsto m'] \sim_A M$.

If $S = l : \mathbf{if}(x)\mathbf{then}\ S_1\ \mathbf{else}\ S_2$, then $l = \mathtt{B}$, and $\Gamma, \mathtt{B} \vdash S_i$ for $i = 1, 2$. Suppose $M(x) = (v, \mathtt{B})$, then $\langle M, S\rangle \xrightarrow{(\epsilon,\epsilon,i'_b,t'_b)} \langle M, S_c\rangle$, where $v = 1 \Rightarrow c = 1$ and $v \neq 1 \Rightarrow c = 2$. There are two cases: (1) $M' = M$ and $S = S_c$, then the conclusion is trivial; (2) $\langle M, S_c\rangle \xrightarrow{(i''_a,t''_a,i''_b,t''_b)} \langle M', S'\rangle$. In this case, by induction assumption, we have $M' \sim_A M$, $i''_a \equiv \epsilon$ and $t''_a \equiv \epsilon$, so that $i_a = \epsilon@i''_a \equiv \epsilon$ and $t_a = \epsilon@t''_a \equiv \epsilon$.

For $S = l : \mathbf{while}(x)\mathbf{do}\ S'$, the conclusion can be proven similarly to the if-case.

Finally, for $S = S_1; S_2$, we know either (1) $\langle S_1, M\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle S'_1, M'\rangle$; or (2) $\langle S_1, M\rangle \xrightarrow{(i'_a,t'_a,i'_b,t'_b)} \langle l : \mathbf{skip}, M''\rangle$ and $\langle S_2, M''\rangle \xrightarrow{(i''_a,t''_a,i''_b,t''_b)} \langle S', M'\rangle$, where $i_a = i'_a@i''_a$, and $t_a = t'_a@t''_a$. In both cases, the conclusions can be proven easily. □

The following lemma is the main lemma saying that when evaluating over Alice-similar memories, sim_A and SCVM will generate the same instruction traces and memory traces, and produce Alice-similar memory profiles.

Lemma 23. *If* $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M_1, S'\rangle : D$ *where* $\Gamma, \mathtt{P} \vdash S$, *and* $M \sim_A M'$,

and $\langle M', S\rangle \xrightarrow{(i,t)}_A \langle M_1', S''\rangle$ *(for* $D = \epsilon$*) or* $\langle M', S\rangle \xrightarrow{(i,t)}{}^{\star}_A \langle M_1', S''\rangle$ *(for* $D \neq \epsilon$*), then* $S' = S''$*,* $i_a \equiv i$ *and* $t_a \equiv t$*. If* $D = \epsilon$*, then* $M_1 \sim_A M_1'$*; otherwise, suppose* $D = (x, v)$*, then* $M_1 = M_1'[x \mapsto v]$*.*

Proof. The conclusion $S' = S''$ can be trivially done by examining the correspondence of each E- and S- rules and Sim- rules. Therefore, we only prove (1) $M_1 \sim_A M_1'$, (2) $i_a \equiv i$, and (3) $t_a \equiv t$.

We prove by induction on the length of steps L toward generating declassification event D. If $L = 0$, then we know $S = \mathtt{O} : x := \mathbf{declass}_l(y); S_2$ (or $\mathtt{O} : x := \mathbf{declass}_l(y)$). Since we assume $\Gamma, \mathtt{P} \vdash S$, by typing rule T-Declass, we have $l \neq \mathtt{O}$, $\Gamma(x) = \mathbf{Nat}\ l$. If $l \sqsubseteq \mathtt{A}$, then $D[A] = (x, v)$, and thus $M_1'[x \mapsto v] \sim_A M[x \mapsto v] = M_1$. Further, we know $i_a = \mathbf{declass}(x, y) = i$, and $t_a = y = t$. Second, if $l_x = \mathbf{B}$, then $M_1' = M' \sim_A M = M_1$, $i_a = \mathbf{declass}(x, y) = i$, and $t_a = y = t$.

We next consider $L > 0$, then $S = S_1; S_2$. Since $(S_a; S_b); S_c$ is equivalent to $S_a; (S_b; S_c)$ in the sense that if $\langle M, (S_a; S_b); S_c\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : D$, then $\langle M, S_a; (S_b; S_c)\rangle \xrightarrow{(i_a',t_a',i_b',t_b')} \langle M', S'\rangle : D$, where $i_a \equiv i_a'$, $i_b \equiv i_b'$, $t_a \equiv t_a'$, and $t_b \equiv t_b'$. Therefore we only consider S_1 not to be a Seq statement, then we know $S_1 = l : s_1$. By taking one step, we only need to prove claims (1)-(3), then the conclusion can be shown by induction assumption. In the following, we consider how this step is executed.

Case l : **skip.** If $S_1 = l$: **skip**, the conclusion is trivial, i.e. $i_a = \epsilon = i$ and $t_a = \epsilon = t$ and $M_1' = M' \sim_A M = M_1$.

Case $l : x := e$**.** If $S_1 = l : x := e$, $i_a = l : x := e = i$. Then we show $t \equiv t_a$.

If $l \sqsubseteq \mathtt{A}$, $t \equiv t_a$ directly follows Lemma 20. If $l = \mathtt{B}$, then by Lemma 21, we have $t \equiv \epsilon \equiv t_b$. If $l = \mathtt{O}$, then we consider e separately. If $e = y$, then $t = y@x = t_a$. If $e = y[z]$, then $t = z@y@x = t_a$. If $e = n$, then $t = x = t_a$. If $e = y\ op\ z$, then $t = y@z@x = t_a$. Finally, if $e = \mathbf{mux}(x_1, x_2, x_3)$, then $t = x_1@x_2@x_3@x = t_a$.

Finally, we prove the memory equivalence. If $l \sqsubseteq \mathtt{A}$, then according to Lemma 20, e evaluates to the same value v in the semantics, and in the simulator. Therefore $M_1' = M'[x \mapsto v] \sim_A M[x \mapsto v] = M_1$. If $\mathtt{B} \sqsubseteq l$, then $M_1' = M' \sim_A M \sim_A M[x \mapsto v] = M_1$. Therefore, the conclusion is true.

Case $\mathtt{O} : x := \mathbf{oram}(y)$**.** It is easy to see that $M_1' = M' \sim_A M \sim_A M[x \mapsto m] = M_1$, and $i = \mathtt{O} : \mathbf{init}(x, y) = i_a$. Suppose $\Gamma(y) = \mathbf{Nat}\ l$, then we know $l \neq \mathtt{O}$. If $l \sqsubseteq \mathtt{A}$, then $t = y@x = t_a$. Otherwise, $l = \mathtt{B}$, then we know $t = x = t_a$.

Case $l : y[x_1] := x_2$**.** By typing rule T-ArrAss, we know $\Gamma(y) = \mathbf{Array}\ l$, $\Gamma(x_1) = \mathbf{Nat}\ l_1$, $\Gamma(x_2) = \mathbf{Nat}\ l_2$, where $l_1 \sqsubseteq \mathtt{A}$ and $l_2 \sqsubseteq \mathtt{A}$. If $l \sqsubseteq \mathtt{A}$, then we have $t_a = \mathbf{read}(x_1, v_1)@\mathbf{read}(x_2, v_2)@\mathbf{writearr}(a, v_1, v_2) = t$, and $i_a = l : y[x_1] := x_2 = i$. For memory, $M^{\star\star} = M'[y \mapsto set(m, v_1, v_2)] \sim_A M[y \mapsto set(m, v_1, v_2)] = M^{\star}$, where $(m, l) = M'(y) = M(y)$, $(v_1, l_1) = M(x_1)$, and $(v_2, l_2) = M(x_2)$.

If $l = \mathtt{B}$, then $M_1' = M' \sim_A M \sim_A M[y \mapsto m'] = M_1$, $i = \epsilon = i_a$, $t = \epsilon = t_a$.

Case $l : \mathbf{if}(x)\mathbf{then}\ S_1\mathbf{else}\ S_2$**.** If $l = \mathtt{B}$, then according to Lemma 22, $M_1' = M' \sim_A M \sim_A M_1$, $t \equiv \epsilon t_a$, and $i \equiv \epsilon i_a$. If $l \sqsubseteq \mathtt{A}$, then $i = l : \mathbf{if}(x) = i_a$, and $t = \mathbf{read}(x, v) = t_a$. Further, $M_1' = M' \sim_A M = M_1$. Therefore, the conclusion is also true.

Case $l : \mathbf{while}(x)\mathbf{do}\ S'$**.** For $S_1 = \mathbf{while}(x)\mathbf{do}\ S_b$, the proof is very similar to the if-statement. □

Lemma 23 immediately shows that sim_A can simulate the correct traces. Therefore Theorem 3 holds true. Q.E.D

C.1 Proof of Theorem 4

Theorem 4 is a corollary of the following theorem:

Theorem 8. *If* $\Gamma, pc \vdash S$*, then either* S *is* l : ***skip****, or for any* Γ*-compatible memory* M*, there exist* $i_a, t_a, i_b, t_b, M', S', D$ *such that* $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : D$*,* M' *is* Γ*-compatible, and* $\Gamma, pc \vdash S'$*.*

Proof. We prove by induction on the structure of S. If $S = l$: **skip**, then the conclusion is trivial.

If $S = l : x := e$, then $\Gamma(x) = \mathbf{Nat}\ l$, $\Gamma \vdash e : \mathbf{Nat}\ l'$, $pc \sqcup l' \sqsubseteq l$. We discuss the type of e. If $e = x'$, then we know $\Gamma(x') = \mathbf{Nat}\ l'$. Since M is $\Gamma-$compatible, we know $M(x') = (v, l')$, where $v \in \mathbf{Nat}$. Therefore, $\langle M, x'\rangle \Downarrow_{(t_a,t_b)} v$, where $(t_a, t_b) = select(l, \mathbf{read}(x', v), x')$, and thus $\langle M, S\rangle \xrightarrow{(i_a,t'_a,i_b,t'_b)} \langle M', l : \mathbf{skip}\rangle : \epsilon$, where $(i_a, i_b) = inst(l, x := e)$, $t'_a = t_a @ t''_a$, and $t'_b = t_b @ t''_b$, where $(t''_a, t''_b) = select(l, \mathbf{write}(x, v), x)$. Further, $M' = M[x \mapsto (v, l)]$. Therefore, M' is also Γ-compatible, and the conclusion holds true. Similarly, we can prove that if $\Gamma \vdash e : \tau$ is derived using T-Const, T-Op, T-Array, or T-Mux, then the conclusion is also true.

If $S = \mathtt{0} : x := \mathbf{declass}_l(y)$, then $\Gamma(y) = \mathbf{Nat}\ O$, $\Gamma(x) = \mathbf{Nat}\ l$ where $l \neq \mathbf{O}$, and $pc = \mathbf{P}$. Since M is $\Gamma-$compatible, we know $M(y) = (v, \mathbf{O})$, $M' = M[x \mapsto (v, l)]$. Therefore M' is $\Gamma-$compatible. Further, $t_a = y = t_b$, $i_a = i_b = \mathtt{0}$: $\mathbf{declass}(x, y)$, $D = select(l, (x, v), \epsilon)$, and $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', \mathtt{0} : \mathbf{skip}\rangle$, and we

know that $\Gamma, \mathtt{P} \vdash \mathtt{O} : \textbf{skip}$. Therefore the conclusion is true.

Similarly, we can prove the conclusion is true for $S = O : x := \textbf{oram}(y)$.

For $S = l : y[x_1] := x_2$, then $\Gamma(y) = \textbf{Array}\ l$, $\Gamma(x_1) = \textbf{Nat}\ l_1$, $\Gamma(x_2) = \textbf{Nat}\ l_2$, and $pc \sqcup l_1 \sqcup l_2 \sqsubseteq l$. Since M is $\Gamma-$compatible, we know $M(y) = (m, l)$, $M(x_1) = (v_1, l_1)$, and $M(x_2) = (v_2, l_2)$. Therefore $M' = M[y \mapsto (set(m, v_1, v_2), l)]$ is also $\Gamma-$compatible. Further, $(t'_a, t'_b) = select(l, \textbf{writearr}(y, v_1, v_2), y)$, $t_a = t_{1a}@t_{2a}@t'_a$, $t_b = t_{1b}@t_{2b}@t'_b$, and $(i_a, i_b) = inst(l, y[x_1] := x_2)$. Therefore, $\langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M', l : \textbf{skip}\rangle$, where we can prove $\Gamma, pc \vdash l : \textbf{skip}$ easily. Therefore, the conclusion is true.

For $S = l : \textbf{if}(x)\textbf{then}\ S_1\ \textbf{else}\ S_2$, we have $\Gamma(x) = \textbf{Nat}\ l$. Therefore $M(x) = (v, l)$. If $v = 1$, then $\langle M, S\rangle \xrightarrow{i_a, t_a, i_b, t_b} \langle M, S_1$ where $(i_a, i_b) = inst(l, \textbf{if}(x))$ and $(t_a, t_b) = select(l, \textbf{read}(x, v), x)$. Further, we know $\Gamma, l \vdash S_1$. Since $pc \sqsubseteq l$, it is easy to prove by induction that $\Gamma, pc \vdash S_1$ is true as well. Therefore, the conclusion is true. On the other hand, if $v \neq 1$, then $\langle M, S\rangle \xrightarrow{i_a, t_a, i_b, t_b} \langle M, S_2\rangle$. We can also prove the conclusion.

The proof for $S = l : \textbf{while}(x)\textbf{do}\ S'$ is similar to the branching-statement by using rule S-While-True and S-While-False.

For $S = S_1; S_2$, then we know $\Gamma \vdash S_1$. The conclusion directly follows the induction assumption by applying rule S-Seq and rule S-Skip. $\square$

Appendix D: The hybrid protocol and the proof of Theorem 5

In this section, we present the hybrid protocol, and show it emulates the ideal world functionality $\mathcal{F}$. To start with, we present smaller ideal functionalities in $\mathcal{G}$ used by the hybrid world protocol.

1. $\mathcal{F}_{op}^{(l_1,l_2)}$ are the ideal functionalities for binary operation *op*. They are parameterized by two type labels l_1 and l_2 from $\{\mathtt{P}, \mathtt{A}, \mathtt{B}, \mathtt{O}\}$ indicating which party provides the data to the functionality. Suppose the operation is x *op* y. l_1 and l_2 correspond to x and y respectively. If l_1 is P, then both Alice and Bob will hand in the value of x, and the functionality verifies these two values are the same. If l_1 is $\mathtt{A}$ (or $\mathtt{B}$), then Alice (or Bob) hands in the value of x to the functionality. If l_1 is $\mathtt{O}$, then both Alice and Bob hand in their secret shares to the functionality respectively. The value of l_2 has the same meaning but is for the data source of y. These ideal functionalities output secret shares of the result to Alice and Bob respectively. For example, $\mathcal{F}_{op}^{(\mathtt{P},\mathtt{A})}$ accepts input x, y from Alice, and x from Bob and return the results $[v]_a$ to Alice and $[v]_b$ to Bob. We denote this as $([v]_a, [v]_b) = \mathcal{F}_{op}^{(\mathtt{P},\mathtt{A})}(x@y, x)$.

2. $\mathcal{F}_{\mathtt{mux}}^{(l_1,l_2,l_3)}$ are the ideal functionalities for the multiplex operations. The three parameters l_1, l_2, and l_3 have the same meaning as above, but correspond to

the three input of the multiplex operation. These functionalities also return secret shares to Alice and Bob.

3. $\mathcal{F}^{x}_{\texttt{oram}}$ for each array x is an interactive Oblivious RAM functionality. It supports three operations.

   - $\texttt{init}^{l}$ to initialize the ORAM with a given array. l is from $\{\texttt{P}, \texttt{A}, \texttt{B}\}$. If l is `P` or `A`, then Alice hands in her array. If l is `B`, then Bob hands in his array.
   - `read` to read the content for a given index. The index is provided as a pair of secret shares from Alice and Bob. The output is also a pair of secret shares, which are returned to Alice and Bob respectively.
   - `write` to write a value into a given index. It takes four inputs: the secret shares of the index and the secret shares of the values from Alice and Bob respectively.

4. $\mathcal{F}^{l}_{\texttt{declass}}$ is the declassification function, which takes secret shares from Alice and Bob as its input, and returns the revealed value to the party corresponding to l.

The protocol $\Pi^{\mathcal{G}}$ is then presented in Figure D.1 and Figure D.2. During the protocol's execution, Alice and Bob consumes their instruction traces and memory traces. Since the memory traces contain all information of the public memory and their local memories, both Alice and Bob only store locally their secret shares $[M]_A$ and $[M]_B$ and the instruction- and memory- traces.

Figure D.1 presents the rules for local execution. Since all local and public data to be used in secure computation are contained in memory traces, Alice and Bob do not maintain their local data and public data. The rules are in the form of $(i, t) \rightarrow (\epsilon, \epsilon)$, which means the instruction trace i and memory trace t will be consumed. In each rule, only one local instruction, i.e. the security label $l \neq \mathtt{0}$, and its corresponding memory trace for each instruction will be consumed. It is not hard to verify the following proposition:

Proposition 1. *Assuming* $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : \epsilon$ *and* s *is a statement in the set* $\{x := e, x[x] := x, \textbf{\textit{if}}(x), \textbf{\textit{while}}(x)\}$. *If* $i_a = l : s$, *where* $l \neq \mathtt{0}$, *then* $(i_a, t_a) \rightarrow (\epsilon, \epsilon)$; *If* $i_b = l : s$, *where* $l \neq \mathtt{0}$, *then* $(i_b, t_b) \rightarrow (\epsilon, \epsilon)$.

Note local execution rules only handle executing one instruction. The sequence of multiple instructions are handled using rule H-LocalA, H-LocalB, and H-Seq explained later.

Figure D.2 presents two parts. The first part consists the rules, in the form of $\langle [M]_A, t_a, [M]_B, t_b, e\rangle \Downarrow ([v]_a, [v]_b)$, which securely evaluate an expression e. $[M]_A$ and $[M]_B$ are the mapping from variables to their secret shares, and t_a and t_b are memory traces from Alice and Bob respectively. All information to evaluate e are contained in $[M]_A$, $[M]_B$, t_a, and t_b. The result is in the format of $([v]_a, [v]_b)$, where $[v]_a$ and $[v]_b$ are secret shares of the result for Alice and Bob respectively. The rules restrict that t_a and t_b must be the memory traces generated by the ideal functionality $\mathcal{F}$ when evaluating e.

Rule SE-Const deals with constant expression n. $([v]_a, [v]_b)$ can be acquired by secret-sharing n, which is implemented using $\mathcal{F}_{+}^{(A,B)}(n, 0)$. Rule SE-Var secret shares the value of a variable expression x. The value of x can be extracted from $[M]_A$ and $[M]_B$, t_a, or t_b according to $\Gamma(x)$. If $\Gamma(x)$ is $\mathtt{P}$ or $\mathtt{A}$, then $t_a = \mathbf{read}(x, v)$, and $[v]_a$ and $[v]_b$ can be computed using $\mathcal{F}_{+}^{(A,B)}(v, 0)$. If $\Gamma(x)$ is $\mathtt{B}$, the computation is similar, but Bob hands in his value v. If $\Gamma(x)$ is $\mathtt{O}$, then $[M]_A(x)$ and $[M]_B(x)$ can be directly returned.

Rule SE-OP handles a binary operation $x\ op\ y$. It can be directly computed using a binary operation functionality $\mathcal{F}_{op}^{(\Gamma(x),\Gamma(y))}$. The input to the functionality is $[M]_A\langle t_a\rangle$ and $[M]_B\langle t_b\rangle$, which is defined as follows. Suppose $[M]$ is a mapping from variables to secret shares, and t is a trace. Then $[M]\langle t\rangle$ is defined inductively as

$$[M]\langle \mathbf{read}(x, v)\rangle = v \quad [M]\langle x\rangle = [M](x) \quad [M]\langle t_1 @ t_2\rangle = [M]\langle t_1\rangle @ [M]\langle t_2\rangle$$

Notice that $[M]\langle t\rangle$ is defined over only $\mathbf{read}(x, v)$, x, and concatenations of them. This is because this notion is used for binary operation and multiplex, where array read events and write events do not occur. The rule SE-MUX for multiplex operation is similar.

For array expression $y[x]$, there are two rules, SE-ArrVar and SE-L-ArrVar. If $\Gamma(y) = \mathtt{O}$, then evaluating $y[x]$ is an ORAM read operation. Rule SE-ArrVar calls the ORAM functionality $\mathcal{F}_{\mathtt{oram}}^{y}$ to get the secret shares $([v]_a, [v]_b)$. Otherwise, $y[x]$ can be computed locally, and rule SE-L-ArrVar handles this case.

The second part of the rules (Figure D.2) are for hybrid protocol, which are in the form of $\langle [M]_A, i_a, t_a\rangle, \langle [M]_B, i_b, t_b\rangle \rightsquigarrow \langle [M']_A, i'_a, t'_a\rangle, \langle [M']_B, i'_b, t'_b\rangle : D$, meaning that Alice and Bob keeping their shares of secret variables, i.e. $[M]_A$ and $[M]_B$ respectively, execute over their simulated traces, i.e. i_a and t_a for Alice, and i_b and t_b for Bob, evaluates to new shares $[M']_A$ and $[M']_B$, and new traces i'_a, t'_a, i'_b and t'_b, and generate declassification D, which is either ϵ or (d_a, d_b).

Rule H-Assign deals with the instruction $\texttt{0} : x := e$. The trace must be in the format of $t_a = t'_a@x$ and $t_b = t'_b@x$, where (t'_a, t'_b) are the memory traces for Alice and Bob to evaluate e. This rule first evaluates the expression e to get $[v]_a$ and $[v]_b$. Then it substitute the mapping for x in $[M]_A$ and $[M]_B$ accordingly.

Rule H-ORAM handles ORAM initialization instruction $\texttt{0} : \mathbf{init}(x, y)$. Either of Alice's or Bob's memory trace must be $\mathbf{readarr}(y, 0, m(0))@...@\mathbf{readarr}(y, l, m(l))@x$, where $l = |m| - 1$. From this trace, one party is able to reconstruct the memory m, which is later fed into ORAM functionality $\mathcal{F}^x_{\texttt{oram}}$ to initialize it.

Rule H-ArrAss handles the instruction $\texttt{0} : y[x_1] := x_2$. First, the secret shares for evaluating x_i are $[v]_{ia}$ and $[v]_{ib}$ for $i = 1, 2$ respectively. Then they are fed into the ORAM functionality $\mathcal{F}^y_{\texttt{oram}}$ to perform a write operation.

Rule H-Cond-While handles $\texttt{0} : \mathbf{if}(x)$ and $\texttt{0} : \mathbf{while}(x)$, which only consumes the corresponding memory traces x, and does not modify $[M]_A$ and $[M]_B$.

Rule H-Declass handles the instruction $\texttt{0} : \mathbf{declass}(x, y)$, which is the only instruction generating non-empty declassification. According to rule S-Declass, both memory traces are y. It calls the declassification functionality $\mathcal{F}^{\Gamma(x)}_{\text{declass}}([M]_A(y), [M]_B(y))$ to release the value of v to the party corresponding to $\Gamma(x)$.

The rules discuss above handles only one instructions. There is a proposition similar to Proposition 1 that holds true for hybrid rules. We start by introducing the concept of consistency of secret-sharing mapping with a memory:

Definition 13. *Given a type environment G, we say a pair of secret share mappings $[M]_A$ and $[M]_B$ is consistent with a G-compatible memory M if and only if for all x such that $\Gamma(x) = \mathsf{0}$, $M(x) = \mathcal{F}^{P}_{declass}([M]_A(x), [M]_B(x))$.*

Now we are ready to present the following proposition.

Proposition 2. *Assuming $\langle M, P\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', P'\rangle : \epsilon$. We use the notation s to denote one element of the set $\{x := e, x[x] := x, \textbf{\textit{if}}(x), \textbf{\textit{while}}(x), \textbf{\textit{init}}(x, y), \textbf{\textit{declass}}(x, y)\}$. If $i_a = i_b = \mathsf{0} : s$, and $[M]_A$ and $[M]_B$ are consistent with M, then*

$$\langle [M]_A, i_a, t_a\rangle, \langle [M]_B, i_b, t_b\rangle \rightsquigarrow \langle [M']_A, i'_a, t'_a\rangle, \langle [M']_B, i'_b, t'_b\rangle : D,$$

and $[M']_A$ and $[M']_B$ are consistent with M'.

The rest four rules deal with multiple instructions. H-Seq and H-Concat are similar to S-Seq and S-Concat correspondingly. H-LocalA and H-LocalB are used to execute local and public instructions.

We first show the hybrid protocol $\pi^{\mathcal{G}}$ generates the same declassification events. This can be easily proved by induction leveraging Proposition 2.

We then show that the hybrid protocol $\pi^{\mathcal{G}}$ securely emulates the ideal world functionality $\mathcal{F}$ (Theorem 5). We suppose Alice is the semi-honest adversary, and Bob's case is symmetric. To show this, the adversary of $\pi^{\mathcal{G}}$ can learn i_a, t_a, a sequence

of secret share mappings $[M]_A, [M']_A, ...$, and declassification events $D_A^1, D_A^2, ...$ In the ideal world, and adversary can learn all the declassification events $D_A^1, ...$, and it can simulate to get i_a and t_a. Further the secret share mappings $[M]_A, [M']_A, ...$ are indistinguishable to random bits. Therefore, the adversary in real world can securely simulates the hybrid world's adversary.

$$\boxed{\langle M, S\rangle \xrightarrow{(i,t)}_A \langle M', S'\rangle}$$

$$\text{Sim-Skip } \langle M, l : \mathbf{skip}; S\rangle \xrightarrow{(\epsilon,\epsilon)}_A \langle M, S\rangle$$

$$\text{Sim-ORAM}\frac{\begin{array}{c} lab(M(x)) = \mathtt{O} \qquad lab(M(y)) = l \\ t = select(l, y, y)@x \qquad i = l : \mathbf{init}(x, y)\end{array}}{\langle M, \mathtt{O} : x := \mathbf{oram}(y)\rangle \xrightarrow{(i,t)}_A \langle M, \mathtt{O} : \mathbf{skip}\rangle}$$

$$\text{Sim-ArrAss}\frac{\begin{array}{c} l \vdash_A \langle M, x_j\rangle \Downarrow_{t_j} v_j \ j = 1, 2 \qquad lab(M(y)) = l \\ l \sqsubseteq \mathtt{A} \Rightarrow M' = M[y \mapsto set(val(M(y)), v_1, v_2)] \\ \mathtt{B} \sqsubseteq l \Rightarrow M' = M \\ t = t_1@t_2@select_A(l, \mathbf{writearr}(y, v_1, v_2), y) \\ i = select_A(l, l : y[x_1] := x_2, l : y[x_1] := x_2)\end{array}}{\langle M, l : y[x_1] := x_2\rangle \xrightarrow{(i,t)}_A \langle M', l : \mathbf{skip}\rangle}$$

$$\text{Sim-Assign}\frac{\begin{array}{c} lab(M(x)) = l \qquad l \vdash_A \langle M, e\rangle \Downarrow_{t'} v \\ l \sqsubseteq \mathtt{A} \Rightarrow M' = M[x \mapsto (v, l')] \\ \mathtt{B} \sqsubseteq l \Rightarrow M' = M \\ t = t'@select_A(l, \mathbf{write}(x, v), x) \\ i = select_A(l, l : x := e, l : x := e)\end{array}}{\langle M, l : x := e\rangle \xrightarrow{(i,t)}_A \langle M', l : \mathbf{skip}\rangle}$$

$$\text{Sim-Cond}\frac{\begin{array}{c} l \vdash_A \langle M, x\rangle \Downarrow_{t'} v \\ v = 1 \Rightarrow c = 1 \qquad v \neq 1 \Rightarrow c = 2 \\ i = select_A(l, l : \mathbf{if}(x), l : \mathbf{if}(x)) \\ t = select_A(l, t', t') \\ l \sqsubseteq \mathtt{A} \Rightarrow S' = S_c \qquad l = \mathtt{B} \Rightarrow S' = \mathtt{P} : \mathbf{skip}\end{array}}{\langle M, l : \mathbf{if}(x)\mathbf{then}\ S_1\mathbf{else}\ S_2\rangle \xrightarrow{(i,t)}_A \langle M, S'\rangle}$$

$$\text{Sim-While-True}\frac{\begin{array}{c} lab(M(x)) = l \qquad l \sqsubseteq \mathtt{A} \\ l \vdash_A \langle M, x\rangle \Downarrow_t v \qquad v \neq 0 \\ S = l : \mathbf{while}(x)\mathbf{do}\ S'\end{array}}{\langle M, S\rangle \xrightarrow{(l:\mathbf{while}(x),t)}_A \langle M, S'; S\rangle}$$

$$\text{Sim-While-False}\frac{\begin{array}{c} lab(M(x)) = l \qquad l \sqsubseteq \mathtt{A} \\ l \vdash_A \langle M, x\rangle \Downarrow_t v \qquad v = 0 \\ S = l : \mathbf{while}(x)\mathbf{do}\ S'\end{array}}{\langle M, S\rangle \xrightarrow{(l:\mathbf{while}(x),t)}_A \langle M, \mathtt{P} : \mathbf{skip}\rangle}$$

$$\text{Sim-While-Ignore}\frac{S = \mathtt{B} : \mathbf{while}(x)\mathbf{do}\ S'}{\langle M, S\rangle \xrightarrow{(\epsilon,\epsilon)}_A \langle M, l : \mathbf{skip}\rangle}$$

Figure C.3: Operational semantics for statements in sim_A (part 2)

$$\boxed{\langle [M]_A, t_a, [M]_B, t_b, e\rangle \Downarrow ([v]_a, [v]_b)}$$

$$\text{SE-Const}\frac{([v]_a, [v]_b) = \mathcal{F}_{+}^{(A,B)}(n, 0)}{\langle [M]_A, \epsilon, [M]_B, \epsilon, n\rangle \Downarrow ([v]_a, [v]_b)}$$

$$\text{SE-Op}\frac{([v]_a, [v]_b) = \mathcal{F}_{op}^{(\Gamma(x),\Gamma(y))}([M]_A\langle t_a\rangle, [M]_B\langle t_b\rangle)}{\langle [M]_A, t_a, [M]_B, t_b, x\ op\ y\rangle \Downarrow ([v]_a, [v]_b)}$$

$$\text{SE-Var}\frac{\begin{array}{c}(t_a, t_b) = \mathit{select}(\Gamma(x), \mathbf{read}(x, v), x)\\ \Gamma(x) \sqsubseteq \mathtt{A} \Rightarrow ([v]_a, [v]_b) = \mathcal{F}_{+}^{(A,B)}(v, 0)\\ \Gamma(x) = \mathtt{B} \Rightarrow ([v]_a, [v]_b) = \mathcal{F}_{+}^{(A,B)}(0, v)\\ \Gamma(x) = \mathtt{O} \Rightarrow ([v]_a, [v]_b) = ([M]_A(x), [M]_B(x))\end{array}}{\langle [M]_A, t_a, [M]_B, t_b, x\rangle \Downarrow ([v]_a, [v]_b)}$$

$$\text{SE-Mux}\frac{([v]_a, [v]_b) = \mathcal{F}_{\mathbf{mux}}^{(\Gamma(x),\Gamma(y),\Gamma(z))}([M]_A\langle t_a\rangle, [M]_B\langle t_b\rangle)}{\langle [M]_A, t_a, [M]_B, t_b, \mathbf{mux}(x, y, z)\rangle \Downarrow ([v]_a, [v]_b)}$$

$$\text{SE-ArrVar}\frac{\begin{array}{c}t_a = t'_a@y \qquad t_b = t'_b@y\\ \langle [M]_A, t'_a, [M]_B, t'_b, x\rangle \Downarrow ([v']_a, [v']_b)\\ ([v]_a, [v]_b) = \mathcal{F}_{\mathtt{oram}}^{y}(\mathtt{read}, [v']_a, [v']_b)\end{array}}{\langle [M]_A, t_a, [M]_B, t_b, y[x]\rangle \Downarrow ([v]_a, [v]_b)}$$

$$\text{SE-L-ArrVar}\frac{\begin{array}{c}t_a = t'_a@t''_a \qquad t_b = t'_b@t''_b \qquad \Gamma(y) \neq \mathtt{O}\\ (t''_a, t''_b) = \mathit{select}(\Gamma(y), \mathbf{readarr}(y, v_1, v), y)\\ \Gamma(y) \sqsubseteq \mathtt{A} \Rightarrow ([v]_a, [v]_b) = \mathcal{F}_{+}^{(A,B)}(v, 0)\\ \Gamma(y) = \mathtt{B} \Rightarrow ([v]_a, [v]_b) = \mathcal{F}_{+}^{(A,B)}(0, v)\end{array}}{\langle [M]_A, t_a, [M]_B, t_b, y[x]\rangle \Downarrow ([v]_a, [v]_b)}$$

Figure D.1: Hybrid Protocol $\pi^{\mathcal{G}}$ (Part I)

$$\boxed{\langle [M]_A, i, t_a\rangle, \langle [M]_B, i, t_b\rangle \rightsquigarrow \langle [M']_A, \epsilon, \epsilon\rangle, \langle [M]'_B, \epsilon, \epsilon\rangle : D}$$

$$\text{H-ORAM}\ \frac{\begin{array}{c} i = \mathtt{0} : \mathbf{init}(x, y) \qquad t_a = t'_a @ x \qquad t_b = t'_b @ x \\ (t'_a, t'_b) = \mathit{select}(\Gamma(y), \mathit{arr}(y, m)) \\ \mathcal{F}^{x}_{\text{oram}}(\mathtt{init}^{\Gamma(y)}, m) \end{array}}{\langle [M]_A, i, t_a\rangle, \langle [M]_B, i, t_b\rangle \rightsquigarrow \langle [M]_A, \epsilon, \epsilon\rangle, \langle [M]_B, \epsilon, \epsilon\rangle : \epsilon}$$

$$\text{H-Assign}\ \frac{\begin{array}{c} i = \mathtt{0} : x := e \qquad \langle [M]_A, t'_a, [M]_B, t'_b, e\rangle \Downarrow ([v]_a, [v]_b) \\ t_a = t'_a @ x \qquad\qquad t_b = t'_b @ x \\ [M']_A = [M]_A[x \mapsto [v]_a] \qquad [M']_B = [M]_A[x \mapsto [v]_b] \end{array}}{\langle [M]_A, i, t_a\rangle, \langle [M]_B, i, t_b\rangle \rightsquigarrow \langle [M']_A, \epsilon, \epsilon\rangle, \langle [M']_B, \epsilon, \epsilon\rangle : \epsilon}$$

$$\text{H-ArrAss}\ \frac{\begin{array}{c} i = \mathtt{0} : y[x_1] := x_2 \qquad t_a = t'_a @ y \qquad t_b = t'_b \\ t_a = t_{1a} @ t_{2a} @ y \qquad\qquad t_b = t_{1b} @ t_{2b} @ y \\ (t_{ia}, t_{ib}) = \mathit{select}(\Gamma(x_i), \mathbf{read}(x_i, v_i), x_i) \\ \langle [M]_A, t'_{ia}, [M]_B, t'_{ib}, x_i\rangle \Downarrow ([v]_{ia}, [v]_{ib}) \qquad i = 1, 2 \\ \mathcal{F}^{y}_{\mathtt{oram}}(\mathtt{write}, [v]_{1a}, [v]_{1b}, [v]_{2a}, [v]_{2b}) \end{array}}{\langle [M]_A, i, t_a\rangle, \langle [M]_B, i, t_b\rangle \rightsquigarrow \langle [M]_A, \epsilon, \epsilon\rangle, \langle [M]_B, \epsilon, \epsilon\rangle : \epsilon}$$

$$\text{H-Cond-While}\ \frac{i = \mathtt{0} : \mathbf{if}(x) \text{ or } i = \mathtt{0} : \mathbf{while}(x) \qquad t_a = t_b = x}{\langle [M]_A, i, t_a\rangle, \langle [M]_B, i, t_b\rangle \rightsquigarrow \langle [M]_A, \epsilon, \epsilon\rangle, \langle [M]_B, \epsilon, \epsilon\rangle : \epsilon}$$

$$\text{H-Declass}\ \frac{\begin{array}{c} i = \mathtt{0} : \mathbf{declass}(x, y) \qquad t_a = t_b = y \\ v = \mathcal{F}^{\Gamma(x)}_{\text{declass}}([M]_A(y), [M]_B(y)) \\ D = \mathit{select}(\Gamma(x), (x, v), (x, v)) \end{array}}{\langle [M]_A, i, t_a\rangle, \langle [M]_B, i, t_b\rangle \rightsquigarrow \langle [M]_A, \epsilon, \epsilon\rangle, \langle [M]_B, \epsilon, \epsilon\rangle : D}$$

$$\text{H-Seq}\ \frac{\begin{array}{c} \langle [M]_A, i'_a, t'_a\rangle, \langle [M]_B, i'_b, t'_b\rangle \rightsquigarrow \langle [M']_A, \epsilon, \epsilon\rangle, \langle [M'], \epsilon, \epsilon\rangle : D \\ i_a = i'_a @ i''_a \qquad\qquad i_b = i'_b @ i''_b \\ t_a = t'_a @ t''_a \qquad\qquad t_b = t'_b @ t''_b \end{array}}{\langle [M]_A, i_a, t_a\rangle, \langle [M]_B, i_b, t_b\rangle \rightsquigarrow \langle [M']_A, i''_a, t''_a\rangle, \langle [M']_B, i''_b, t''_b\rangle : D}$$

$$\text{H-Concat}\ \frac{\begin{array}{c} \langle [M]_A, i_a, t_a\rangle, \langle [M]_B, i_b, t_b\rangle \rightsquigarrow \langle [M']_A, i'_a, t'_a\rangle, \langle [M']_B, i'_b, t'_b\rangle : \epsilon \\ \langle [M']_A, i'_a, t'_a\rangle, \langle [M']_B, i'_b, t'_b\rangle \rightsquigarrow \langle [M'']_A, i''_a, t''_a\rangle, \langle [M'']_B, i''_b, t''_b\rangle : D \end{array}}{\langle [M]_A, i_a, t_a\rangle, \langle [M]_B, i_b, t_b\rangle \rightsquigarrow \langle [M'']_A, i''_a, t''_a\rangle, \langle [M'']_B, i''_b, t''_b\rangle : D}$$

$$\text{H-LocalA}\ \frac{(i, t) \rightarrow \qquad i_a = i @ i'_a \qquad t_a = t @ t'_a}{\langle [M]_A, i_a, t_a\rangle, \langle [M]_B, i_b, t_b\rangle \rightsquigarrow \langle [M]_A, i'_a, t'_a\rangle, \langle [M]_B, i_b, t_b\rangle : \epsilon}$$

$$\text{H-LocalB}\ \frac{(i, t) \rightarrow \qquad i_b = i @ i'_b \qquad t_b = t @ t'_b}{\langle [M]_A, i_a, t_a\rangle, \langle [M]_B, i_b, t_b\rangle \rightsquigarrow \langle [M]_A, i_a, t_a\rangle, \langle [M]_B, i'_b, t'_b\rangle : \epsilon}$$

Figure D.2: Hybrid Protocol $\Pi^{\mathcal{G}}(PartII)$

Bibliography

[1] http://humangenomeprivacy.org/2015.

[2] Graphlab. http://graphlab.org.

[3] Rsa distributed credential protection. http://www.emc.com/security/rsa-distributed-credential-protection.htm.

[4] Trusted computing group. http://www.trustedcomputinggroup.org/.

[5] Private communication, 2014.

[6] Johan Agat. Transforming out timing leaks. In *POPL*, 2000.

[7] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *In POPL*, 1988.

[8] Benny Applebaum. Garbling xor gates "for free" in the standard model. In *TCC*, 2013.

[9] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, 2008.

[10] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153(2):33–55, May 2006.

[11] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *S & P*, 2013.

[12] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIA CCS*, 2013.

[13] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*. 2008.

[14] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *ASIACRYPT*, 2005.

[15] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 2000.

[16] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *Usenix Security Symposium*, 2013.

[17] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-xor" technique. In *TCC*, 2012.

[18] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *ACM Cloud Computing Security Workshop (CCSW)*, pages 85–90, 2009.

[19] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[20] COMPCERT: Compilers you can *formally* trust. http://compcert.inria.fr/.

[21] Convey Computer. The convey HC2 architectural overview. http://www.conveycomputer.com/files/4113/5394/7\097/Convey_HC-2_Architectual_Overview.pdf.

[22] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 45–60, Washington, DC, USA, 2009. IEEE Computer Society.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms, Third Edition*. MIT Press, 2009.

[24] Dana Dachman-Soled, Chang Liu, Charalampos Papamanthou, Elaine Shi, and Uzi Vishkin. Oblivious network RAM. Cryptology ePrint Archive, Report 2015/073, 2015. http://eprint.iacr.org/.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[26] Zhenyue Deng and Geoffrey Smith. Lenient array operations for practical secure information flow. In *Proceedings of the 17th Computer Security Foundations Workshop*, pages 115–124. IEEE, June 2004.

[27] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[28] 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 Cryptology ePrint Archive*, 2014:431, 2014.

[29] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

[30] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-Insensitive Type Qualifiers. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 28(6):1035–1087, November 2006.

[31] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[32] Tanguy Gilmont, Jean didier Legat, and Jean jacques Quisquater. Enhancing security in the memory management unit. In *EUROMICRO*, 1999.

[33] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[34] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[35] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[36] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[37] Michael T. Goodrich and Joseph A. Simons. Data-Oblivious Graph Algorithms in Outsourced External Memory. *CoRR*, abs/1409.0597, 2014.

[38] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[39] S. Dov Gordon, Allen McIntosh, Jonathan Katz, Elaine Shi, and Xiao Shaun Wang. Secure computation of MIPS machine code. Manuscript, 2015.

[40] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[41] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2001.

[42] Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. *Electron. Notes Theor. Comput. Sci.*, 141(1):163–182, December 2005.

[43] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *CCS*, 2010.

[44] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure Two-party Computations in ANSI C. In *CCS*, 2012.

[45] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ansi c. In *CCS*, 2012.

[46] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Usenix Security Symposium*, 2011.

[47] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, 2003.

[48] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO 2003*. 2003.

[49] Jif: Java + information flow. http://www.cs.cornell.edu/jif/.

[50] Anatolii Alexeevitch Karatsuba. The Complexity of Computations, 1995.

[51] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *Asiacrypt*, 2014.

[52] Florian Kerschbaum. Automatically optimizing secure computation. In *CCS*, 2011.

[53] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP*, 2008.

[54] Ben Kreuter, Benjamin Mood, Abhi Shelat, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Usenix Security*, 2013.

[55] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, 2012.

[56] David Lie, John Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings*

of the 2003 IEEE Symposium on Security and Privacy, SP '03, Washington, DC, USA, 2003. IEEE Computer Society.

[57] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[58] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. CSF '13, pages 51–65, 2013.

[59] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating Efficient RAM-model Secure Computation. In *S & P*, May 2014.

[60] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *S & P*, 2015.

[61] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation, 2015. http://www.cs.umd.edu/~elaine/docs/oblivmtr.pdf.

[62] Steve Lu and Rafail Ostrovsky. How to garble ram programs. In *EUROCRYPT*, 2013.

[63] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[64] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[65] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay: a secure two-party computation system. In *USENIX Security*, 2004.

[66] John C. Mitchell and Joe Zimmerman. Data-Oblivious Data Structures. In *STACS*, pages 554–565, 2014.

[67] 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 *ICISC*, 2005.

[68] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *SODA*, 2001.

[69] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. EC '99, 1999.

[70] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S & P*, 2015.

[71] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *CCS*, 2013.

[72] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *S & P*, 2013.

[73] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *TACAS*, 1998.

[74] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.

[75] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *S & P*, 2014.

[76] Aseem Rastogi, Piotr Mardziel, Matthew Hammer, and Michael Hicks. Knowledge inference for optimizing secure multi-party computation. In *PLAS*, 2013.

[77] riscv.org. Launching the Open-Source Rocket Chip Generator, October 2014. https://blog.riscv.org/2014/10/launching-the-open-source-rocket-chip-generator/.

[78] Alejandro Russo, John Hughes, David A. Naumann, and Andrei Sabelfeld. Closing internal timing channels by transformation. In *ASIAN*, 2006.

[79] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[80] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[81] Sergei Skorobogatov. Low temperature data remanence in static RAM. Technical Report UCAM-CL-TR-536, University of Cambridge, Computer Laboratory, June 2002.

[82] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *IEEE S & P*, 2015.

[83] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[84] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

[85] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *International conference on Supercomputing*, ICS '03, pages 160–171, 2003.

[86] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *SIGOPS Oper. Syst. Rev.*, 34(5):168–177, November 2000.

[87] Amit Vasudevan, Jonathan McCune, James Newsome, Adrian Perrig, and Leendert van Doorn. CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, May 2012.

[88] Huy Vo, Yunsup Lee, Andrew Waterman, and Krste Asanović. A Case for OS-Friendly Hardware Accelerators. In *WIVOSCA*, 2013.

[89] Abraham Waksman. A permutation network. *J. ACM*, 15, 1968.

[90] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound.

[91] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *CCS*, 2014.

[92] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *CCS*, 2014.

[93] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[94] Lance Whitney. Microsoft Urges Laws to Boost Trust in the Cloud. http://news.cnet.com/8301-1009_3-10437844-83.html.

[95] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, 1982.

[96] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[97] Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *S & P*, 2013.

[98] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In *CCS*, 2013.